

WebSphere Message Broker



ESQL

Version 6 Release 1

WebSphere Message Broker



ESQL

Version 6 Release 1

Note

Before you use this information and the product that it supports, read the information in the Notices appendix.

This edition applies to version 6, release 1, modification 0, fix pack 6 of IBM WebSphere Message Broker and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 2000, 2010.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this topic collection.	v	ESQL constants.	377
Part 1. Developing ESQL	1	Broker properties that are accessible from ESQL and Java	377
Developing ESQL	3	Special characters, case sensitivity, and comments in ESQL	380
ESQL overview	3	ESQL reserved keywords	381
Managing ESQL files	13	ESQL non-reserved keywords	382
Writing ESQL.	25	Example message	385
Part 2. Reference	157	Part 3. Appendixes	387
ESQL reference	159	Appendix. Notices for WebSphere Message Broker	389
Syntax diagrams: available types	160	Trademarks in the WebSphere Message Broker Information Center	391
ESQL data types in message flows	160	Index	393
ESQL variables	173		
ESQL field reference overview.	174		
ESQL operators.	181		
ESQL statements	188		
ESQL functions: reference material, organized by function type	275		

About this topic collection

This PDF file has been created from the WebSphere Message Broker Version 6.1 (fix pack 6 update, March 2010) information center topics. Always refer to the WebSphere Message Broker online information center to access the most current information. The information center is periodically updated on the document update site and this PDF and others that you can download from that Web site might not contain the most current information.

The topic content included in the PDF does not include the "Related Links" sections provided in the online topics. Links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection. Links to topics outside this topic collection are also shown, but result in a "file not found" error message. Use the online information to navigate freely between topics.

Feedback: do not provide feedback on this PDF. Refer to the online information to ensure that you have access to the most current information, and use the Feedback link that appears at the end of each topic to report any errors or suggestions for improvement. Using the Feedback link provides precise information about the location of your comment.

The content of these topics is created for viewing online; you might find that the formatting and presentation of some figures, tables, examples, and so on are not optimized for the printed page. Text highlighting might also have a different appearance.

Part 1. Developing ESQL

Developing ESQL	3
ESQL overview	3
ESQL data types	4
ESQL variables	4
Broker properties	8
ESQL field references	9
ESQL operators	9
ESQL statements.	10
ESQL functions	11
ESQL procedures	12
ESQL modules	12
Managing ESQL files	13
Creating an ESQL file	14
Opening an existing ESQL file	15
Creating ESQL for a node	16
Modifying ESQL for a node	19
Saving an ESQL file	20
Copying an ESQL file	21
Renaming an ESQL file	21
Moving an ESQL file	22
Changing ESQL preferences	23
Deleting ESQL for a node	24
Deleting an ESQL file	25
Writing ESQL.	25
ESQL examples	26
Tailoring ESQL code for different node types	27
Manipulating message body content	28
Manipulating other parts of the message tree	48
Transforming from one data type to another	60
Adding keywords to ESQL files	67
Interaction with databases using ESQL	67
Coding ESQL to handle errors	79
Using the SELECT function	85
Manipulating messages in the XML domains	99
Manipulating messages in the MRM domain	132
Manipulating messages in the JMS domains	148
Manipulating messages in the IDOC domain	149
Manipulating messages in the MIME domain	150
Manipulating messages in the BLOB domain	152
Using the CALL statement to call a user-written routine	152
Accessing broker properties from ESQL	154
Configuring a message flow at deployment time with user-defined properties	154

Developing ESQL

Customize processing implemented by the Compute, Database, and Filter nodes in your message flows by coding ESQL.

You must create, for each node, an ESQL module in which you code the ESQL statements and functions to tailor the behavior of the node. You can access message content, or database content, or both, to achieve the results that you require. ESQL modules are maintained in ESQL files, managed through the Broker Application Development perspective.

This section provides information about:

- “ESQL overview”
- “Managing ESQL files” on page 13
- “Writing ESQL” on page 25

You can use the ESQL debugger, which is part of the flow debugger, to debug the code that you write. The debugger steps through ESQL code statement by statement, so that you can view and check the results of every line of code that is executed.

ESQL overview

Extended Structured Query Language (ESQL) is a programming language defined by WebSphere® Message Broker to define and manipulate data within a message flow.

This section contains introductory information about ESQL.

- For descriptions of ESQL user tasks, see “Writing ESQL” on page 25.
- For reference information about ESQL, see “ESQL reference” on page 159.

Read the following information before you proceed:

- An overview of message flows in Message flows overview.
- An overview of message trees in The message tree, and the topics within this container, paying special attention to Logical tree structure.

ESQL is based on Structured Query Language (SQL) which is in common usage with relational databases such as DB2®. ESQL extends the constructs of the SQL language to provide support for you to work with message and database content to define the behavior of nodes in a message flow.

The ESQL code that you create to customize nodes within a message flow is defined in an ESQL file, typically named <message_flow_name>.esql,, which is associated with the message flow project. You can use ESQL in the following built-in nodes:

- Compute node
- Database node
- Filter node

You can also use ESQL to create functions and procedures that you can use in the following built-in nodes:

- DataDelete node

- DataInsert node
- DataUpdate node
- Extract node
- Mapping node
- Warehouse node

To use ESQL correctly and efficiently in your message flows, you must also understand the following concepts:

- Data types
- Variables
- Field references
- Operators
- Statements
- Functions
- Procedures
- Modules

Use the ESQL debugger, which is part of the flow debugger, to debug the code that you write. The debugger steps through ESQL code statement by statement, so that you can view and check the results of every line of code that is executed.

ESQL data types

A data type defines the characteristics of an item of data, and determines how that data is processed. ESQL supports six data types, listed below. Data that is retrieved from databases, received in a self-defining message, or defined in a message model (using MRM data types), is mapped to one of these basic ESQL types when it is processed in ESQL expressions.

Within a broker, the fields of a message contain data that has a definite data type. It is also possible to use intermediate variables to help process a message. You must declare all such variables with a data type before use. A variable's data type is fixed; if you try to assign values of a different type you get either an implicit cast or an exception. Message fields do not have a fixed data type, and you can assign values of a different type. The field adopts the new value and type.

It is not always possible to predict the data type that results from evaluating an expression. This is because expressions are compiled without reference to any kind of message schema, and so some type errors are not caught until run time.

ESQL defines the following categories of data. Each category contains one or more data types.

- Boolean
- Datetime
- Null
- Numeric
- Reference
- String

ESQL variables

An ESQL variable is a data field that is used to help process a message.

You must declare a variable and state its type before you can use it. A variable's data type is fixed; if you code ESQL that assigns a value of a different type, either an implicit cast to the data type of the target is implemented or an exception is raised (if the implicit cast is not supported).

To define a variable and give it a name, use the DECLARE statement.

The names of ESQL variables are case sensitive; therefore, make sure that you use the correct case in all places. The simplest way to guarantee that you are using the correct case is always to define variables using uppercase names.

The workbench marks variables that have not been defined. Remove all these warnings before deploying a message flow.

You can assign an initial value to the variable on the DECLARE statement. If an initial value is not specified, scalar variables are initialized with the special value NULL, and ROW variables are initialized to an empty state. Subsequently, you can change the variable's value using the SET statement.

Three types of built-in node can contain ESQL code and therefore support the use of ESQL variables:

- Compute node
- Database node
- Filter node

Variable scope, lifetime, and sharing

How widespread and for how long a particular ESQL variable is available, is described by its scope, lifetime, and sharing:

A variable's scope

is a measure of the range over which it is visible. In the broker environment, the scope of variables is typically limited to the individual node.

A variable's lifetime

is a measure of the time for which it retains its value. In the broker environment, the lifetime of a variable varies but is typically restricted to the life of a thread within a node.

A variable's sharing characteristics

indicate whether each thread has its own copy of the variable or one variable is shared between many threads. In the broker environment, variables are typically not shared.

Types of variable

External

External variables (defined with the EXTERNAL keyword) are also known as *user-defined properties*, see “User-defined properties in ESQL” on page 6. They exist for the entire lifetime of a message flow and are visible to all messages passing through the flow. You can define *external variables* only at the module and schema level. You can modify their initial values (optionally set by the DECLARE statement) at design time, using the Message Flow editor, or at deployment time, using the Broker Archive editor. You can query and set the values of user-defined properties at run time by using the Configuration Manager Proxy (CMP). For more information, see Setting user-defined properties at run time in a CMP application.

Normal

Normal variables variables have a lifetime of just one message passing

through a node. They are visible to that message only. To define a *normal variables*, omit both the EXTERNAL and SHARED keywords.

Shared

Shared variables can be used to implement an in-memory cache in the message flow, see Optimizing message flow response times. *Shared variables* have a long lifetime and are visible to multiple messages passing through a flow, see “Long-lived variables” on page 7. They exist for the lifetime of the execution group process, the lifetime of the flow or node, or the lifetime of the node’s SQL that declares the variable (whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker startup.

See also the ATOMIC option of the “BEGIN ... END statement” on page 191. The BEGIN ATOMIC construct is useful when a number of changes need to be made to a shared variable and it is important to prevent other instances seeing the intermediate states of the data.

For information about specific types of variable, see:

- “User-defined properties in ESQL” (*external variables*)
- “Long-lived variables” on page 7 (*shared variables*)

User-defined properties in ESQL

You can access user-defined properties (UDP) as variables in your ESQL program by specifying the EXTERNAL keyword on a DECLARE statement. For example, the ESQL statement DECLARE today EXTERNAL CHARACTER 'monday' defines a user-defined property called today with an initial value monday.

Before you can use a user-defined property, you must define the property in the Message Flow editor when you construct a message flow that uses it. When you define a UDP in the Message Flow editor, you must define a value and the property type. The value can be a default value, which varies according to the type of the UDP. The value that is assigned to the UDP in the Message Flow editor takes precedence over a value that you have assigned to the UDP in your ESQL program.

Before you deploy the message flow that uses the UDP, you can change the value of the UDP in the Broker Archive editor. If you try to deploy a message flow that contains a UDP that has had no value assigned to it, a deployment failure occurs. For more information, see “Configuring a message flow at deployment time with user-defined properties” on page 154.

You can use UDPs to set configuration data, and use them like typical properties. No external calls to user-written plug-ins or parsing of environment trees are involved, and parsing costs of reading data out of trees are removed. The value of the UDP is finalized in the variable at deployment time.

You can declare UDPs only in modules or schemas. You can query, discover, and set UDPs at run time, to dynamically change the behavior of a message flow. For more information, see User-defined properties.

You can access UDPs from the following built-in nodes that use ESQL:

- Compute
- Database
- Filter

For a description of how to access a UDP from a JavaCompute node, see [Accessing user-defined properties from a JavaCompute node](#).

Long-lived variables

You can use appropriate long-lived ESQL data types to cache data in memory.

Sometimes data has to be stored beyond the lifetime of a single message passing through a flow. One way to store this data is to store the data in a database. Using a database is good for long-term persistence and transactionality, but access (particularly write access) is slow.

Alternatively, you can use appropriate long-lived ESQL data types to provide an in-memory cache of the data for a certain period of time. Using long-lived ESQL data types makes access faster than from a database, although this speed is at the expense of shorter persistence and no transactionality.

You create long-lifetime variables by using the SHARED keyword on the DECLARE statement. For further information, see “[DECLARE statement](#)” on page 235.

The following sample demonstrates how to define shared variables using the DECLARE statement. The sample demonstrates how to store routing information in a database table and use shared variables to store the database table in memory in the message flow to improve performance.

- Message Routing

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Long-lived data types have an extended lifetime beyond that of a single message passing through a node. Long-lived data types are shared between threads and exist for the life of a message flow (strictly speaking the time between configuration changes to a message flow), as described in the following table.

	Scope	Life	Shared
Short lifetime variables			
Schema & Module	Node	Thread within node	Not at all
Routine Local	Node	Thread within routine	Not at all
Block Local	Node	Thread within block	Not at all

Long lifetime variables			
Node Shared	Node	Life of node	All threads of flow
Flow Shared	Flow	Life of flow	All threads of flow

Features of long-lived ESQL data types include:

- The ability to handle large amounts of long-lifetime data.
- The joining of data to messages fast.
- On multiple processor machines, multiple threads can access the same data simultaneously.

- Subsequent messages can access the data left by a previous message.
- Long lifetime read-write data can be shared between threads, because there is no long-term association between threads and messages.
- In contrast to data stored in database tables in the environment, this type of data is stored privately; that is, within the broker.
- ROW variables can be used to create a modifiable copy of the input message; see “ESQL ROW data type” on page 169.
- It is possible to create shared constants.

A typical use of these data types might be in a flow in which data tables are 'read-only' as far as the flow is concerned. Although the table data is not actually static, the flow does not change it, and thousands of messages pass through the flow before there is any change to the table data.

Examples include:

- A table which contains a day's credit card transactions. The table is created each day and that day's messages are run against it. Then the flow is stopped, the table updated and the next day's messages run. These flows might perform better if they cache the table data rather than read it from a database for each message.
- The accumulation and integration of data from multiple messages.

Broker properties

For each broker, WebSphere Message Broker maintains a set of properties. You can access some of these properties from your ESQL programs. A subset of the properties is also accessible from Java™ programs. It can be useful, at run time, to have real-time access to details of a specific node, flow, or broker.

Broker properties are divided into four categories:

- Properties that relate to a specific node
- Properties that relate to nodes in general
- Properties that relate to a message flow
- Properties that relate to the execution group

For a description of the broker, flow, and node properties that are accessible from ESQL and Java, see “Broker properties that are accessible from ESQL and Java” on page 377.

Broker properties have the following characteristics.

- They are grouped by broker, execution group, flow, and node.
- They are case sensitive. Their names always start with an uppercase letter.
- They return NULL if they do not contain a value.

All nodes that allow user programs to edit ESQL support access to broker properties. These nodes are:

- Compute nodes
- Database nodes
- Filter nodes
- All derivatives of these nodes

User-defined properties can be queried, discovered, and set at run time to dynamically change the behavior of a message flow. You can use the Configuration

Manager Proxy (CMP) API to manipulate these properties, which can be used by a systems monitoring tool to perform automated actions in response to situations that it detects in the monitored systems. For more information, see User-defined properties.

A *complex property* is a property to which you can assign multiple values. Complex properties are displayed in a table in the Properties view, where you can add, edit, and delete values, and change the order of the values in the table. You cannot promote complex properties; therefore, they do not appear in the Promote properties dialog box. Nor can you configure complex properties; therefore, they are not supported in the Broker Archive editor. For an example of a complex property, see the Query elements property of the DatabaseRoute node.

For more information about editing a node's properties, see Configuring a message flow node.

ESQL field references

An ESQL field reference is a sequence of period-separated values that identify a specific field (which might be a structure) within a message tree or a database table. The path from the root of the information to the specific field is traced using the parent/child relationships.

A field reference is used in an ESQL statement to identify the field that is to be referenced, updated, or created within the message or database table. For example, you might use the following identifier as a message field reference:

```
Body.Invoice.Payment
```

You can use an ESQL variable of type REFERENCE to set up a dynamic pointer to contain a field reference. This might be useful in creating a fixed reference to a commonly-referenced point within a message; for example the start of a particular structure that contains repeating fields.

A field reference can also specify element types, XML namespace identifications, indexes and a type constraint; see “ESQL field reference overview” on page 174 for further details.

The first name in a field reference is sometimes known as a *Correlation name*.

ESQL operators

An ESQL operator is a character or symbol that you can use in expressions to specify relationships between fields or values.

ESQL supports the following groups of operators:

- Comparison operators, to compare one value to another value (for example, less than). Refer to “ESQL simple comparison operators” on page 181 and “ESQL complex comparison operators” on page 182 for details of the supported operators and their use.
- Logical operators, to perform logical operations on one or two terms (for example, AND). Refer to “ESQL logical operators” on page 185 for details of the supported operators and their use.
- Numeric operators, to indicate operations on numeric data (for example, +). Refer to “ESQL numeric operators” on page 186 for details of the supported operators and their use.

There are some restrictions on the application of some operators to data types; not all lead to a meaningful operation. These are documented where they apply to each operator.

Operators that return a Boolean value (TRUE or FALSE), for example the greater than operator, are also known as predicates.

ESQL statements

An ESQL statement is an instruction that represents a step in a sequence of actions or a set of declarations.

ESQL provides a large number of different statements that perform different types of operation. All ESQL statements start with a keyword that identifies the type of statement and end with a semicolon. An ESQL program consists of a number of statements that are processed in the order they are written.

As an example, consider the following ESQL program:

```
DECLARE x INTEGER;  
SET x = 42;
```

This program consists of two statements. The first starts with the keyword DECLARE and ends at the first semicolon. The second statement starts with the keyword SET and ends at the second semicolon. These two statements are written on separate lines and it is conventional (but not required) that they be so. You will notice that the language keywords are written in capital letters. This is also the convention but is not required; mixed case and lowercase are acceptable.

The first statement declares a variable called x of type INTEGER, that is, it reserves a space in the computer's memory large enough to hold an integer value and allows this space to be subsequently referred to in the program by the name x. The second statement sets the value of the variable x to 42. A number appearing in an ESQL program without decimal point and not within quotes is known as an integer literal.

ESQL has a number of data types and each has its own way of writing literal values. These are described in "ESQL data types" on page 4.

For a full description of all the ESQL statements, see "ESQL statements" on page 188.

ESQL nested statements

An ESQL nested statement is a statement that is contained within another statement.

Consider the following ESQL program fragment:

```
IF Size > 100.00 THEN  
  SET X = 0;  
  SET Y = 0;  
  SET REVERSE = FALSE;  
ELSE  
  SET X = 639;  
  SET Y = 479;  
  SET REVERSE = TRUE;  
END IF;
```

In this example, you can see a single IF statement containing the optional ELSE clause. Both the IF and ELSE portions contain three nested statements. Those within the IF clause are executed if the operator > (greater than) returns the value TRUE (that is, if Size has a value greater than 100.00); otherwise, those within the ELSE clause are processed.

Many statements can have expressions nested within them, but only a few can have statements nested within them. The key difference between an expression and a statement is that an expression calculates a value to be used, whereas a statement performs an action (usually changing the state of the program) but does not produce a value.

ESQL functions

A function is an ESQL construct that calculates a value from a number of given input values.

A function usually has input parameters and can, but does not usually have, output parameters. It returns a value calculated by the algorithm described by its statement. This statement is usually a compound statement, such as BEGIN... END, because this allows an unlimited number of nested statements to be used to implement the algorithm.

ESQL provides a number of predefined, or “built-in”, functions which you can use freely within expressions. You can also use the CREATE FUNCTION statement to define your own functions.

When you define a function, you must give it a unique name. The name is handled in a case insensitive way (that is, use of the name with any combination of uppercase and lowercase letters matches the declaration). This is in contrast to the names that you declare for schemas, constants, variables, and labels, which are handled in a case sensitive way, and which you must specify exactly as you declared them.

Consider the following ESQL program fragment:

```
SET Diameter = SQRT(Area / 3.142) * 2;
```

In this example, the function SQRT (square root) is given the value inside the brackets (itself the result of an expression, a divide operation) and its result is used in a further expression, a multiply operation. Its return value is assigned to the variable Diameter. See “Calling ESQL functions” on page 278 for information about all the built-in ESQL functions.

In addition, an ESQL expression can refer to a function in another broker schema (that is, a function defined by a CREATE FUNCTION statement in an ESQL file in the same or in a different dependent project). To resolve the name of the called function, you must do one of the following:

- Specify the fully-qualified name (<SchemaName>.<FunctionName>) of the called function.
- Include a PATH statement to make all functions from the named schema visible. Note that this technique only works if the schemas do not contain identically-named functions. The PATH statement must be coded in the same ESQL file, but not within any MODULE.

Note that you cannot define a function within an EVAL statement or an EVAL function.

ESQL procedures

An ESQL procedure is a subroutine that has no return value. It can accept input parameters from, and return output parameters to, the caller.

Procedures are very similar to functions. The main difference between them is that, unlike functions, procedures have no return value. Thus they cannot form part of an expression and are invoked by using the CALL statement. Procedures commonly have output parameters

You can implement a procedure in ESQL (an internal procedure) or as a database stored procedure (an external procedure). The ESQL procedure must be a single ESQL statement, although that statement can be a compound statement such as BEGIN END. You cannot define a procedure within an EVAL statement or an EVAL function.

When you define a procedure, give it a name. The name is handled in a case insensitive way (that is, use of the name with any combination of uppercase and lowercase letters matches the declaration). That is in contrast to the names that you declare for schemas, constants, variables, and labels, which are handled in a case sensitive way, and which you must specify exactly as you declared them.

An ESQL expression can include a reference to a procedure in another broker schema (defined in an ESQL file in the same or a different dependent project). If you want to use this technique, either fully qualify the procedure, or include a PATH statement that sets the qualifier. The PATH statement must be coded in the same ESQL file, but not within a MODULE.

An external database procedure is indicated by the keyword EXTERNAL and the external procedure name. This procedure must be defined in the database and in the broker, and the name specified with the EXTERNAL keyword and the name of the stored database procedure must be the same, although parameter names do not have to match. The ESQL procedure name can be different from the external name it defines.

Overloaded procedures are not supported to any database. (An overloaded procedure is one that has the same name as another procedure in the same database schema which has a different number of parameters, or parameters with different types.) If the broker detects that a procedure has been overloaded, it raises an exception.

Dynamic schema name resolution for stored procedures is supported; when you define the procedure you must specify a wildcard for the schema that is resolved before invocation of the procedure by ESQL. This is explained further in “Invoking stored procedures” on page 77.

ESQL modules

A module is a sequence of declarations that define variables and their initialization, and a sequence of subroutine (function and procedure) declarations that define a specific behavior for a message flow node.

A module must begin with the `CREATE node_type MODULE` statement and end with an `END MODULE` statement. The `node_type` must be one of `COMPUTE`, `DATABASE`, or `FILTER`. The entry point of the ESQL code is the function named `MAIN`, which has `MODULE` scope.

Each module is identified by a name which follows `CREATE node_type MODULE`. The name might be created for you with a default value, which you can modify, or you can create it yourself. The name is handled in a case insensitive way (that is, use of the name with any combination of uppercase and lowercase letters matches the declaration). That is in contrast to the names that you declare for schemas, constants, variables, and labels, which are handled in a case sensitive way, and which you must specify exactly as you declared them.

You must create the code for a module in an ESQL file which has a suffix of `.esql`. You must create this file in the same broker schema as the node that references it. There must be one module of the correct type for each corresponding node, and it is specific to that node and cannot be used by any other node.

When you create an ESQL file (or complete a task that creates one), you indicate the message flow project and broker schema with which the file is associated as well as specifying the name for the file.

Within the ESQL file, the name of each module is determined by the value of the corresponding property of the message flow node. For example, the property *ESQL Module* for the Compute node specifies the name of the node's module in the ESQL file. The default value for this property is the name of the node. You can specify a different name, but you must ensure that the value of the property and the name of the module that provides the required function are the same.

The module must contain the function `MAIN`, which is the entry point for the module. This is included automatically if the module is created for you. Within `MAIN`, you can code ESQL to configure the behavior of the node. If you include ESQL within the module that declares variables, constants, functions, and procedures, these are of local scope only and can be used within this single module.

If you want to reuse ESQL constants, functions, or procedures, you must declare them at broker schema level. You can then refer to these from any resource within that broker schema, in the same or another project. If you want to use this technique, either fully qualify the procedure, or include a `PATH` statement that sets the qualifier. The `PATH` statement must be coded in the same ESQL file, but not within any `MODULE`.

Managing ESQL files

Within a message flow project, you can create ESQL files to contain the ESQL code that you provide to modify or customize the behavior of Compute, Database, or Filter nodes.

The ESQL code is contained within a module that is associated with the node. Each module must be created within an ESQL file. The name of the module within the ESQL file must match the name specified for the module in the *ESQL Module* property of the corresponding node. Although you can modify the module name, and change it from its default value (which is the name of the message flow,

concatenated with the name of the node with which the module is associated), ensure that the module in the ESQL file matches the node property.

The following topics describe how you can manage these files:

- “Creating an ESQL file”
- “Opening an existing ESQL file” on page 15
- “Creating ESQL for a node” on page 16
- “Modifying ESQL for a node” on page 19
- “Saving an ESQL file” on page 20
- “Copying an ESQL file” on page 21
- “Renaming an ESQL file” on page 21
- “Moving an ESQL file” on page 22
- “Changing ESQL preferences” on page 23
- “Deleting ESQL for a node” on page 24
- “Deleting an ESQL file” on page 25

Creating an ESQL file

When you include a node in your message flow that requires ESQL to customize its function (the Compute, Database, and Filter nodes), you must code the ESQL statements that provide the customization in an ESQL module within an ESQL file. You can use the same ESQL file for more than one module.

Before you start

To complete this task, you must have completed the following task:

- Creating a message flow project

ESQL files are stored in a file system or in a shared repository. If you are using a file system, this can be the local file system or a shared drive. If you store files in a repository, you can use any of the available repositories that are supported by Eclipse, for example CVS.

To create an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Click **File** → **New** → **Message Flow ESQL File**.

You can also press Ctrl+N. This displays a dialog box that allows you to select the wizard to create a new object. Click Message Brokers in the left view; the right view displays a list of objects that you can create for WebSphere Message Broker. Click Message Flow ESQL File in the right view, then click **Next**. The New Message Flow ESQL File wizard is displayed.

3. Enter the name of the message flow project in which to create the ESQL file. You must enter the name of an existing message flow project. The dialog box is displayed with the current project name entered in the project name field. You can accept this value or change it to specify a different project. You can also click **Browse** to view a list of valid projects (projects that are defined and displayed in the Navigator view), and select the appropriate value from that list. The list is filtered to only show projects in the active working set.

If you type in the name of a project that does not exist, the error message The specified project does not exist is displayed in the dialog box and you cannot continue until you specify a valid project name.

4. If you want the ESQL file to be defined within a specific broker schema, enter the name of the broker schema in the appropriate entry field, or click **Browse**

to select the broker schema from the list of valid broker schema for this project. (If only the default broker schema is defined in this project, **Browse** is disabled.)

5. Enter a name for the new ESQL file. If you enter a name that is already in use for an ESQL file in this project, the error message The resource <name>.esql already exists is displayed in the dialog box and you cannot continue until you specify a valid name.

When creating ESQL files, the overall file path length must not exceed 256 characters, due to a Windows file system limitation. If you try to add a message flow to a broker archive file with ESQL or mapping files with a path length that exceeds 256 characters, the compiled message flow will not be generated and cannot be deployed. Therefore, make sure that the names of your ESQL files, mapping files, projects, and broker schema are as short as possible.

An ESQL file can also be created automatically for you. If you select Open ESQL from the menu displayed when you right-click a Compute, Database, or Filter node, and the module identified by the appropriate property does not already exist within the broker schema, a module is automatically created for you. This is created in the file <message_flow_name>.esql in the same broker schema within the same project as the <message_flow_name>.msgflow file. If that ESQL file does not already exist, that is also created for you.

The contents of a single ESQL file do not have any specific relationship with message flows and nodes. It is your decision which modules are created in which files (unless the specified module, identified by the appropriate property, is created by default in the file <message_flow_name>.esql as described above). Monitor the size and complexity of the ESQL within each file, and split the file if it becomes difficult to view or manage.

If you create reusable subroutines (at broker schema level) within an ESQL file, you might want to refer to these routines from ESQL modules in another project. To do this, specify that the project that wants to invoke the subroutines depends on the project in which the ESQL file containing them is defined. You can specify this when you create the second project, or you can update project dependencies by selecting the project, clicking **Properties**, and updating the dependencies in the Project Reference page of the Properties dialog box.

Opening an existing ESQL file

You can add to and modify ESQL code that you have created in an ESQL file in a message flow project.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 14

To open an existing ESQL file:

1. Switch to the Broker Application Development perspective.
2. In the Broker Development view, double-click the ESQL file that you want to open. The file is opened in the editor view.
3. Work with the contents of file to make your changes. The file can contain modules relating to specific nodes in a message flow, PATH statements, and

declarations at broker schema level such as reusable constants and procedures. Scroll through the file to find the specific content that you want to work with.

4. You can select the content that you want to work with by selecting its name in the Outline view. The code for the selected resource is highlighted.

You can also open an ESQL file when you have a message flow open in the editor view by selecting an appropriate node (of type Compute, Database, or Filter), right-clicking, and selecting **Open ESQL**. In this case, the ESQL file that contains this module is opened, and the module for the selected node is highlighted in the editor view.

Creating ESQL for a node

Create ESQL to customize the behavior of a Compute, Database, or Filter node within an ESQL file.

Before you start

Complete the following task:

- “Creating an ESQL file” on page 14

Within the ESQL file, create a module that is associated with a node in your message flow. A module can be associated with only one node of a particular type (Compute, Database, or Filter). Within the module you can create and use functions and procedures as well as the supplied statements and functions. You can also create local constants and variables.

If you have created constants, functions, or procedures at the broker schema level, you can also refer to these within the module. You can define routines at a level at which many different modules can use them, which can save you development time and maintenance effort.

To create ESQL for a node:

1. Switch to the Broker Application Development perspective.
2. In the Broker Development view, double-click the message flow that includes the node for which you want to create ESQL. The message flow opens in the editor view.
3. Right-click the node (which must be Compute, Database, or Filter) and then click **Open ESQL**. The default ESQL file for this message flow, *message_flow_name.esql*, is opened in the editor view. The file is created if it does not already exist.

If you have already created the file, it is opened in the editor view and a new module is created and highlighted. If the file is created for you, it contains a skeleton module for this node at the end. Its exact content depends on the type of node.

The following module is created for a Compute node:


```

CREATE COMPUTE MODULE module_name
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    -- CALL CopyMessageHeaders();
    -- CALL CopyEntireMessage();
    RETURN TRUE;
  END;

  CREATE PROCEDURE CopyMessageHeaders() BEGIN
    DECLARE I INTEGER 1;
    DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
    WHILE I < J DO
      SET OutputRoot.*[I] = InputRoot.*[I];
      SET I = I + 1;
    END WHILE;
  END;

  CREATE PROCEDURE CopyEntireMessage() BEGIN
    SET OutputRoot = InputRoot;
  END;
END MODULE;

```

The module name is determined by the value that you have set for the corresponding node property. The default is *message_flow_name_node_type*. The Main function contains calls to two procedures, described below, that are declared within the Compute node module following the function Main. These calls are commented out. If you want to include the function that they provide, uncomment the lines and place them at the appropriate point in the ESQL that you create for Main.

CopyMessageHeaders

This procedure loops through the headers contained in the input message and copies each one to the output message.

CopyEntireMessage

This procedure copies the entire contents of the input message, including the headers, to the output message.

If you create an ESQL module for a Database node, the following module is created:

```

CREATE DATABASE MODULE module_name
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    RETURN TRUE;
  END;
END MODULE;

```

For a Filter node, the module is identical to that created for the Database node except for the first line, which reads:

```

CREATE FILTER MODULE module_name

```

4. Add ESQL to this file to customize the behavior of the node.

Start by adding ESQL statements within the Main function, that is after the BEGIN statement, and before RETURN TRUE. You can add DECLARE statements within the module that are not within the Main function. To add a new line into the file, press Enter.

To help you to code valid ESQL, the editor displays a list of valid statements and functions at the point of the cursor. To invoke this assistance, click **Edit** → **Content Assist**. On some systems, you can use the key combination Ctrl+Space.

Scroll through the list displayed to find and highlight the one that you want, and press Enter. The appropriate code is inserted into your module, and the list disappears.

Content assistance is provided in the following areas:

- Applicable keywords, based on language syntax.
- Blocks of code that go together, such as `BEGIN END;`.
- Constants that you have defined, identifiers, labels, functions, and procedures that can be used, where the routines can be in any projects, even if the current project does not reference them.
- Database schema and table names after the database correlation name, as well as table column names in `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements, and, in most cases, the `WHERE` clauses of those statements.
- Elements of message field reference: runtime domain (parser) names, format of type expression, namespace identifiers, namespace-qualified element and attribute names, and format of index expression.
- Content in the Properties folder under the output message root.
- For the `DECLARE NAMESPACE` statement, target namespaces of message sets and schema names.

Content assistance works only if the ESQL can be parsed correctly. Errors such as `END` missing after `BEGIN`, and other unterminated block statements, cause parser failures and no content assistance is provided. Try content assistance in other areas around the statement where it does not work to narrow down the point of error. Alternatively, save the ESQL file; saving the file causes validation, and all syntax errors are written to the Problems view. Refer to the errors reported to understand and correct the ESQL syntax. If you use content assistance to generate most statements (such as block statements), these are correctly entered and there is less opportunity for error.

5. When you have finished working with this module, you can close the ESQL file. Save the file before you close it to retain all your changes and validate your ESQL.

If you prefer, you can open the ESQL file directly and create the module within that file using the editor:

1. Switch to the Broker Application Development perspective.
2. Select the ESQL file in which you want to create the module. Either double-click to open this file in the editor view, or right-click and click **Open**.
3. In the editor view, position your cursor on a new line and use content assistance to select the appropriate module skeleton for this type of node, for example `CREATE COMPUTE MODULE END MODULE;`. You can type this text in yourself if you prefer, but you must ensure that what you type is consistent with the required skeleton, shown earlier. Use content assistance to give you additional help by inserting only valid ESQL, and by inserting matching end statements (for example, `END MODULE;`) where these are required.
4. Complete the coding of the module as appropriate.

Whichever method you use to open the ESQL file, be aware that the editor provides functions to help you to code ESQL. This section refers to content assistance; other functions are available. For information about these functions, see ESQL editor.

Modifying ESQL for a node

If you want to change the customization of a node that requires ESQL (Compute, Database, or Filter), you can modify the ESQL statements within the module that you created for that node.

Before you start

To complete this task, you must have completed the following task:

- “Creating ESQL for a node” on page 16

To modify ESQL code:

1. Switch to the Broker Application Development perspective.
2. In the Broker Development view, select the message flow that you want to work with and double-click it. The message flow is opened in the editor view.
3. Right-click the node corresponding to the ESQL module that you want to modify and click **Open ESQL**. The ESQL file is opened in the editor view. The module for this node is highlighted.
4. Make the changes that you want in the module, by entering new statements (remember that you can use Content Assist, available from the Edit menu or, on some systems, by pressing Ctrl+Space), changing existing statements by overtyping, or deleting statements using the Delete or backspace keys. Note that, to get Content Assist to work with message references, you must set up a project reference from the project containing the ESQL to the project containing the message set. For information about setting up a project reference, see Project references.
5. You can change the name of the module that you are working with, by over-typing the current name with the new one. Remember that, if you do that, you must also change the node property *ESQL Module* to reflect the new name to ensure that the correct ESQL code is deployed with the node.
6. When you have finished working with this module, you can close the ESQL file. Save the file before you close it to retain all your changes and validate your ESQL.

If you prefer, you can open the ESQL file directly by double-clicking it in the Broker Development view. You can select the module that you want to work with from the Outline view.

The editor provides functions that you can use to help you modify your ESQL code. These functions are described in ESQL editor.

You can also modify the ESQL source by selecting **Source** → **Format**. This option formats all selected lines of code (unless only partially selected, when they are ignored), or, if no lines are selected, formats the entire file (correcting alignments and indentation).

Adding comments to ESQL

You can add comments to and remove comments from your ESQL code:

1. To change an existing line of code into a comment line, click **Source** → **Comment**.
2. To change a comment line to a code line, click **Source** → **Uncomment**.

3. To create a new comment line, press Enter to create a new line and either type the comment identifier -- or click **Source** → **Comment**. You can enter any text after the identifier: everything you type is ignored by the ESQL editor.

Saving an ESQL file

When you edit your ESQL file, you can save it both to preserve the additions and modifications that you have made, and to force the editor to validate the content of the file.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 14

To save an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Create an ESQL file, or open an existing ESQL file.
3. Change the contents of the ESQL file.
4. When you have finished working, save the file to retain all your changes by clicking **File** → **Save filename.esql** or **File** → **Save All** (the menu always shows the current filename correctly).


When you save the file, the validator is called by the editor to check that the ESQL obeys all grammar and syntax rules (specified by the syntax diagrams and explanations in “ESQL reference” on page 159).

You can request additional validation when you set ESQL preferences:


- a. Click **Window** → **Preferences**. The Preferences dialog is displayed.
- b. Expand the item for **Broker Development** on the left, expand **ESQL**, and click **Validation**.
- c. Select the level of validation that you require for each category of error:
 - 1) Unresolved identifiers
 - 2) Message references do not match message definitions
 - 3) Database references do not match database schema
 - 4) Use of deprecated keywords

The default level is **Warning**; you can change this value to **Error** or **Ignore**.

Validating message definitions can affect response times in the editor, particularly if you have complicated ESQL that makes many references to a complex message definition. You might choose to delay this validation. Call validation when you have finished developing the message flow and are about to deploy it, to avoid runtime errors. For each error found, the editor writes the code line number and the reason for the error; errors are created as entries in the Problems view.

5. If you double-click the error, the editor positions your cursor on the line in which it found that error. The line is also highlighted by the error icon  in the margin to the left.

The editor might also find potential error situations, that it highlights as

warnings (with the warning icon ); it also writes these warnings to the Problems view. For example, you might have included a BROKER SCHEMA statement that references an invalid schema (namespace).

Check your code, and make the corrections required by that statement or function.

Save As

You can save a copy of this ESQL file by using **File** → **Save As**.

1. Click **File** → **Save name As**.
2. Specify the message flow project in which you want to save a copy of the ESQL file. The project name defaults to the current project. You can accept this name, or choose another name from the valid options that are displayed in the File Save dialog.
3. Specify the name for the new copy of the ESQL file. If you want to save this ESQL file in the same project, you must either give it another name, or confirm that you want to overwrite the current copy (that is, copy the file to itself).
If you want to save this ESQL file in another project, the project must exist (you can only select from the list of existing projects). You can save the file with the same or another name in another project.
4. Click **OK**. The message flow is saved and the message flow editor validates its contents. The editor provides a report of all errors that it finds in the Problems view.

Copying an ESQL file

You might find it useful to copy an ESQL file as a starting point for a new ESQL file that has similar function.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 14

To copy an ESQL file:

1. Switch to the Broker Application Development perspective.
2. In the Broker Development view, select the ESQL file (`<message_flow_name>.esql`) that you want to copy. Right-click the file and click **Copy** from the menu.
3. Right-click the broker schema within the message flow project to which you want to copy the ESQL file and click **Paste**. You can copy the ESQL file to the same broker schema within the same message flow project, or to a different broker schema within the same message flow project, or to a broker schema in a different message flow project.

When you copy an ESQL file, the associated files (message flow, and mapping if present) are not automatically copied to the same target message flow project. If you want these files copied as well, you must do this explicitly following this procedure.

If you want to use this ESQL file with another message flow, ensure that the modules within the ESQL file match the nodes that you have in the message flow, and that the node properties are set correctly.

You can also use **File** → **Save As** to copy an ESQL file. This is described in “Saving an ESQL file” on page 20.

Renaming an ESQL file

You can rename an ESQL file within the message flow project. You might want to do this, for example, if you have renamed the message flow that it is associated with.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 14

To rename an ESQL file:

1. Switch to the Broker Application Development perspective.
2. In the Broker Development view, right-click the ESQL file that you want to rename. Its default name is <message_flow_name>.esql. Click **Rename** or click **File** → **Rename**. If you have selected the ESQL file, you can press F2. The Rename Resource dialog is displayed.
3. Enter the new name for the ESQL file. Click **OK** to complete the action, or **Cancel** to cancel the request. If you click **OK**, the ESQL file is renamed.

Moving an ESQL file



If you move a message flow from one broker schema to another, or from one project to another, you might want to move all ESQL files that are associated with that message flow.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 14

To move an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Move the ESQL file in one of the following ways:
 - a. Drag the ESQL file that you want to move from its current location to a broker schema within the same or another message flow project.
If the target location that you have chosen is not valid (for example, if an ESQL file of this name exists in the broker schema), the invalid icon is displayed and the move is not completed.
 - b. Right-click the ESQL file and click **Move**, or click **File** → **Move**. The Move dialog is displayed.
Select the project and the broker schema from the list of valid targets that is shown in the dialog.
Click **OK** to complete the move, or **Cancel** to cancel the request.
If you click **OK**, the ESQL file is moved to its new location.
3. Check the Problems view for errors (indicated by the error icon ) or warnings (indicated by the warning icon ) generated by the move.
The errors in the Problems view include those errors caused by broken references. When the move is completed, all references to this ESQL file are checked. If you have moved the file within the same named broker schema within the same message flow project, all references are still valid. If you have moved the file to another broker schema in the same or another message flow project, the references are broken. If you have moved the file to the same named broker schema in another message flow project, the references might be broken if the project references are not set correctly to recognize external references in this file. These errors occur because resources are linked by a fully qualified name.

4. Double-click each error or warning to open the message flow that has the error in the editor view, and highlight the node in error. You can now correct the error.

When you move an ESQL file, its associated files (for example, the message flow file) are not automatically moved to the same target broker schema. You must move these files yourself.

Changing ESQL preferences

You can modify the way in which ESQL is displayed in the editor and validated by the editor:

- “Changing ESQL editor settings”
- “Changing ESQL validation settings” on page 24

Changing ESQL editor settings

When you open an ESQL file in the editor view, you can tailor the editor appearance by changing editor settings.

1. To change ESQL editor settings:

- a. Switch to the Broker Application Development perspective.
- b. Click **Window** → **Preferences**. The Preferences dialog box opens.
- c. Expand the item for ESQL on the left and click **ESQL Editor**.
- d. Update the settings available for tab width and colors:
 - Click the **General** tab to change the displayed tab width within the ESQL editor.
 - Click the **Colors** tab to change the color of the editor view background, and of the entities displayed in the editor view. These include comments and keywords in your ESQL code.
- e. When you have completed your changes, click either **Apply** (to apply your changes and leave the Preferences dialog box open), or **OK** (to apply your changes and close the dialog box). Alternatively, to close the dialog box and discard your changes, click **Cancel**.
- f. To return the ESQL editor settings to the initial values, click **Restore Defaults**. All values are reset to the original settings.

If you change the editor settings when you have an editor session active, the changes are implemented immediately. If you do not have an editor session open, you see the changes when you next edit an ESQL file.

2. To change font settings for the ESQL editor:

- a. Click **Window** → **Preferences**. The Preferences dialog box opens.
- b. Expand the item for Workbench on the left of the Preferences dialog box, and click **Colors and Fonts**.
- c. On the **Colors and Fonts** tab, expand Basic.
- d. Select a font or text color option and click **Change**. The Font dialog box opens.
- e. When you have completed your changes, click either **Apply** (to apply your changes and leave the Preferences dialog box open) or **OK** (to apply your changes and close the dialog box). Alternatively, to close the dialog box and discard your changes, click **Cancel**.
- f. To return the ESQL editor settings to the initial values, click **Restore Defaults**.

Changing ESQL validation settings

You can specify the level of validation that the ESQL editor performs when you save a .esql file. If the validation you have requested results in warnings, you can deploy a BAR file containing this message flow. However, if errors are reported, you cannot deploy the BAR file.

To change ESQL validation settings:

1. Switch to the Broker Application Development perspective.
2. Click **Window** → **Preferences**. The Preferences dialog is displayed.
3. Expand the item for ESQL on the left and click Validation.
4. Update the settings for what is validated, and for what warnings or errors are reported. See ESQL editor for details of the settings and their values.
5. When you have completed your changes, click **Apply** to close the Preferences dialog, apply your changes and leave the Preferences dialog open. Click **OK** to apply your changes and close the dialog. Click **Cancel** to close the dialog and discard your changes.
6. If you want to return your ESQL editor preferences to the initial values, click **Restore Defaults**. All values are reset to the original settings.

If you make changes to the validation settings, the changes are implemented immediately for currently open edit sessions and for subsequent edit sessions.

Deleting ESQL for a node

If you delete a node from a message flow, you can delete the ESQL module that you created to customize its function.

Before you start

To complete this task, you must have completed the following task:

- “Creating ESQL for a node” on page 16

To delete ESQL code:

1. Switch to the Broker Application Development perspective.
2. Open the message flow that you want to work with by double-clicking it in the Broker Development view. The message flow is opened in the editor view.
3. Select the node for which you want to delete the ESQL module, right-click and click **Open ESQL**. The ESQL file is opened in the editor view, with the module for this node highlighted.
4. Press the Delete or backspace key to delete the whole module.
5. When you have finished working with this module, you can close the ESQL file. Save the file before you close it to retain all your changes. Save also validates your ESQL: see “Saving an ESQL file” on page 20.

If you prefer, you can open the ESQL file directly by double-clicking it in the Broker Development view. The ESQL file is opened in the editor view. Select the module that you want to delete from the Outline view and delete it as described above. You can also right-click on the module name in the Broker Development view (the modules in the ESQL file are visible if you expand the view of the file by clicking the + beside the file name) and click **Delete**.

Deleting an ESQL file

If you delete a message flow, or if you have deleted all the ESQL code in an ESQL file, you can delete the ESQL file.

Before you start

To complete this task, you must have completed the following task:

- “Creating an ESQL file” on page 14

To delete an ESQL file:

1. Switch to the Broker Application Development perspective.
2. Within the Broker Development view, right-click the ESQL file that you want to delete, and click **Delete**. A dialog is displayed that asks you to confirm the deletion.

You can also select the file in the Broker Development view, and click **Edit** → **Delete**. A dialog is displayed that asks you to confirm the deletion.

3. Click **Yes** to delete the file, or **No** to cancel the delete request.

If you maintain resources in a shared repository, a copy is retained in that repository. You can follow the instructions provided by the repository supplier to retrieve the file if required.

If you are using the local file system or a shared file system to store your resources, no copy of the file is retained. Be careful to select the correct file when you complete this task.

Writing ESQL

How you can use ESQL to customize nodes.

When you create a message flow, you include input nodes that receive the messages and, optionally, output nodes that send out new or updated messages. If required by the processing that must be performed on the message, you can include other nodes after the input node that complete the actions that your applications need.

Some of the built-in nodes enable you to customize the processing that they provide. The Compute, Database, and Filter nodes require you to provide a minimum level of ESQL, and you can provide much more than the minimum to control precisely the behavior of each node. This set of topics discusses ESQL and the ways in which you can use it to customize these nodes.

The DataDelete, DataInsert, DataUpdate, Extract, Mapping, and Warehouse nodes provide a mapping interface with which you can customize their function. The ways in which you can use the mapping functions associated with these nodes are described in developing message mappings, see Developing message mappings.

ESQL provides a rich and flexible syntax for statements and functions that enable you to check and manipulate message and database content. You can:

- Read the contents of the input message
- Modify message content with data from databases
- Modify database content with data from messages
- Construct new output messages created from all, part, or none of the input message (in the Compute node only)

The following topics provide more information about these and other tasks that you can perform with ESQL. Unless otherwise stated, these guidelines apply to messages in all message domains except the BLOB domain, for which you can implement a limited set of actions.

- “Tailoring ESQL code for different node types” on page 27
- “Manipulating message body content” on page 28
- “Manipulating other parts of the message tree” on page 48
- “Transforming from one data type to another” on page 60
- “Adding keywords to ESQL files” on page 67
- “Interaction with databases using ESQL” on page 67
- “Coding ESQL to handle errors” on page 79
- “Accessing broker properties from ESQL” on page 154
- “Configuring a message flow at deployment time with user-defined properties” on page 154

The following topics provide additional information specific to the parser that you have specified for the input message:

- “Manipulating messages in the MRM domain” on page 132
- “Manipulating messages in the XML domain” on page 132
- “Manipulating messages in the XMLNS domain” on page 122
- “Manipulating messages in the XMLNSC domain” on page 109
- “Manipulating messages in the JMS domains” on page 148
- “Manipulating messages in the IDOC domain” on page 149
- “Manipulating messages in the MIME domain” on page 150
- “Manipulating messages in the BLOB domain” on page 152

ESQL examples

Most of the examples included in the topics listed previously show parser-independent ESQL. If examples include a reference to MRM, they assume that you have modeled the message in the MRM and that you have set the names of the MRM objects to be identical to the names of the corresponding tags or attributes in the XML source message. Some examples are also shown for the XML domain. Unless stated otherwise, the principals illustrated are the same for all message domains. For domain-specific information, use the appropriate link in the previous list.

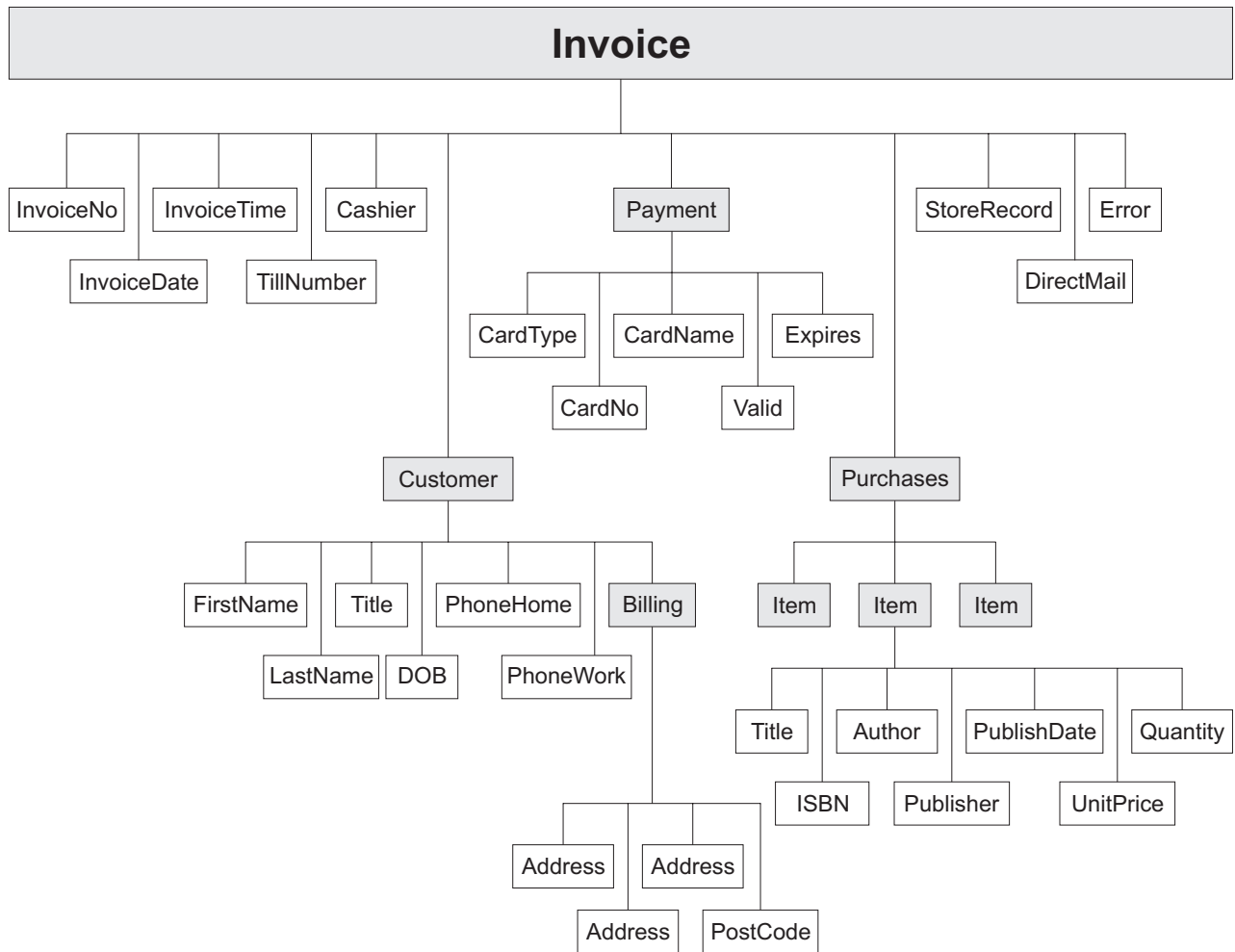
Most of the topics that include example ESQL use the ESQL sample message, *Invoice*, as the input message to the logic. This message is provided in XML source format (with tags and attributes), see “Example message” on page 385. The example message is shown in the following diagram.

The topics specific to the MRM domain use the message that is created in the following sample:

- Video Rental

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

A few other input messages are used to show ESQL that provides function on messages with a structure or content that is not included in the *Invoice* or *Video* samples. Where this occurs, the input message is included in the topic that refers to it.



Tailoring ESQL code for different node types

When you code ESQL to configure Compute, Database, and Filter node behavior, be aware of the limitations of each type of node.

Compute node

You can configure the Compute node to do any of the following operations:

- Update data in a database.
- Insert data into a database.
- Delete data from a database.
- Update the environment tree.
- Update the local environment tree.
- Create one or more output messages, with none, some, or all the content of the input message, and propagate these new messages to the next node in the message flow.

To propagate the input LocalEnvironment to the output LocalEnvironment, remember to set the Compute node property Compute mode to an appropriate value. The Environment is always propagated in the output message.

Database node

You can configure the Database node to do any of the following operations:

- Update data in a database.
- Insert data into a database.
- Delete data from a database.
- Update the environment tree.
- Update the local environment tree.
- Propagate the input message to the next node in the message flow.

Filter node

You can configure the Filter node to do any of the following operations:

- Update data in a database.
- Insert data into a database.
- Delete data from a database.
- Update the environment tree.
- Update the local environment tree.
- Propagate the input message to the next node in the message flow (the terminal through which the message is propagated depends on the result of the filter expression).

View the remaining tasks in this section to find the details of how you can perform these operations.

Manipulating message body content

The message body is always the last child of root, and is identified by its parser name, for example XML or MRM. You can refer to, modify, and create message body data.

The following topics describe how you can refer to, modify, and create message body data. The information provided here is domain independent.

- “Referencing field types”
- “Accessing elements in the message body” on page 29
- “Accessing known multiple occurrences of an element” on page 33
- “Accessing unknown multiple occurrences of an element” on page 34
- “Using anonymous field references” on page 35
- “Creating dynamic field references” on page 36
- “Creating new fields” on page 37
- “Generating multiple output messages” on page 39
- “Using numeric operators with datetime values” on page 40
- “Calculating a time interval” on page 41
- “Selecting a subfield from a larger field” on page 42
- “Copying repeating fields” on page 43
- “Manipulating repeating fields in a message tree” on page 47

Referencing field types

Some message parsers have complex models in which it is not enough to identify a field simply by its name and an array subscript. In these cases, you associate an optional field type with an element of data in the tree format.

Each element within the parsed tree can be one of three types:

Name element

A name element has a string, which is the name of the element, associated with it. An example of a name element is `XMLElement`, described in `XML element`.

Value element

A value element has a value associated with it. An example of a value element is `XMLContent`, described in `XML content`.

Name-value element

A name-value element is an optimization of the case where a name element contains only a value element and nothing else. The element contains both a name and a value. An example of a name-value element is `XMLAttribute`, described in `XML attribute`.

Accessing elements in the message body

When you want to access the contents of a message, for reading or writing, use the structure and arrangement of the elements in the tree that is created by the parser from the input bit stream.

Follow the relevant parent and child relationships from the top of the tree downwards, until you reach the required element.

- If you are referring to the input message tree to interrogate its content in a Compute node, use correlation name `InputBody` followed by the path to the element to which you are referring. `InputBody` is equivalent to `InputRoot` followed by the parser name (for example, `InputRoot.MRM`), which you can use if you prefer.
- If you are referring to the output message tree to set or modify its content in the Compute node, use correlation name `OutputRoot` followed by the parser name (for example, `OutputRoot.MRM`).
- If you are referring to the input message to interrogate its contents in a Database or Filter node, use correlation name `Body` to refer to the start of the message. `Body` is equivalent to `Root` followed by the parser name (for example, `Root.XMLNS`), which you can use if you prefer.

You must use these different correlation names because there is only one message to which to refer in a Database or Filter node; you cannot create an output message in these nodes. Use a Compute node to create an output message.

When you construct field references, the names that you use must be valid ESQL identifiers that conform to ESQL rules. If you enclose an item in double quotation marks, ESQL interprets it as an identifier. If you enclose an item in single quotation marks, ESQL interprets it as a character literal. You must enclose all strings (character strings, byte strings, or binary (bit) strings) in quotation marks, as shown in the following examples. To include a single or double quotation mark within a string, include two consecutive single or double quotation marks.

Important: For a full description of field reference syntax, see “ESQL field reference overview” on page 174.

For more information about ESQL data types, see “ESQL data types in message flows” on page 160.

Assume that you have created a message flow that handles the message Invoice, shown in the figure in “Writing ESQL” on page 25. If, for example, you want to interrogate the element CardType from within a Compute node, use the following statement:

```
IF InputBody.Invoice.Payment.CardType='Visa' THEN
  DO;
    -- more ESQL --
END IF;
```

If you want to make the same test in a Database or Filter node (where the reference is to the single input message), code:

```
IF Body.Invoice.Payment.CardType='Visa' THEN
  DO;
    -- more ESQL --
END IF;
```

If you want to copy an element from an input XML message to an output message in the Compute node without changing it, use the following ESQL:

```
SET OutputRoot.XMLNS.Invoice.Customer.FirstName =
    InputBody.Invoice.Customer.FirstName;
```

If you want to copy an element from an input XML message to an output message and update it, for example by folding to uppercase or by calculating a new value, code:

```
SET OutputRoot.XMLNS.Invoice.Customer.FirstName =
    UPPER(InputBody.Invoice.Customer.FirstName);
SET OutputRoot.XMLNS.Invoice.InvoiceNo = InputBody.Invoice.InvoiceNo + 1000;
```

If you want to set a STRING element to a constant value, code:

```
SET OutputRoot.XMLNS.Invoice.Customer.Title = 'Mr';
```

You can also use the equivalent statement:

```
SET OutputRoot.XMLNS.Invoice.Customer.Title VALUE = 'Mr';
```

If you want to update an INTEGER or DECIMAL, for example the element TillNumber, with the value 26, use the following assignment (valid in the Compute node only):

```
SET OutputRoot.MRM.Invoice.TillNumber=26;
```

The integer data type stores numbers using the 64-bit twos complement form, allowing numbers in the range -9223372036854775808 to 9223372036854775807. You can specify hexadecimal notation for integers as well as normal integer literal format. The hexadecimal letters A to F can be written in uppercase or lowercase, as can the X after the initial zero, which is required. The following example produces the same result as the example shown earlier:

```
SET OutputRoot.MRM.Invoice.TillNumber= 0x1A;
```

The following examples show SET statements for element types that do not appear in the example Invoice message.

To set a FLOAT element to a non-integer value, code:

```
SET OutputRoot.MRM.FloatElement1 = 1.2345e2;
```

To set a BINARY element to a constant value, code:

```
SET OutputRoot.MRM.BinaryElement1 = X'F1F1';
```

For BINARY values, you must use an initial character X (uppercase or lowercase) and enclose the hexadecimal characters (also uppercase or lowercase) in single quotation marks, as shown.

To set a BOOLEAN element to a constant value (the value 1 equates to true, 0 equates to false), code:

```
SET OutputRoot.MRM.BooleanElement1 = true;
```

or

```
SET OutputRoot.MRM.BooleanElement1 = 1;
```

You can use the SELECT statement to filter records from an input message without reformatting the records, and without any knowledge of the complete format of each record. Consider the following example:

```
-- Declare local variable
DECLARE CurrentCustomer CHAR 'Smith';

-- Loop through the input message
SET OutputRoot.XMLNS.Invoice[] =
  (SELECT I FROM InputRoot.XMLNS.Invoice[] AS I
   WHERE I.Customer.LastName = CurrentCustomer
  );
```

This code writes all records from the input Invoice message to the output message if the WHERE condition (LastName = Smith) is met. All records that do not meet the condition are not copied from input message to output message. I is used as an alias for the correlation name InputRoot.XMLNS.Invoice[].

The declared variable CurrentCustomer is initialized on the DECLARE statement: this option is the most efficient way of declaring a variable for which the initial value is known.

You can use this alias technique with other SELECT constructs. For example, if you want to select all the records of the input Invoice message, and create an additional record:

```
-- Loop through the input message
SET OutputRoot.XMLNS.Invoice[] =
  (SELECT I, 'Customer' || I.Customer.LastName AS ExtraField
   FROM InputRoot.XMLNS.Invoice[] AS I
  );
```

You could also include an AS clause to place records in a subfolder in the message tree:

```
-- Loop through the input message
SET OutputRoot.XMLNS.Invoice[] =
  (SELECT I AS Order
   FROM InputRoot.XMLNS.Invoice[] AS I
  );
```

If you are querying or setting elements that contain, or might contain, null values, be aware of the following considerations:

Querying null values

When you compare an element to the ESQL keyword `NULL`, this tests whether the element is present in the logical tree that has been created from the input message by the parser.

For example, you can check if an invoice number is included in the current Invoice message with the following statement:

```
IF InputRoot.XMLNS.Invoice.InvoiceNo IS NULL THEN
  DO;
  -- more ESQL --
END IF;
```

You can also use an ESQL reference, as shown in the following example:

```
DECLARE cursor REFERENCE TO InputRoot.MRM.InvoiceNo;

IF LASTMOVE(cursor) = FALSE THEN
  SET OutputRoot.MRM.Analysis = 'InvoiceNo does not exist in logical tree';
ELSEIF FIELDVALUE(cursor) IS NULL THEN
  SET OutputRoot.MRM.Analysis =
    'InvoiceNo does exist in logical tree but is defined as an MRM NULL value';
ELSE
  SET OutputRoot.MRM.Analysis = 'InvoiceNo does exist and has a value';
END IF;
```

For more information about declaring and using references, see “Creating dynamic field references” on page 36. For a description of the `LASTMOVE` and `FIELDVALUE` functions, see “`LASTMOVE` function” on page 326 and “`FIELDTYPE` function” on page 321.

If the message is in the MRM domain, there are additional considerations for querying null elements that depend on the physical format. For further details, see “Querying null values in a message in the MRM domain” on page 141.

Setting null values

You can use two statements to set null values:

1. If you set the element to `NULL` by using the following statement, the element is deleted from the message tree:

```
SET OutputRoot.XMLNS.Invoice.Customer.Title = NULL;
```

If the message is in the MRM domain, there are additional considerations for null values that depend on the physical format. For further details, see “Setting null values in a message in the MRM domain” on page 142.

This technique is called implicit null processing.

2. If you set the value of this element to `NULL` as follows:

```
SET OutputRoot.XMLNS.Invoice.Customer.Title VALUE = NULL;
```

the element is not deleted from the message tree. Instead, a special value of `NULL` is assigned to the element.

```
SET OutputRoot.XMLNS.Invoice.Customer.Title = NULL;
```


If the message is in the MRM domain, the content of the output bit stream depends on the settings of the physical format null handling properties. For further details, see “Setting null values in a message in the MRM domain” on page 142.

This technique is called explicit null processing.

If you set an MRM complex element or an XML, XMLNS, or JMS parent element to NULL without using the VALUE keyword, that element and all its children are deleted from the logical tree.

Accessing known multiple occurrences of an element

When you refer to or create the content of messages, it is very likely that the data contains repeating fields. If you know how many instances there are of a repeating field, and you want to access a specific instance of such a field, you can use an array index as part of a field reference.

For example, you might want to filter on the first line of an address, to expedite the delivery of an order. Three instances of the element Billing.Address are always present in the sample message. To test the first line, write an expression such as:

```
IF Body.Invoice.Customer.Billing.Address[1] = 'Patent Office' THEN
  DO;
  -- more ESQL --
END IF;
```

The array index [1] indicates that it is the first instance of the repeating field that you are interested in (array indices start at 1). An array index such as this can be used at any point in a field reference, so you could, for example, filter on the following test:

```
IF Body.Invoice."Item"[1].Quantity > 2 THEN
  DO;
  -- more ESQL --
END IF;
```

You can refer to the last instance of a repeating field using the special [<] array index, and to instances relative to the last (for example, the second to last) as follows:

- Field[<] indicates the last element.
- Field[<1] indicates the last element.
- Field[<2] indicates the last but one element (the penultimate element).

You can also use the array index [>] to represent the first element, and elements relative to the first element in a similar way.

- Field[>] indicates the first element. This is equivalent to Field[1].

The following examples refer to the Invoice message using these indexes:

```
IF Body.Invoice.Customer.Billing.Address[<] = 'Hampshire' THEN
  DO;
  -- more ESQL --
END IF;
IF Body.Invoice.Customer.Billing.Address[<2] = 'Southampton' THEN
  DO;
  -- more ESQL --
END IF;
```

You can also use these special indexes for elements that repeat an unknown number of times.

Deleting repeating fields:

If you pass a message with several repeats of an element through a message flow and you want to delete some of the repeats, be aware that the numbering of the repeats is reordered after each delete. For example, if you have a message with five repeats of a particular element, and in the message flow you have the following ESQL:

```
SET OutputRoot.MRM.e_PersonName[1] = NULL;  
SET OutputRoot.MRM.e_PersonName[4] = NULL;
```

You might expect elements one and four to be deleted. However, because repeating elements are stored on a stack, when you delete one, the one above it takes its place. This means that, in the above example, elements one and five are deleted. To avoid this problem, delete in reverse order, that is, delete element four first, then delete element one.

Accessing unknown multiple occurrences of an element

To access repeating fields in a message, you must use a construct that can iterate over all instances of a repeating field.

You are likely to deal with messages that contain repeating fields with an unknown number of repeats (such as the `Item` field in “Example message” on page 385).

To write a filter that takes into account all instances of the `Item` field, you need to use a construct that can iterate over all instances of a repeating field. The quantified predicate allows you to execute a predicate against all instances of a repeating field, and collate the results.

For example, you might want to verify that none of the items that are being ordered has a quantity greater than 50. To do this you could write:

```
FOR ALL Body.Invoice.Purchases."Item" []  
  AS I (I.Quantity <= 50)
```

With the quantified predicate, the first thing to note is the brackets `[]` on the end of the field reference after `FOR ALL`. These tell you that you are iterating over all instances of the `Item` field.

In some cases, this syntax appears unnecessary because you can get that information from the context, but it is done for consistency with other pieces of syntax.

The `AS` clause associates the name `I` with the current instance of the repeating field. This is similar to the concept of iterator classes used in some object oriented languages such as C++. The expression in parentheses is a predicate that is evaluated for each instance of the `Item` field.

A description of this example is:

Iterate over all instances of the field `Item` inside `Body.Invoice`. For each iteration:

1. Bind the name `I` to the current instance of `Item`.

2. Evaluate the predicate `I.Quantity <= 50`. If the predicate:
 - Evaluates to TRUE for all instances of `Item`, return TRUE.
 - Is FALSE for any instance of `Item`, return FALSE.
 - For a mixture of TRUE and UNKNOWN, return UNKNOWN.

The above is a description of how the predicate is evaluated if you use the ALL keyword. An alternative is to specify SOME, or ANY, which are equivalent. In this case the quantified predicate returns TRUE if the sub-predicate returns TRUE for any instance of the repeating field. Only if the sub-predicate returns FALSE for all instances of the repeating field does the quantified predicate return FALSE. If a mixture of FALSE and UNKNOWN values are returned from the sub-predicate, an overall value of UNKNOWN is returned.

In the following filter expression:

```
FOR ANY Body.Invoice.Purchases."Item" []  
  AS I (I.Title = 'The XML Companion')
```

the sub-predicate evaluates to TRUE. However this next expression returns FALSE:

```
FOR ANY Body.Invoice.Purchases."Item" []  
  AS I (I.Title = 'C Primer')
```

because the C Primer is not included on this invoice. If some of the items in the invoice do not include a book title field, the sub-predicate returns UNKNOWN, and the quantified predicate returns the value UNKNOWN.

To deal with the possibility of null values appearing, write this filter with an explicit check on the existence of the field, as follows:

```
FOR ANY Body.Invoice.Purchases."Item" []  
  AS I (I.Book IS NOT NULL AND I.Book.Title = 'C Primer')
```

The predicate IS NOT NULL ensures that, if an Item field does not contain a Book, a FALSE value is returned from the sub-predicate.

You can also manipulate arbitrary repeats of fields within a message by using a SELECT expression, as described in “Referencing columns in a database” on page 68.

You can refer to the first and last instances of a repeating field using the [`>`] and [`<`] array indexes, and to instances relative to the first and last, even if you do not know how many instances there are. These indexes are described in “Accessing known multiple occurrences of an element” on page 33.

Alternatively, you can use the CARDINALITY function to determine how many instances of a repeating field there are. For example:

```
DECLARE I INTEGER CARDINALITY(Body.Invoice.Purchases."Item" [])
```

Using anonymous field references

You can refer to the array of all children of a particular element by using a path element of `*`. So, for example:

```
InputRoot.*[]
```

is a path that identifies the array of all children of InputRoot. This is often used in conjunction with an array subscript to refer to a particular child of an entity by position, rather than by name. For example:

InputRoot.*[<]

Refers to the last child of the root of the input message, that is, the body of the message.

InputRoot.*[1]

Refers to the first child of the root of the input message, the message properties.

You might want to find out the name of an element that has been identified with a path of this kind. To do this, use the FIELDNAME function, which is described in “FIELDNAME function” on page 320.

Creating dynamic field references

You can use a variable of type REFERENCE as a dynamic reference to navigate a message tree. This acts in a similar way to a message cursor or a variable pointer. It is generally simpler and more efficient to use reference variables in preference to array indexes when you access repeating structures. Reference variables are accepted everywhere. Field references are accepted and come with a set of statements and functions to allow detailed manipulation of message trees.

You must declare a dynamic reference before you can use it. A dynamic reference is declared and initialized in a single statement. The following example shows how to create and use a reference.

```
-- Declare the dynamic reference
DECLARE myref REFERENCE TO OutputRoot.XMLNS.Invoice.Purchases.Item[1];

-- Continue processing for each item in the array
WHILE LASTMOVE(myref)=TRUE
DO
-- Add 1 to each item in the array
  SET myref = myref + 1;
-- Move the dynamic reference to the next item in the array
  MOVE myref NEXTSIBLING;
END WHILE;
```

This example declares a dynamic reference, *myref*, which points to the first item in the array within Purchases. The value in the first item is incremented by one, and the pointer (dynamic reference) is moved to the next item. Once again the item value is incremented by one. This process continues until the pointer moves outside the scope of the message array (all the items in this array have been processed) and the LASTMOVE function returns FALSE.

The examples below show further examples.

```
DECLARE ref1 REFERENCE TO InputBody.Invoice.Purchases.Item[1];

DECLARE ref2 REFERENCE TO
  InputBody.Invoice.Purchases.NonExistentField;

DECLARE scalar1 CHARACTER;
DECLARE ref3 REFERENCE TO scalar1;
```

In the second example, *ref2* is set to point to InputBody because the specified field does not exist.

With the exception of the MOVE statement, which changes the position of the dynamic reference, you can use a dynamic reference anywhere that you can use a static reference. The value of the dynamic reference in any expression or statement is the value of the field or variable to which it currently points. For example, using the message in “Example message” on page 385, the value of Invoice.Customer.FirstName is Andrew. If the dynamic reference myref is set to point at the FirstName field as follows:

```
DECLARE myref REFERENCE TO Invoice.Customer;
```

the value of myref is Andrew. You can extend this dynamic reference as follows:

```
SET myref.Billing.Address[1] = 'Oaklands';
```

This changes the address in the example to Oaklands Hursley Village Hampshire SO213JR.

The position of a dynamic reference remains fixed even if a tree is modified. To illustrate this point the steps that follow use the message in “Example message” on page 385 as their input message and create a modified version of this message as an output message:

1. Copy the input message to the output message.
2. To modify the output message, first declare a dynamic reference ref1 that points at the first item, The XML Companion.

```
DECLARE ref1 REFERENCE TO  
OutputRoot.XMLNS.Invoice.Purchases.Item[1];
```

The dynamic reference is now equivalent to the static reference
OutputRoot.XMLNS.Invoice.Purchases.Item[1].

3. Use a create statement to insert a new first item for this purchase.

```
CREATE PREVIOUSIBLING OF ref1 VALUES 'Item';
```

The dynamic reference is now equivalent to the static reference
OutputRoot.XMLNS.Invoice.Purchases.Item[2].

Creating new fields

You can use a Compute node to create a new output message by adding new fields to an existing input message.

This topic provides example ESQL code for a Compute node that creates a new output message based on the input message, to which are added a number of additional fields.

The input message received by the Compute node within the message flow is an XML message, and has the following content:

```
<TestCase description="This is my TestCase">  
  <Identifier>ES03B305_T1</Identifier>  
  <Sport>Football</Sport>  
  <Date>01/02/2000</Date>  
  <Type>LEAGUE</Type>  
</TestCase>
```

The Compute node is configured and an ESQL module is created that includes the following ESQL. The code shown below copies the headers from the input message

to the new output message, then creates the entire content of the output message body.

```
-- copy headers
DECLARE i INTEGER 1;
DECLARE numHeaders INTEGER CARDINALITY(InputRoot.*[]);

WHILE i < numHeaders DO
    SET OutputRoot.*[i] = InputRoot.*[i];
    SET i = i + 1;
END WHILE;

CREATE FIELD OutputRoot.XMLNS.TestCase.description TYPE NameValue VALUE 'This is my TestCase';
CREATE FIRSTCHILD OF OutputRoot.XMLNS.TestCase Domain('XMLNS') NAME 'Identifier'
    VALUE InputRoot.XMLNS.TestCase.Identifier;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase Domain('XMLNS') NAME 'Sport'
    VALUE InputRoot.XMLNS.TestCase.Sport;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase Domain('XMLNS') NAME 'Date'
    VALUE InputRoot.XMLNS.TestCase.Date;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase Domain('XMLNS') NAME 'Type'
    VALUE InputRoot.XMLNS.TestCase.Type;
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Number TYPE NameValue
    VALUE 'Premiership';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Result[1].Number TYPE NameValue VALUE '1';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Result[1].Home TYPE Name;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[1].Home NAME 'Team'
    VALUE 'Liverpool' ;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[1].Home NAME 'Score'
    VALUE '4';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Result[1].Away TYPE Name;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[1].Away NAME 'Team'
    VALUE 'Everton';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[1].Away NAME 'Score'
    VALUE '0';

CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Result[2].Number TYPE NameValue VALUE '2';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Result[2].Home TYPE Name;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[2].Home NAME 'Team'
    VALUE 'Manchester United';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[2].Home NAME 'Score'
    VALUE '2';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[1].Result[2].Away TYPE Name;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[2].Away NAME 'Team'
    VALUE 'Arsenal';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[1].Result[2].Away NAME 'Score'
    VALUE '3';

CREATE FIELD OutputRoot.XMLNS.TestCase.Division[2].Number TYPE NameValue
    VALUE '2';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[2].Result[1].Number TYPE NameValue
    VALUE '1';
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[2].Result[1].Home TYPE Name;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[2].Result[1].Home NAME 'Team'
    VALUE 'Port Vale';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[2].Result[1].Home NAME 'Score'
    VALUE '9' ;
CREATE FIELD OutputRoot.XMLNS.TestCase.Division[2].Result[1].Away TYPE Name;
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[2].Result[1].Away NAME 'Team'
    VALUE 'Brentford';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Division[2].Result[1].Away NAME 'Score'
    VALUE '5';
```

The output message that results from the ESQL shown above has the following structure and content:

```

<TestCase description="This is my TestCase">
  <Identifier>ES03B305_T1</Identifier>
  <Sport>Football</Sport>
  <Date>01/02/2000</Date>
  <Type>LEAGUE</Type>
  <Division Number="Premiership">
    <Result Number="1">
      <Home>
        <Team>Liverpool</Team>
        <Score>4</Score>
      </Home>
      <Away>
        <Team>Everton</Team>
        <Score>0</Score>
      </Away>
    </Result>
    <Result Number="2">
      <Home>
        <Team>Manchester United</Team>
        <Score>2</Score>
      </Home>
      <Away>
        <Team>Arsenal</Team>
        <Score>3</Score>
      </Away>
    </Result>
  </Division>
  <Division Number="2">
    <Result Number="1">
      <Home>
        <Team>Port Vale</Team>
        <Score>9</Score>
      </Home>
      <Away>
        <Team>Brentford</Team>
        <Score>5</Score>
      </Away>
    </Result>
  </Division>
</TestCase>

```

Generating multiple output messages

You can use the PROPAGATE statement to generate multiple output messages in the Compute node. The output messages that you generate can have the same or different content. You can also direct output messages to any of the four alternate output terminals of the Compute node, or to a Label node.

For example, to create three copies of the input message received by the Compute node, and send one to the standard Out terminal of the Compute node, one to the first alternate Out1 terminal of the Compute node, and one to the Label node ThirdCopy, code the following ESQL:

```

SET OutputRoot = InputRoot;
PROPAGATE;
SET OutputRoot = InputRoot;
PROPAGATE TO TERMINAL 'out1';
SET OutputRoot = InputRoot;
PROPAGATE TO LABEL 'ThirdCopy';

```

In the above example, the content of OutputRoot is reset before each PROPAGATE, because by default the node clears the output message buffer and reclaims the memory when the PROPAGATE statement completes. An alternative method is to

instruct the node not to clear the output message on the first two PROPAGATE statements, so that the message is available for routing to the next destination. The code to achieve this result is:

```
SET OutputRoot = InputRoot;
PROPAGATE DELETE NONE;
SET OutputRoot = InputRoot;
PROPAGATE TO TERMINAL 'out1' DELETE NONE;
SET OutputRoot = InputRoot;
PROPAGATE TO LABEL 'ThirdCopy';
```

If you do not initialize the output buffer, an empty message is generated, and the message flow detects an error and throws an exception.

Also ensure that you copy all required message headers to the output message buffer for each output message that you propagate.

If you want to modify the output message content before propagating each message, code the appropriate ESQL to make the changes that you want before you code the PROPAGATE statement.

If you set up the contents of the last output message that you want to generate, and propagate it as the final action of the Compute node, you do not have to include the final PROPAGATE statement. The default action of the Compute node is to propagate the contents of the output buffer when it terminates. This is implemented by the RETURN TRUE statement, included as the final statement in the module skeleton.

For example, to generate three copies of the input message, and not perform any further action, include this code immediately before the RETURN TRUE statement:

```
SET OutputRoot = InputRoot;
PROPAGATE DELETE NONE;
PROPAGATE DELETE NONE;
```

Alternatively, you can modify the default behavior of the node by changing RETURN TRUE to RETURN FALSE:

```
SET OutputRoot = InputRoot;
PROPAGATE DELETE NONE;
PROPAGATE DELETE NONE;
PROPAGATE;
RETURN FALSE;
```

Three output messages are generated by the three PROPAGATE statements. The final RETURN FALSE statement causes the node to terminate but not propagate a final output message. Note that the final PROPAGATE statement does not include the DELETE NONE clause, because the node must release the memory at this stage.

Using numeric operators with datetime values

The following examples show the ESQL that you can code to manipulate datetime values with numeric operators.

Adding an interval to a datetime value

The simplest operation that you can perform is to add an interval to, or subtract an interval from, a datetime value. For example, you could write the following expressions:


```
DATE '2000-03-29' + INTERVAL '1' MONTH  
TIMESTAMP '1999-12-31 23:59:59' + INTERVAL '1' SECOND
```

The following example shows how to calculate a retirement date by adding the retirement age to the birth date.

```
DECLARE retAge CHARACTER '65';  
DECLARE birthDate DATE DATE '1953-06-01';  
  
SET OutputRoot.XML.Test.retirementDate = birthDate + CAST(retAge AS INTERVAL YEAR);
```

The repetition of the word DATE in the above example is intentional. The first occurrence of DATE specifies the data type of the declared variable, birthDate. The second occurrence initializes the same variable with the constant character string that is enclosed in quotes as a DATE.

Adding or subtracting two intervals

You can combine two interval values by using addition or subtraction. The two interval values must be of compatible types. It is not valid to add a year-month interval to a day-second interval as shown in the following example:

```
INTERVAL '1-06' YEAR TO MONTH + INTERVAL '20' DAY
```

The interval qualifier of the resultant interval is sufficient to encompass all the fields that are present in the two operand intervals. For example:

```
INTERVAL '2 01' DAY TO HOUR + INTERVAL '123:59' MINUTE TO SECOND
```

results in an interval with qualifier DAY TO SECOND, because both day and second fields are present in at least one of the operand values.

Subtracting two datetime values

You can subtract two datetime values to return an interval. You must include an interval qualifier in the expression to indicate what precision the result should be returned in. For example:

```
(CURRENT_DATE - DATE '1776-07-04') DAY
```

returns the number of days since the 4th July 1776, whereas:

```
(CURRENT_TIME - TIME '00:00:00') MINUTE TO SECOND
```

returns the age of the day in minutes and seconds.

Scaling intervals

You can multiply or divide an interval value by an integer factor:

```
INTERVAL '2:30' MINUTE TO SECOND / 4
```

Calculating a time interval

You can use ESQL to calculate the time interval between two events, and to set a timer to be triggered after a specified interval.

This ESQL example calculates the time interval between an input WebSphere MQ message being put on the input queue, and the time that it is processed in the current Compute node.

(When you make a call to a CURRENT_ datetime function, the value that is returned is identical to the value returned to another call in the same node. This ensures that you can use the function consistently within a single node.)

```
CALL CopyMessageHeaders();
Declare PutTime INTERVAL;

SET PutTime = (CURRENT_GMTIME - InputRoot.MQMD.PutTime) MINUTE TO SECOND;

SET OutputRoot.XMLNS.Test.PutTime = PutTime;
```

The output message has the format (although actual values vary):

```
<Test>
  <PutTime>INTERVAL &apos;1:21.862&apos; MINUTE TO SECOND</PutTime>
</Test>
```

The following code snippet sets a timer, to be triggered after a specified interval from the start of processing, in order to check that processing has completed. If processing has not completed within the elapsed time, the firing of the timer might, for example, trigger some recovery processing.

The StartTime field of the timeout request message is set to the current time plus the allowed delay period, which is defined by a user-defined property on the flow. (The user-defined property has been set to a string of the form "HH:MM:SS" by the administrator.)

```
DECLARE StartDelyIntervalStr EXTERNAL CHARACTER '01:15:05';

CREATE PROCEDURE ValidateTimeoutRequest() BEGIN

  -- Set the timeout period
  DECLARE timeoutStartTimeRef REFERENCE TO
    OutputRoot.XMLNSC.Envelope.Header.TimeoutRequest.StartTime;
  IF LASTMOVE(timeoutStartTimeRef)
  THEN
    -- Already set
  ELSE
    -- Set it from the UDP StartDelyIntervalStr
    DECLARE startAtTime TIME CURRENT_TIME
      + CAST(StartDelyIntervalStr AS INTERVAL HOUR TO SECOND);

    -- Convert "TIME 'hh.mm.ss.fff'" to hh.mm.ss format
    -- needed in StartTime field
    DECLARE startAtTimeStr CHAR;
    SET startAtTimeStr = startAtTime;
    SET startAtTimeStr = SUBSTRING(startAtTimeStr FROM 7 FOR 8);
    SET OutputRoot.XMLNSC.Envelope.Header.TimeoutRequest.StartTime
      = startAtTimeStr;
  END IF;
END;
```

Selecting a subfield from a larger field

You might have a message flow that processes a message containing delimited subfields. You can code ESQL to extract a subfield from the surrounding content if you know the delimiters of the subfield.

If you create a function that performs this task, or a similar one, you can invoke it both from ESQL modules (for Compute, Database, and Filter nodes) and from mapping files (used by DataDelete, DataInsert, DataUpdate, Extract, Mapping, and Warehouse nodes).

The following function example extracts a particular subfield of a message that is delimited by a specific character.

```

CREATE FUNCTION SelectSubField
  (SourceString CHAR, Delimiter CHAR, TargetStringPosition INT)
  RETURNS CHAR
-- This function returns a substring at parameter position TargetStringPosition within the
-- passed parameter SourceString. An example of use might be:
-- SelectSubField(MySourceField,' ',2) which will select the second subfield from the
-- field MySourceField delimited by a blank. If MySourceField has the value
-- "First Second Third" the function will return the value "Second"
BEGIN
  DECLARE DelimiterPosition INT;
  DECLARE CurrentFieldPosition INT 1;
  DECLARE StartNewString INT 1;
  DECLARE WorkingSource CHAR SourceString;
  SET DelimiterPosition = POSITION(Delimiter IN SourceString);
  WHILE CurrentFieldPosition < TargetStringPosition
  DO
    IF DelimiterPosition = 0 THEN
      -- DelimiterPosition will be 0 if the delimiter is not found
      -- exit the loop
      SET CurrentFieldPosition = TargetStringPosition;
    ELSE
      SET StartNewString = DelimiterPosition + 1;
      SET WorkingSource = SUBSTRING(WorkingSource FROM StartNewString);
      SET DelimiterPosition = POSITION(Delimiter IN WorkingSource);
      SET CurrentFieldPosition = CurrentFieldPosition + 1;
    END IF;
  END WHILE;
  IF DelimiterPosition > 0 THEN
    -- Remove anything following the delimiter from the string
    SET WorkingSource = SUBSTRING(WorkingSource FROM 1 FOR DelimiterPosition);
    SET WorkingSource = TRIM(TRAILING Delimiter FROM WorkingSource);
  END IF;
  RETURN WorkingSource;
END;

```

Copying repeating fields

You can configure a node with ESQL to copy repeating fields in several ways.

Consider an input XML message that contains a repeating structure:

```

...
<Field_top>
  <field1></field1>
  <field1></field1>
  <field1></field1>
  <field1></field1>
  <field1></field1>
</Field_top>
.....

```

You cannot copy this whole structure field with the following statement:

```
SET OutputRoot.XMLNS.Output_top.Outfield1 = InputRoot.XMLNS.Field_top.field1;
```

That statement copies only the first repeat, and therefore produces the same result as this statement:

```
SET OutputRoot.XMLNS.Output_top.Outfield1[1] = InputRoot.XMLNS.Field_top.field1[1];
```

You can copy the fields within a loop, controlling the iterations with the **CARDINALITY** of the input field:

```

SET I = 1;
SET J = CARDINALITY(InputRoot.XMLNS.Field_top.field1[]);
WHILE I <= J DO
  SET OutputRoot.XMLNS.Output_top.Outfield1[I] = InputRoot.XMLNS.Field_top.field1[I];
  SET I = I + 1;
END WHILE;

```

This might be appropriate if you want to modify each field in the output message as you copy it from the input field (for example, add a number to it, or fold its contents to uppercase), or after it has been copied. If the output message already contained more Field1 fields than existed in the input message, the surplus fields would not be modified by the loop and would remain in the output message.

The following single statement copies the iterations of the input fields to the output fields, and deletes any surplus fields in the output message.

```

SET OutputRoot.XMLNS.Output_top.Outfield1.[] = InputRoot.XMLNS.Field_top.field1[];

```

The example below shows how you can rename the elements when you copy them into the output tree. This statement does not copy across the source element name, therefore each field1 element becomes a Target element.

```

SET OutputRoot.XMLNS.Output_top.Outfield1.Target[] =
  (SELECT I FROM InputRoot.XMLNS.Field_top.field1[] AS I );

```

The next example shows a different way to do the same operation; it produces the same end result.

```

SET OutputRoot.XMLNS.Output_top.Outfield2.Target[]
  = InputRoot.XMLNS.Field_top.field1[];

```

The following example copies across the source element name. Each field1 element is retained as a field1 element under the Target element.

```

SET OutputRoot.XMLNS.Output_top.Outfield3.Target.[]
  = InputRoot.XMLNS.Field_top.field1[];

```

This example is an alternative way to achieve the same result, with field1 elements created under the Target element.

```

SET OutputRoot.XMLNS.Output_top.Outfield4.Target.*[]
  = InputRoot.XMLNS.Field_top.field1[];

```

These examples show that there are several ways in which you can code ESQL to copy repeating fields from source to target. Select the most appropriate method to achieve the results that you require.

The principals shown here apply equally to all areas of the message tree to which you can write data, not just the output message tree.

A note about copying fields:

When you copy an input message element to an output element, the *value* and *type* of the output element is set to that of the input element. Therefore, if, for example, you have an input XML document with an attribute, and you want to set a Field element (rather than an attribute) in your output message to the value of the input attribute, you must include a TYPE clause cast to change the element-type from attribute to Field.

For example, given the following input:

```

<Field01 Attrib01='Attrib01_Value'>Field01_Value</Field01>

```

To create an output, such as in the following example:

```
<MyField_A MyAttrib_A='Attrib01_Value' MyAttrib_B='Field01_Value' >
  <MyField_B>Field01_Value</MyField_BC>
  <MyField_C>Attrib01_Value</MyField_C>
</MyField_A'>
```

You would use the following ESQL:

```
-- Create output attribute from input attribute
SET OutputRoot.XMLNSC.MyField_A.MyAttrib_A = InputRoot.XMLNSC.Field01.Attrib01;
-- Create output field from input field
SET OutputRoot.XMLNSC.MyField_A.MyField_B = InputRoot.XMLNSC.Field01;

-- Create output attribute from input field value, noting we have to
-- "cast" back to an attribute element
SET OutputRoot.XMLNSC.MyField_A.(XMLNSC.Attribute)MyAttrib_B =
    InputRoot.XMLNSC.Field01;

-- Create output field from input attribute value, noting we have to
-- "cast" back to a field element
SET OutputRoot.XMLNSC.MyField_A.(XMLNSC.Field)MyField_C =
    InputRoot.XMLNSC.Field01.Attrib01;
```

Working with elements of type list

The XML Schema specification allows an element, or attribute, to contain a list of values that are based on a simple type, with the individual values separated by white space. In the message tree, an `xsd:list` element is represented as a name node, with an anonymous child node for each list item. Repeating lists can be handled without any loss of information.

Consider the following XML input message:

```
<message1>
  <listE1>one two three</listE1>
</message1>
```

This XML element produces the following message tree:

```
MRM
listE1 (Name)
  "one" (Value)
  "two" (Value)
  "three" (Value)
```

Individual list items can be accessed as `ElementName.*[n]`.

For example, use the following ESQL to access the third item of `listE1`:

```
SET X = FIELDVALUE (InputBody.message1.listE1.*[3]);
```

An attribute can also be of type `xsd:list`.

Consider the following XML input message:

```
<message1>
  <Element listAttr="one two three"/>
</message1>
```

This XML element produces the following message tree:

```
MRM
Element (Name)
  listAttr (Name)
    "one" (Value)
    "two" (Value)
    "three" (Value)
```

As before, individual list items can be accessed as `AttrName.*[n]`.

For example, use the following ESQL to access the third item of `listAttr`:

```
SET X = FIELDVALUE (InputBody.message1.Element.listAttr.*[3]);
```

A list element can occur more than once.

Consider the following XML message:

```
<message1>
<listE1>one two three/listE1>
<listE1>four five six/listE1>
</message1>
```

The message tree for this XML message is:

```
MRM
listE1 (Name)
  "one" (Value)
  "two" (Value)
  "three" (Value)
listE1 (Name)
  "four" (Value)
  "five" (Value)
  "six" (Value)
```

Code the following ESQL to access the first item in the second occurrence of the list:

```
SET X = FIELDVALUE (InputBody.message1.listE1[2].*[1]);
```

Mapping between a list and a repeating element:

Consider the form of the following XML input message:

```
<MRM>
  <inner>abcde fghij 12345</inner>
</MRM>
```

where the element `inner` is of type `xsd:list`, and therefore has three associated string values, rather than a single value.

To copy the three values into an output message, where each value is associated with an instance of repeating elements as shown here:

```
<MRM>
  <str1>abcde</str1>
  <str1>fghij</str1>
  <str1>12345</str1>
</MRM>
```

you might expect that the following ESQL syntax works:

```
DECLARE D INTEGER;
SET D = CARDINALITY(InputBody.str1.*[]);
DECLARE M INTEGER 1;
```

```

WHILE M <= D DO
  SET OutputRoot.MRM.str1[M] = InputBody.inner.*[M];
  SET M = M + 1;
END WHILE;

```

However, the statement:

```

SET OutputRoot.MRM.str1[M] = InputBody.inner.*[M];

```

requests a tree copy from source to target. Because the target element does not yet exist, the statement creates it, and its value and type are set from the source.

Therefore, to create the output message with the required format, given an input element which is of type `xsd:list`, use the “FIELDVALUE function” on page 323 to explicitly retrieve only the value of the source element:

```

SET OutputRoot.MRM.str1[M] = FIELDVALUE(InputBody.inner.*[M]);

```

Manipulating repeating fields in a message tree

This topic describes the use of the SELECT function, and other column functions, to manipulate repeating fields in a message tree.

Suppose that you want to perform a special action on invoices that have a total order value greater than a certain amount. To calculate the total order value of an Invoice field, you must multiply the Price fields by the Quantity fields in all the Items in the message, and total the result. You can do this using a SELECT expression as follows:

```

(
  SELECT SUM( CAST(I.Price AS DECIMAL) * CAST(I.Quantity AS INTEGER) )
  FROM Body.Invoice.Purchases."Item"[] AS I
)

```

The example assumes that you need to use CAST expressions to cast the string values of the fields Price and Quantity into the correct data types. The cast of the Price field into a decimal produces a decimal value with the *natural* scale and precision, that is, whatever scale and precision is necessary to represent the number. These CASTs would not be necessary if the data were already in an appropriate data type.

The SELECT expression works in a similar way to the quantified predicate, and in much the same way that a SELECT works in standard database SQL. The FROM clause specifies what is being iterated, in this case, all Item fields in Invoice, and establishes that the current instance of Item can be referred to using I. This form of SELECT involves a column function, in this case the SUM function, so the SELECT is evaluated by adding together the results of evaluating the expression inside the SUM function for each Item field in the Invoice. As with standard SQL, NULL values are ignored by column functions, with the exception of the COUNT column function explained below, and a NULL value is returned by the column function only if there are no non-NULL values to combine.

The other column functions that are provided are MAX, MIN, and COUNT. The COUNT function has two forms that work in different ways with regard to NULLS. In the first form you use it much like the SUM function above, for example:

```
SELECT COUNT(I.Quantity)
  FROM Body.Invoice.Purchases."Item"[] AS I
```

This expression returns the number of Item fields for which the Quantity field is non-NULL. That is, the COUNT function counts non-NULL values, in the same way that the SUM function adds non-NULL values. The alternative way of using the COUNT function is as follows:

```
SELECT COUNT(*)
  FROM Body.Invoice.Purchases."Item"[] AS I
```

Using COUNT(*) counts the total number of Item fields, regardless of whether any of the fields is NULL. The above example is in fact equivalent to using the CARDINALITY function, as in the following:

```
CARDINALITY(Body.Invoice.Purchases."Item"[])
```

In all the examples of SELECT given here, just as in standard SQL, you could use a WHERE clause to provide filtering on the fields.

Manipulating other parts of the message tree

You can access message tree headers, the properties tree, the local environment tree, the environment tree and the exception list tree.

The following topics describe how you can access parts of the message tree other than the message body data. These parts of the logical tree are independent of the domain in which the message exists, and all these topics apply to messages in the BLOB domain in addition to all other supported domains.

- “Accessing headers”
- “Accessing the Properties tree” on page 52
- “Accessing the local environment tree” on page 53
- “Accessing the environment tree” on page 58
- “Accessing the ExceptionList tree using ESQL” on page 58

Accessing headers

If the input message received by an input node includes message headers that are recognized by the input node, the node invokes the correct parser for each header. You can access these headers using ESQL.

Parsers are supplied for most WebSphere MQ headers. The topics listed below provide some guidance for accessing the information in the MQMD, MQRFH2, and MQPCF headers, which you can follow as general guidance for accessing other headers also present in your messages.

Every header has its own correlation name, for example, MQMD, and you must use this name in all ESQL statements that refer to or set the content of this tree:

- “Accessing the MQMD header” on page 49
- “Accessing the MQRFH2 header” on page 49
- “Accessing the MQCFH header” on page 50

For further details of the contents of these and other WebSphere MQ headers for which WebSphere Message Broker provides a parser, see Element definitions for message parsers.

Accessing transport headers:

You can manipulate WebSphere MQ, HTTP, and JMS transport headers and their properties without writing Compute nodes:

- Use the MQHeader node to add, modify, or delete MQ Message Descriptor (MQMD) and MQ Dead Letter Header (MQDLH) headers.
- Use the HTTPHeader node to add, modify, or delete HTTP headers such as HTTPRequest and HTTPReply.
- Use the JMSHeader node to modify contents of the JMS Header_Values and Application properties so that you can control the node's output without programming.

Accessing the MQMD header:

Code ESQL statements to access the fields of the MQMD header.

WebSphere MQ, WebSphere MQ Everyplace[®], and SCADA messages include an MQMD header.

You can refer to the fields within the MQMD, and you can update them in a Compute node.

For example, you might want to copy the message identifier MSGID in the MQMD to another field in your output message. To do that, code:

```
SET OutputRoot.MRM.Identifier = InputRoot.MQMD.MsgId;
```

If you send a message to an EBCDIC system from a distributed system, you might need to convert the message to a compatible CodedCharSetId and Encoding. To do this conversion, code the following ESQL in the Compute node:

```
SET OutputRoot.MQMD.CodedCharSetId = 500;  
SET OutputRoot.MQMD.Encoding = 785;
```

The MQMD properties of CodedCharSetId and Encoding define the code page and encoding of the section of the message that follows (typically this is either the MQRFH2 header or the message body itself).

Differences exist in the way the Properties folder and the MQMD folder are treated with respect to which folder takes precedence for the same fields. For more information, see Properties versus MQMD folder behavior for various transports.

Accessing the MQRFH2 header:

Code ESQL statements to access the fields of the MQRFH2 header.

When you construct an MQRFH2 header in a Compute node, it includes two types of fields:

- Fields in the MQRFH2 header structure; for example, Format and NameValueCCSID.
- Fields in the MQRFH2 NameValue buffer; for example, mcd and psc.

To differentiate between these two field types, insert a value in front of the referenced field in the MQRFH2 field to identify its type; a value for the

NameValue buffer is not required because this is the default. The value that you specify for the header structure is (MQRFH2.Field).

For example:

- To create or change an MQRFH2 Format field, specify the following ESQL:

```
SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR  ';
```

- To create or change the psc folder with a topic:

```
SET OutputRoot.MQRFH2.psc.Topic = 'department';
```

- To add an MQRFH2 header to an outgoing message that is to be used to make a subscription request:

```
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

SET OutputRoot.MQRFH2.(MQRFH2.Field)Version = 2;
SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR';
SET OutputRoot.MQRFH2.(MQRFH2.Field)NameValueCCSID = 1208;
SET OutputRoot.MQRFH2.psc.Command = 'RegSub';
SET OutputRoot.MQRFH2.psc.Topic = "InputRoot"."MRM"."tope1";
SET OutputRoot.MQRFH2.psc.QMgrName = 'DebugQM';
SET OutputRoot.MQRFH2.psc.QName = 'PUBOUT';
SET OutputRoot.MQRFH2.psc.RegOpt = 'PersAsPub';
```

Variable J is initialized to the value of the cardinality of the existing headers in the message. Using a variable is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

The MQRFH2 header can be parsed using either the MQRFH2 parser or the MQRFH2C compact parser. To consume less memory, use the MQRFH2C compact parser by selecting the Use MQRFH2C compact parser for MQRFH2 Header check box on the input node of the message flow. This results in paths that contain MQRFH2C instead of MQRFH2; for example: SET OutputRoot.MQRFH2C.psc.Topic = 'department';

Target MQRFH2 fields are created only if the headers are copied, and the MQRFH2C parser option is not selected on the MQInput node. In all other circumstances, an MQRFH2C field is created on output.

Accessing the MQCFH header:

Code ESQL statements to access the fields of the MQCFH header (root name MQPCF).

Messages that are of PCF format (MQPCF, MQADMIN, and MQEVENT) include the MQCFH header. You can process the contents of the MQCFH header, accessing parameters, parameter lists, strings, and groups.

The *ParameterCount* field is hidden from view to keep the value synchronized with the true number of parameters. As a result, you cannot directly view, access, or edit the *ParameterCount* field and this applies when you are using:

- ESQL
- The Mapping Node
- Trace, or debugging code.

You can implement your own *ParameterCount* field with a specific value, but this value will be overwritten by the actual number of parameters on the flow exit.

- To access or to construct parameters in the header, use the following syntax:

```
SET OutputRoot.MQPCF.Parameter[nn] =  
    Integer parameter ID
```

If you access a 64-bit parameter, use the following syntax to differentiate from 32-bit parameters:

```
SET OutputRoot.MQPCF.Parameter64[nn] =  
    Integer parameter ID
```

For example:

```
SET OutputRoot.MQPCF.Parameter[1] =  
    MQCACF_AUTH_PROFILE_NAME;
```

- For parameter lists, use the following syntax:

```
SET OutputRoot.MQPCF.ParameterList64[nn] =  
    Integer parameter ID  
SET OutputRoot.MQPCF.ParameterList64[nn].*[xx] =  
    Integer parameter values
```

For example:

```
SET OutputRoot.MQPCF.ParameterList[1] =  
    MQIACF_AUTH_ADD_AUTHS;  
SET OutputRoot.MQPCF.ParameterList[1].*[1] =  
    MQAUTH_SET;  
SET OutputRoot.MQPCF.ParameterList[1].*[2] =  
    MQAUTH_SET_ALL_CONTEXT;
```

- A byte string is a byte array data type, and is supported with this syntax:

```
SET OutputRoot.MQPCF.Parameter[nn] =  
    Integer parameter ID  
SET OutputRoot.MQPCF.Parameter[nn].* =  
    Integer, String or ByteArray Parameter value
```

- A group is implemented as a folder containing more PCF, and requires the following syntax:

```
SET OutputRoot.MQPCF.Group[xx] =  
    Group Parameter ID
```

For example:

```
SET OutputRoot.MQPCF.Group[1] =  
    MQGACF_Q_ACCOUNTING_DATA;  
SET OutputRoot.MQPCF.Group[1].Parameter[1] =  
    MQCA_CREATION_DATE;  
SET OutputRoot.MQPCF.Group[1].Parameter[1].* =  
    '2007-02-05';
```

You can also use nested groups; for example;

```

SET OutputRoot.MQPCF.Group[1].Group[1] =
  MQGACF_Q_ACCOUNTING_DATA;
SET OutputRoot.MQPCF.Group[1].Group[1].Parameter[1] =
  MQCA_CREATION_DATE;
SET OutputRoot.MQPCF.Group[1].Group[1].Parameter[1].* =
  '2007-02-05';

```

- A filter is almost identical to a parameter, but it also includes an operator. Therefore the syntax is a tree with named values:

```

SET OutputRoot.MQPCF.Filter[xx] =
  Integer parameter ID
SET OutputRoot.MQPCF.Filter[xx].Operator =
  Integer Filter name
SET OutputRoot.MQPCF.Filter[xx].Value =
  Byte, Integer or String Filter Value

```

- The following is sample code that can be used as an example to create an MQPCF message in a Compute node:

```

CREATE NEXTSIBLING OF OutputRoot.Properties DOMAIN 'MQMD';
CREATE NEXTSIBLING OF OutputRoot.MQMD DOMAIN 'MQADMIN'
NAME 'MQPCF';
CREATE FIELD OutputRoot.MQPCF;
SET OutputRoot.MQMD.MsgType = MQMT_REQUEST;
SET OutputRoot.MQMD.ReplyToQ = 'REPLYQ';
DECLARE refRequest REFERENCE TO OutputRoot.MQPCF;
SET refRequest.Type = 16; --MQCFT_COMMAND_XR z/OS
SET refRequest.StrucLength = MQCFH_STRUC_LENGTH;
SET refRequest.Version = 3; -- required for z/OS
SET refRequest.Command = MQCMD_INQUIRE_Q;
SET refRequest.MsgSeqNumber = 1;
SET refRequest.Control = MQCFC_LAST;
/* First parameter: Queue Name. */
SET refRequest.Parameter[1] = MQCA_Q_NAME;
SET refRequest.Parameter[1].* = 'QUEUENAME.*';
SET refRequest.ParameterList[1] = MQIACF_Q_ATTRS;
SET refRequest.ParameterList[1].* = MQIACF_ALL;

```

Accessing the Properties tree

The Properties tree has its own correlation name, Properties, and you must use this in all ESQL statements that refer to or set the content of this tree.

The fields in the Properties tree contain values that define the characteristics of the message. For example, the Properties tree contains a field for the message domain, and fields for the encoding and CCSID in which message data is encoded. For a full list of fields in this tree, see Data types for elements in the Properties subtree.

You can interrogate and update these fields using the appropriate ESQL statements. If you create a new output message in the Compute node, you must set values for the message properties.

Setting output message properties:

If you use the Compute node to generate a new output message, you must set its properties in the Properties tree. The output message properties do not have to be the same as the input message properties.

For example, to set the output message properties for an output MRM message, set the following properties:

Property	Value
Message domain	MRM

Property	Value
Message set	Message set identifier
Message type	Message name ¹
Message format	Physical format name ²

Notes:

1. For details of the syntax of Message type, see Specifying namespaces in the Message Type property.
2. The name that you specify for the physical layer must match the name that you have defined for it. The default physical layer names are Binary1, XML1, and Text1.

This ESQL procedure sets message properties to values passed in by the calling statement. You might find that you have to perform this task frequently, and you can use a procedure such as this in many different nodes and message flows. If you prefer, you can code ESQL that sets specific values.

```
CREATE PROCEDURE setMessageProperties(IN OutputRoot REFERENCE, IN setName char,
    IN typeName char, IN formatName char) BEGIN
    /*****
    * A procedure that sets the message properties
    *****/
    set OutputRoot.Properties.MessageSet = setName;
    set OutputRoot.Properties.MessageType = typeName;
    set OutputRoot.Properties.MessageFormat = formatName;
END;
```

To set the output message domain, you can set the message property, or you can code ESQL statements that refer to the required domain in the second qualifier of the SET statement, the parser field. For example, the ESQL statement sets the domain to MRM:

```
SET OutputRoot.MRM.Field1 = 'field1 data';
```

This ESQL statement sets the domain to XMLNS:

```
SET OutputRoot.XMLNS.Field1 = 'field1 data';
```

Do not specify more than one domain in the ESQL for any single message. However, if you use PROPAGATE statements to generate several output messages, you can set a different domain for each message.

For information about the full list of elements in the Properties tree, see Data types for elements in the Properties subtree.

Differences exist in the way the Properties folder and the MQMD folder are treated with respect to which folder takes precedence for the same fields. For more information, see Properties versus MQMD folder behavior for various transports.

Accessing the local environment tree

The local environment tree has its own correlation name, LocalEnvironment, and you must use this name in all ESQL statements that refer to or set the content of this tree.

The local environment tree is used by the broker, and you can refer to and modify this information. You can also extend the tree to contain information that you create yourself. You can create subtrees of this tree that you can use as a scratchpad or working area.

The message flow sets up information in two subtrees, `Destination` and `WrittenDestination`, below the `LocalEnvironment` root. You can refer to the content of both of these subtrees, and you can write to the `Destination` tree to influence the way in which the message flow processes your message. However, if you write to the `Destination` tree, follow the defined structure to ensure that the tree remains valid.

The `WrittenDestination` subtree contains the addresses to which the message has been written. Its name is fixed and it is created by the message flow when a message is propagated through the `Out` terminal of a request, output, or reply node. The subtree includes transport-specific information (for example, if the output message has been put to a WebSphere MQ queue, it includes the queue manager and queue names). You can use one of the following methods to obtain information about the details of a message after it has been sent by the nodes:

- Connect a `Compute` node to the `Out` terminal.
- Configure a user exit to process an output message callback event, as described in *Exploiting user exits*.

The topic for each node that supports `WrittenDestination` information contains details about the data that it contains.

If you want the local environment tree to be included in the output message that is propagated by the `Compute` node, you must set the `Compute` node property `Compute mode` to a value that includes the local environment (for example, `All`). If you do not, the local environment tree is not copied to the output message.

The information that you insert into `DestinationData` or `Defaults` depends on the characteristic of the corresponding node property:

- If a node property is represented by a check box (for example, `New Message ID`), set the `Defaults` or `DestinationData` element to `Yes` (equivalent to selecting the check box) or `No` (equivalent to clearing the check box).
- If a node property is represented by a drop-down list (for example, `Transaction Mode`), set the `Defaults` or `DestinationData` element to the appropriate character string (for example `Automatic`).
- If a node property is represented by a text entry field (for example, `Queue Manager Name`), set the `Defaults` or `DestinationData` element to the character string that you would enter in this field.

If necessary, configure the sending node to indicate where the destination information is. For example, for the output node `MQOutput`, set `Destination Mode`:

- If you set `Destination Mode` to `Queue Name`, the output message is sent to the queue identified in the output node properties `Queue Name` and `Queue Manager Name`. `Destination` is not referenced by the node.
- If you set `Destination Mode` to `Destination List`, the node extracts the destination information from the `Destination` subtree. If you use this value you can send a single message to multiple destinations, if you configure `Destination` and a single output node correctly. The node checks the node properties only if a value is not available in `Destination` (as described above).

- If you set Destination Mode to Reply To Queue, the message is sent to the reply-to queue identified in the MQMD in this message (field ReplyToQ). Destination is not referenced by the node.

To find more information about ESQL procedures that perform typical updates to the local environment, see “Populating Destination in the local environment tree” on page 56. Review the ESQL statements in these procedures to see how to modify the local environment. You can use these procedures unchanged, or modify them for your own requirements.

To find more information about how to extend the contents of this tree for your own purposes, see “Using scratchpad areas in the local environment.”

For another example of how you can use the local environment to modify the behavior of a message flow, refer to the XML_PassengerQuery message flow in the following sample program:

- Airline Reservations

The Compute node in this message flow writes a list of destinations in the RouterList subtree of Destination that are used as labels by a later RouteToLabel node that propagates the message to the corresponding Label node. You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Using scratchpad areas in the local environment:

The local environment tree includes a subtree called variables. This subtree is always created, but is never populated by the message flow. Use this area for your own purposes; for example, to pass information from one node to another. You can create other subtrees of the local environment tree.

The advantage of creating your own data in a scratchpad in the local environment is that this data can be propagated as part of the logical tree to subsequent nodes in the message flow. If you create a new output message in a Compute node, you can also include all or part of the local environment tree from the input message in the new output message.

To ensure that the information in the local environment is propagated further down the flow, the Compute mode property of the Compute node must be set to include the local environment as part of the output tree (for example, specify LocalEnvironment and Message). For further details about the Compute mode property, see Setting the mode.

However, any data updates or additions that you make in one node are not retained if the message moves backwards through the message flow (for example, if an exception is thrown). If you create your own data, and want that data to be preserved throughout the message flow, you must use the environment tree.

You can set values in the variables subtree in a Compute node and those values can be used later by another node (Compute, Database, or Filter) for some purpose that you determine when you configure the message flow.

The local environment is not in scope in a Compute node, therefore you must use InputLocalEnvironment and OutputLocalEnvironment instead. For example, you might use the scratchpad in the local environment to propagate the destination of an output message to subsequent nodes in a message flow. Your first Compute

node determines that the output messages from this message flow must go to WebSphere MQ queues. Include the following ESQL to insert this information into the local environment by setting the value of *OutputLocation* in the *OutputLocalEnvironment*:

```
SET OutputLocalEnvironment.Variables.OutputLocation = 'MQ';
```

Your second Compute node can access this information from its input message. In the ESQL in this node, use the correlation name *InputLocalEnvironment* to identify the local environment tree in the input message that contains this data. The following ESQL sets *queueManagerName* and *queueName* based on the content of *OutputLocation* in the local environment, by using *InputLocalEnvironment*:

```
IF InputLocalEnvironment.Variables.OutputLocation = 'MQ' THEN
  SET OutputLocalEnvironment.Destination.MQ.DestinationData.queueManagerName = 'myQMName';
  SET OutputLocalEnvironment.Destination.MQ.DestinationData.queueName = 'myQueueName';
END IF;
```

In the example, *queueManagerName* and *queueName* are set for the *Destination* subtree in the output message. You must set the Compute mode of the second Compute node to include the local environment tree in the output message. Configure the *MQOutput* node to use the destination list that you have created in the local environment tree by setting the *Destination Mode* property to *Destination List*.

For information about the full list of elements in the *DestinationData* subtree, see *Data types for elements in the MQ DestinationData subtree*.

Populating Destination in the local environment tree:

Use the *Destination* subtree to set up the target destinations that are used by output nodes, the *HTTPRequest* node, the *SOAPRequest* node, the *SOAPAsyncRequest* node, and the *RouteToLabel* node. The following examples show how you can create and use an ESQL procedure to perform the task of setting up values for each of these uses.

Copy and use these procedures as shown, or you can modify or extend them to perform similar tasks.

If you are creating this ESQL code for a Compute node, you must configure the node by setting the *Compute Mode* property so that it has access to the local environment tree in the output message. You must select one of the three values *LocalEnvironment*, *LocalEnvironment And Message*, or *All*.

Adding a queue name for the MQOutput node with the Destination Mode property set to Destination List

```
CREATE PROCEDURE addToMQDestinationList(IN LocalEnvironment REFERENCE, IN newQueue char) BEGIN
  /*****
  * A procedure that adds a queue name to the MQ destination list in the local environment.
  * This list is used by an MQOutput node that has its mode set to Destination list.
  *
  * IN LocalEnvironment: the LocalEnvironment to be modified.
  * IN queue: the queue to be added to the list
  *
  *****/
  DECLARE I INTEGER CARDINALITY(LocalEnvironment.Destination.MQ.DestinationData[]);
  IF I = 0 THEN
    SET OutputLocalEnvironment.Destination.MQ.DestinationData[1].queueName = newQueue;
```



```

ELSE
  SET OutputLocalEnvironment.Destination.MQ.DestinationData[I+1].queueName = newQueue;
END IF;
END;

```

For full details of these elements, see Data types for elements in the MQ DestinationData subtree.

Changing the default URL for a SOAPRequest node or a SOAPAsyncRequest node request

```

CREATE PROCEDURE overrideDefaultSOAPRequestURL(IN LocalEnvironment REFERENCE, IN newUrl char) BEGIN
  /*****
  * A procedure that changes the URL to which the SOAPRequest node or
  * SOAPAsyncRequest node sends the request.
  *
  * IN LocalEnvironment: the LocalEnvironment to be modified.
  * IN queue: the URL to which to send the request.
  *
  *****/
  SET OutputLocalEnvironment.Destination.SOAP.Request.Transport.HTTP.WebServiceURL = newUrl;
END;

```

Changing the default URL for an HTTPRequest node request

```

CREATE PROCEDURE overrideDefaultHTTPRequestURL(IN LocalEnvironment REFERENCE, IN newUrl char) BEGIN
  /*****
  * A procedure that changes the URL to which the HTTPRequest node sends the request.
  *
  * IN LocalEnvironment: the LocalEnvironment to be modified.
  * IN queue: the URL to which to send the request.
  *
  *****/
  SET OutputLocalEnvironment.Destination.HTTP.RequestURL = newUrl;
END;

```

Adding a label for the RouteToLabel node

```

CREATE PROCEDURE addToRouteToLabelList(IN LocalEnvironment REFERENCE, IN newLabel char) BEGIN
  /*****
  * A procedure that adds a label name to the RouteToLabel list in the local environment.
  * This list is used by a RoteToLabel node.
  *
  * IN LocalEnvironment: the LocalEnvironment to be modified.
  * IN label: the label to be added to the list
  *
  *****/
  IF LocalEnvironment.Destination.RouterList.DestinationData is null THEN
    SET OutputLocalEnvironment.Destination.RouterList.DestinationData."label" = newLabel;
  ELSE
    CREATE LASTCHILD OF LocalEnvironment.Destination.RouterList.DestinationData
      NAME 'label' VALUE newLabel;
  END IF;
END;

```

Setting up JMS destination lists

You can configure a JMSOutput node to send to multiple JMS Queues, or to publish to multiple JMS Topics by using a destination list that is created in the local environment tree by a transformation node. The following example shows how to set up JMS destination lists in the local environment tree:

```

CREATE PROCEDURE CreateJMSDestinationList() BEGIN
  SET OutputLocalEnvironment.Destination.JMSDestinationList.DestinationData[1] = 'jndi://TestDestQueue1';
  SET OutputLocalEnvironment.Destination.JMSDestinationList.DestinationData[2] = 'jndi://TestDestQueue2';
  SET OutputLocalEnvironment.Destination.JMSDestinationList.DestinationData[3] = 'jndi://TestDestQueue3';
END;

```

Accessing the environment tree

The environment tree has its own correlation name, `Environment`, and you must use this name in all ESQL statements that refer to, or set, the content of this tree.

The environment tree is always created when the logical tree is created for an input message. However, the message flow neither populates it, nor uses its contents. You can use this tree for your own purposes, for example, to pass information from one node to another. You can use the whole tree as a scratchpad or working area.

The advantage of creating your own data in environment is that this data is propagated as part of the logical tree to subsequent nodes in the message flow. If you create a new output message in a Compute node, the environment tree is also copied from the input message to the new output message. (In contrast to the local environment tree, which is only included in the output message if you explicitly request that it is).

Only one environment tree is present for the duration of the message flow. Any data updates, or additions, that you make in one node are retained, and all of the nodes in the message flow have access to the latest copy of this tree. Even if the message flows back through the message flow (for example, if an exception is thrown, or if the message is processed through the second terminal of the `FlowOrder` node), the latest state is retained. (In contrast to the local environment tree, which reverts to its previous state if the message flows back through the message flow.)

You can use this tree for any purpose you choose. For example, you can use the following ESQL statements to create fields in the tree:

```
SET Environment.Variables =  
    ROW('granary' AS bread, 'reisling' AS wine, 'stilton' AS cheese);  
SET Environment.Variables.Colors[] =  
    LIST{'yellow', 'green', 'blue', 'red', 'black'};  
SET Environment.Variables.Country[] = LIST{ROW('UK' AS name, 'pound' AS currency),  
    ROW('USA' AS name, 'dollar' AS currency)};
```

This information is now available to all nodes to which a message is propagated, regardless of their relative position in the message flow.

For another example of how you can use environment to store information used by other nodes in the message flow, look at the Reservation message flow in the following sample:

- Airline Reservations

The Compute node in this message flow writes information to the subtree `Environment.Variables` that it has extracted from a database according to the value of a field in the input message.

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Accessing the ExceptionList tree using ESQL

The `ExceptionList` tree has its own correlation name, `ExceptionList`, and you must use this in all ESQL statements that refer to or set the content of this tree.

This tree is created with the logical tree when an input message is parsed. It is initially empty, and is only populated if an exception occurs during message flow

processing. It is possible that more than one exception can occur; if more than one exception occurs, the ExceptionList tree contains a subtree for each exception.

You can access the ExceptionList tree in Compute, Database, and Filter nodes, and you can update it in a Compute node. You must use the appropriate correlation name; ExceptionList for a Database or Filter node, and InputExceptionList for a Compute node.

You might want to access this tree in a node in an error handling procedure. For example, you might want to route the message to a different path based on the type of exception, for example one that you have explicitly generated using an ESQL THROW statement, or one that the broker has generated.

The following ESQL shows how you can access the ExceptionList and process each child that it contains:

```
-- Declare a reference for the ExceptionList
-- (in a Compute node use InputExceptionList)
DECLARE start REFERENCE TO ExceptionList.*[1];

-- Loop through the exception list children
WHILE start.Number IS NOT NULL DO
  -- more ESQL

  -- Move start to the last child of the field to which it currently points
  MOVE start LASTCHILD;
END WHILE;
```

The following example shows an extract of ESQL that has been coded for a Compute node to loop through the exception list to the last (nested) exception description and extract the error number. This error relates to the original cause of the problem and normally provides the most precise information. Subsequent action taken by the message flow can be decided by the error number retrieved in this way.

```
CREATE PROCEDURE getLastExceptionDetail(IN InputTree reference,OUT messageNumber integer,
OUT messageText char)
  /*****
  * A procedure that will get the details of the last exception from a message
  * IN InputTree: The incoming exception list
  * IN messageNumber: The last message numberr.
  * IN messageText: The last message text.
  *****/
BEGIN
  -- Create a reference to the first child of the exception list
  declare ptrException reference to InputTree.*[1];
  -- keep looping while the moves to the child of exception list work
  WHILE lastmove(ptrException) DO
  -- store the current values for the error number and text
  IF ptrException.Number is not null THEN
    SET messageNumber = ptrException.Number;
    SET messageText = ptrException.Text;
  END IF;
  -- now move to the last child which should be the next exceptionlist
  move ptrException lastchild;
  END WHILE;
END;
```

For more information about the use of ExceptionList, look at the subflow in the following sample which includes ESQL that interrogates the ExceptionList structure and takes specific action according to its content:

- Error Handler

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

For information on accessing the ExceptionList tree using Java, see [Accessing the ExceptionList tree using Java](#)

Transforming from one data type to another

You can use ESQL to transform messages and data types in many ways.

The following topics provide guidance about the following:

- “Casting data from message fields”
- “Converting code page and message encoding”
- “Converting EBCDIC NL to ASCII CR LF” on page 63
- “Changing message format” on page 65

Casting data from message fields

You can use the CAST function to transform the data type of one value to match the data type of the other. For example, you can use the CAST function when you process generic XML messages. All fields in an XML message have character values, so if you want to perform arithmetic calculations or datetime comparisons, for example, you must convert the string value of the field into a value of the appropriate type using CAST.

When you compare an element with another element, variable or constant, ensure that the value with which you are comparing the element is consistent (for example, character with character). If the values are not consistent, the broker generates a runtime error if it cannot provide an implicit casting to resolve the inconsistency. For details of what implicit casts are supported, see “Implicit casts” on page 365.

In the Invoice message, the field InvoiceDate contains the date of the invoice. If you want to refer to or manipulate this field, you must CAST it to the correct format first. For example, to refer to this field in a test:

```
IF CAST(Body.Invoice.InvoiceDate AS DATE) = CURRENT_DATE THEN
```

This converts the string value of the InvoiceDate field into a date value, and compares it to the current date.

Another example is casting from integer to character:

```
DECLARE I INTEGER 1;
DECLARE C CHARACTER;
```

```
-- The following statement generates an error
SET C = I;
```

```
-- The following statement is valid
SET C = CAST(I AS CHARACTER);
```

Converting code page and message encoding

You can use ESQL within a Compute node to convert data for code page and message encoding.

If your message flow is processing WebSphere MQ messages, you can use WebSphere MQ facilities (including get and put options and WebSphere MQ data conversion exits) to provide these conversions. If you are not processing WebSphere MQ messages, or you choose not to use WebSphere MQ facilities, you can use WebSphere Message Broker facilities by coding the appropriate ESQL in a Compute node in your message flow.

The contents of the MQMD, the MQRFH2, and the message body of a message in the MRM domain that has been modeled with a CWF physical format can be subject to code page and encoding conversion. The contents of a message body of a message in the XML, XMLNS, and JMS domains, and those messages in the MRM domain that have been modeled with an XML or TDS physical format, are treated as strings. Only code page conversion applies; no encoding conversion is required.

For messages in the MRM domain modeled with a CWF physical format, you can set the MQMD CCSID and Encoding fields of the output message, plus the CCSID and Encoding of any additional headers, to the required target value.

For messages in the MRM domain modeled with an XML or TDS physical format, you can set the MQMD CCSID field of the output message, plus the CCSID of any additional headers. XML and TDS data is handled as strings and is therefore subject to CCSID conversion only.

An example WebSphere MQ message has an MQMD header, an MQRFH2 header, and a message body. To convert this message to a mainframe CodedCharSetId and Encoding, code the following ESQL in the Compute node:

```
SET OutputRoot.MQMD.CodedCharSetId = 500;  
SET OutputRoot.MQMD.Encoding = 785;  
SET OutputRoot.MQRFH2.CodedCharSetId = 500;  
SET OutputRoot.MQRFH2.Encoding = 785;
```

The following example illustrates what you must do to modify a CWF message so that it can be passed from WebSphere Message Broker to IMS™ on z/OS®.

1. You have defined the input message in XML and are using an MQRFH2 header. Remove the header before passing the message to IMS.
2. The message passed to IMS must have MQIIH header, and must be in the z/OS code page. This message is modeled in the MRM and has the name IMS1. Define the PIC X fields in this message as logical type string for conversions between EBCDIC and ASCII to take place. If the fields are binary logical type, no data conversion occurs; binary data is ignored when a CWF message is parsed by the MRM parser.
3. The message received from IMS is also defined in the MRM and has the name IMS2. Define the PIC X fields in this message as logical type string for conversions between EBCDIC and ASCII to take place. If the fields are binary logical type, no data conversion occurs; binary data is ignored when a CWF message is parsed by the MRM parser.
4. Convert the reply message to the Windows® code page. The MQIIH header is retained on this message.
5. You have created a message flow that contains the following nodes: :
 - a. The outbound flow, **MQInput1 --> Compute1 --> MQOutput1**.
 - b. The inbound flow, **MQInput2 --> Compute2 --> MQOutput2**.
6. Code ESQL in Compute1 (outbound) node as follows, specifying the relevant MessageSet ID. This code shows the use of the default CWF physical layer

name. You must use the name that matches your model definitions. If you specify an incorrect value, the broker fails with message BIP5431.

```
-- Loop to copy message headers
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J - 1 DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

SET OutputRoot.MQMD.CodedCharSetId = 500;
SET OutputRoot.MQMD.Encoding = 785;
SET OutputRoot.MQMD.Format = 'MQIMS  ';
SET OutputRoot.MQIIH.Version = 1;
SET OutputRoot.MQIIH.StrucLength = 84;
SET OutputRoot.MQIIH.Encoding = 785;
SET OutputRoot.MQIIH.CodedCharSetId = 500;
SET OutputRoot.MQIIH.Format = 'MQIMSVS ';
SET OutputRoot.MQIIH.Flags = 0;
SET OutputRoot.MQIIH.LTermOverride = '      ';
SET OutputRoot.MQIIH.MFSMapName = '      ';
SET OutputRoot.MQIIH.ReplyToFormat = 'MQIMSVS ';
SET OutputRoot.MQIIH.Authenticator = '      ';
SET OutputRoot.MQIIH.TranInstanceId = X'00000000000000000000000000000000';
SET OutputRoot.MQIIH.TranState = '  ';
SET OutputRoot.MQIIH.CommitMode = '0';
SET OutputRoot.MQIIH.SecurityScope = 'C';
SET OutputRoot.MQIIH.Reserved = '  ';
SET OutputRoot.MRM.e_e1en08 = 30;
SET OutputRoot.MRM.e_e1en09 = 0;
SET OutputRoot.MRM.e_string08 = InputBody.e_string01;
SET OutputRoot.MRM.e_binary02 = X'31323334353637383940';
SET OutputRoot.Properties.MessageSet = 'DHCJOEG072001';
SET OutputRoot.Properties.MessageType = 'IMS1';
SET OutputRoot.Properties.MessageFormat = 'Binary1';
```

The use of a variable, J, that is initialized to the value of the cardinality of the existing headers in the message, is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

7. Create ESQL in Compute2 (inbound) node as follows, specifying the relevant MessageSet ID. This code shows the use of the default CWF physical layer name. You must use the name that matches your model definition. If you specify an incorrect value, the broker fails with message BIP5431.

```

-- Loop to copy message headers
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

SET OutputRoot.MQMD.CodedCharSetId = 437;
SET OutputRoot.MQMD.Encoding = 546;
SET OutputRoot.MQMD.Format = 'MQIMS  ';
SET OutputRoot.MQIIH.CodedCharSetId = 437;
SET OutputRoot.MQIIH.Encoding = 546;
SET OutputRoot.MQIIH.Format = '      ';
SET OutputRoot.MRM = InputBody;
SET OutputRoot.Properties.MessageSet = 'DHCJ0EG072001';
SET OutputRoot.Properties.MessageType = 'IMS2';
SET OutputRoot.Properties.MessageFormat = 'Binary1';

```

You do not have to set any specific values for the MQInput1 node properties, because the message and message set are identified in the MQRFH2 header, and no conversion is required.

You must set values for message domain, set, type, and format in the MQInput node for the inbound message flow (MQInput2). You do not need to set conversion parameters.

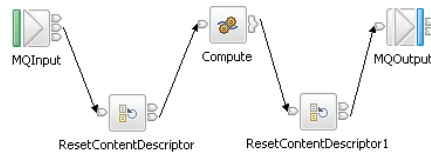
One specific situation in which you might need to convert data in one code page to another is when messages contain newline characters and are passed between EBCDIC and ASCII systems. The required conversion for this situation is described in “Converting EBCDIC NL to ASCII CR LF.”

Converting EBCDIC NL to ASCII CR LF:

You might want to change newline (NL) characters in a text message to carriage return (CR) and line feed (LF) character pairs. This example shows one way in which you can convert these characters.

This conversion might be useful if messages from an EBCDIC platform (for example, using CCSID 1047) are sent to an ASCII platform (for example, using CCSID 437). Problems can arise because the EBCDIC NL character hex '15' is converted to the undefined ASCII character hex '7F'. The ASCII code page has no corresponding code point for the NL character.

In this example, a message flow is created that interprets the input message as a message in the BLOB domain. This message is passed into a ResetContentDescriptor node to reset the data to a message in the MRM domain. The message is called msg_nl (a set of repeating string elements delimited by EBCDIC NL characters). A Compute node is then used to create an output based on another message in the MRM domain called msg_crlf (a set of repeating string elements delimited by CR LF pairs). The message domain is then changed back to BLOB in another ResetContentDescriptor node. This message flow is shown in the following diagram.



The following instructions show how to create the messages and configure the message flow.

1. Create the message models for the messages in the MRM domain:
 - a. Create a message set project called myProj.
 - b. Create a message set called myMessageSet with a TDS physical format (the default name is Text1).
 - c. Create an element string1 of type xsd:string.
 - d. Create a complex type called t_msg_nl and specify the following complex type properties:
 - Composition = Ordered Set
 - Content Validation = Closed
 - Data Element Separation = All Elements Delimited
 - Delimiter = <U+0085> (hex '0085' is the UTF-16 representation of an NL character)
 - Repeat = Yes
 - Min Occurs = 1
 - Max Occurs = 50 (the text of the message is assumed to consist of no more than 50 lines)
 - e. Add Element string1, and set the following property:
 - Repeating Element Delimiter = <U+0085>
 - f. Create a Message msg_nl, and set its associated complex type to t_msg_nl
 - g. Create a complex type called t_msg_crlf, and specify the following complex type properties:
 - Composition = Ordered Set
 - Content Validation = Closed
 - Data Element Separation = All Elements Delimited
 - Delimiter = <CR><LF> (<CR> and <LF> are the mnemonics for the CR and LF characters)
 - Repeat = Yes
 - Min Occurs = 1
 - Max Occurs = 50
 - h. Add Element string1, and set the following property:
 - Repeating Element Delimiter = <CR><LF>
 - i. Create a Message msg_crlf, and set complex type to t_msg_crlf.
2. Configure the message flow shown in the previous figure:
 - a. Start with the MQInput node:
 - Set Message Domain = BLOB
 - Set Queue Name = <Your input message queue name>
 - b. Add the ResetContentDescriptor node, connected to the Out terminal of the MQInput node:
 - Set Message Domain = MRM
 - Select Reset Message Domain

- Set Message Set = <Your Message Set ID> (this field has a maximum of 13 characters)
 - Select Reset Message Set
 - Set Message Type = msg_nl
 - Select Reset Message Type
 - Set Message Format = Text1
 - Select Reset Message Format
- c. Add the Compute node, connected to the Out terminal of the ResetContentDescriptor node:
- Enter a name for the ESQL Module for this node, or accept the default (<message flow name>_Compute).
 - Right-click the Compute node, and select **Open ESQL**. Add the following ESQL code in the module:

```
-- Declare local working variables
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);

-- Loop to copy all message headers from input to output message
WHILE I < J DO
  SET OutputRoot.*[I] = InputRoot.*[I];
  SET I=I+1;
END WHILE;

-- Set new output message type which uses CRLF delimiter
SET OutputRoot.Properties.MessageType = 't_msg_crlf';

-- Loop to copy each instance of string1 child within message body
SET I = 1;
SET J = CARDINALITY("InputBody"."string1"[]);
WHILE I <= J DO
  SET "OutputRoot"."MRM"."string1"[I] = "InputBody"."string1"[I];
  SET I=I+1;
END WHILE;
```

The use of a variable, J, initialized to the value of the cardinality of the existing headers in the message, is more efficient than calculating the cardinality on each iteration of the loop, which happens if you code the following WHILE statement:

```
WHILE I < CARDINALITY(InputRoot.*[]) DO
```

- d. Add the ResetContentDescriptor1 node, connected to the Out terminal of the Compute node:
 - Set Message Domain = BLOB
 - Select Reset Message Domain.
- e. Finally, add the MQOutput node, connected to the Out terminal of the ResetContentDescriptor1 node. Configure its properties to direct the output message to the required queue or queues.

Changing message format

Use the Compute node to copy part of an input message to an output message. The results of such a copy depend on the type of input and output parsers involved.

Like parsers:

Where both the source and target messages have the same folder structure at root level, a *like-parser-copy* is performed. For example:

```
SET OutputRoot.MQMD = InputRoot.MQMD;
```

This statement copies all the children in the MQMD folder of the input message to the MQMD folder of the output message.

Another example of a tree structure that supports a like-parser-copy is:

```
SET OutputRoot.XMLNS.Data.Account = InputRoot.XMLNS.Customer.Bank.Data;
```

To transform an input message in the MRM domain to an output message also in the MRM domain, you can use either the Compute or the Mapping node. The Mapping node can interpret the action that is required because it knows the format of both messages. Content Assist in the ESQL module for the Compute node can also use the message definitions for those messages. If the messages are not in the same namespace, you must use the Compute node.

To use Content Assist with message references, you must set up a project reference from the project containing the ESQL to the project containing the message set. For information about setting up a project reference, see Project references.

If both input and output messages are not in the MRM domain, you must use the Compute node and specify the structure of the messages yourself.

Unlike parsers:

Where the source and target messages have different folder structures at root level, you cannot make an exact copy of the message source. Instead, the *unlike-parser-copy* views the source message as a set of nested folders terminated by a leaf name-value pair. For example, copying the following message from XML to MRM:

```
<Name3><Name31>Value31</Name31>Value32</Name3>
```

produces a name element Name3, and a name-value element called Name31 with the value Value31. The second XML pcdta (Value32) cannot be represented and is discarded.

The unlike-parser-copy scans the source tree, and copies folders, also known as name elements, and leaf name-value pairs. Everything else, including elements flagged as *special* by the source parser, is not copied.

An example of a tree structure that results in an unlike-parser-copy is:

```
SET OutputRoot.MRM.Data.Account = InputRoot.XMLNS.Data.Account;
```

If the algorithm used to make an unlike-parser-copy does not suit your tree structure, you should further qualify the source field to restrict the amount of the tree that is copied.

Be careful when you copy information from input messages to output messages in different domains. You could code ESQL that creates a message structure or content that is not completely consistent with the rules of the parser that processes the output message. This action can result in an output message not being created, or being created with unexpected content. If you believe that the output message

generated by a particular message flow does not contain the correct content, or have the expected form, check the ESQL that creates the output message, and look for potential mismatches of structure, field types, field names, and field values.

When copying trees between unlike parsers, you should set the message format of the target parser. For example, if a message set has been defined with XMLNS and CWF formats, the following commands are required to copy an input XMLNS stream to the MRM parser and set the latter to be generated in CWF format:

```
-- Copy message to the output, moving from XMLNS to MRM domains
SET OutputRoot.MRM = InputRoot.XMLNS.rootElement;

-- Set the CWF format for output by the MRM domain
SET OutputRoot.Properties.MessageType = '<MessageTypeName>';
SET OutputRoot.Properties.MessageSet = '<MessageSetName>';
SET OutputRoot.Properties.MessageFormat = 'CWF';
```

Adding keywords to ESQL files

You can add keywords to ESQL files to contain information that you want to associate with a message flow.

Use one or more of the following methods:

Comment fields

Add the keyword as a comment in the ESQL file:

```
-- $MQSI compiled by = John MQSI$
```

Static strings

Include the keyword as part of a static string in the ESQL file:

```
SET target = '$MQSI_target = production only MQSI$'
```

Variable string

Include the keyword value as a variable string in the ESQL file:

```
$MQSI_VERSION=$id$MQSI$
```

In this example, when the message flow source is extracted from the file repository, the repository's plug-in has been configured to substitute the identifier *\$id\$* with the actual version number. The identifier value that is required depends on the capability and configuration of the repository, and is not part of WebSphere Message Broker.

Restrictions within keywords

Do not use the following characters within keywords, because they cause unpredictable behavior:

```
^ $ . | \ < > ? + * = & [ ] ( )
```

You can use these characters in the values that are associated with keywords; for example:

- `$MQSI RCSVER=$id$ MQSI$` is acceptable
- `$MQSI $name=Fred MQSI$` is not acceptable

Interaction with databases using ESQL

Use ESQL statements and functions to read from, write to, and modify databases from your message flows.

ESQL has a number of statements and functions for accessing databases:

- The “CALL statement” on page 196 invokes a stored procedure.
- The “DELETE FROM statement” on page 243 removes rows from a database table.
- The “INSERT statement” on page 249 adds a row to a database table.
- The “PASSTHRU function” on page 374 can be used to make complex selections.
- The “PASSTHRU statement” on page 259 can be used to invoke administrative operations (for example, creating a table).
- The “SELECT function” on page 345 retrieves data from a table.
- The “UPDATE statement” on page 272 changes one or more values stored in zero or more rows.

You can access user databases from Compute, Database, and Filter nodes. The database access capabilities of all these nodes is identical.

You can use the data in the databases to update or create messages, or use the data in the message to update or create data in the databases.

Select **Throw exception on database error** and **Treat warnings as errors**, and set **Transaction** to **Automatic** on each node that access a database, to provide maximum flexibility.

For information about configuring the broker and the database to support access from message flows, see *Accessing databases from ESQL*.

- “Referencing columns in a database”
- “Selecting data from database columns” on page 70
- “Accessing multiple database tables” on page 74
- “Changing database content” on page 75
- “Checking returns to SELECT” on page 76
- “Committing database updates” on page 77
- “Invoking stored procedures” on page 77

Referencing columns in a database

While the standard SQL SELECT syntax is supported for queries to an external database, there are a number of points to be borne in mind. You must prefix the name of the table with the keyword Database to indicate that the SELECT is to be targeted at the external database, rather than at a repeating structure in the message.

The basic form of database SELECT is:

```
SELECT ...  
  FROM Database.TABLE1  
  WHERE ...
```

If necessary, you can specify a schema name:

```
SELECT ...  
  FROM Database.SCHEMA.TABLE1  
  WHERE ...
```

where SCHEMA is the name of the schema in which the table TABLE1 is defined. Include the schema if the user ID under which you are running does not match the schema. For example, if your userID is USER1, the expression Database.TABLE1 is

equivalent to Database.USER1.TABLE1. However, if the schema associated with the table in the database is db2admin, you must specify Database.db2admin.TABLE1. If you do not include the schema, and this does not match your current user ID, the broker generates a runtime error when a message is processed by the message flow.

If, as in the two previous examples, a data source is not specified, TABLE1 must be a table in the default database specified by the node's data source property. To access data in a database other than the default specified on the node's data source property, you must specify the data source explicitly. For example:

```
SELECT ...
  FROM Database.DataSource.SCHEMA.TABLE1
 WHERE ...
```

Qualify references to column names with either the table name or the correlation name defined for the table by the FROM clause. So, where you could normally execute a query such as:

```
SELECT column1, column2 FROM table1
```

you must write one of the following two forms:

```
SELECT T.column1, T.column2 FROM Database.table1 AS T
```

```
SELECT table1.column1, table1.column2 FROM Database.table1
```

This is necessary in order to distinguish references to database columns from any references to fields in a message that might also appear in the SELECT:

```
SELECT T.column1, T.column2 FROM Database.table1
 AS T WHERE T.column3 = Body.Field2
```

You can use the AS clause to rename the columns returned. For example:

```
SELECT T.column1 AS price, T.column2 AS item
 FROM Database.table1 AS T WHERE...
```

The standard select all SQL option is supported in the SELECT clause. If you use this option, you must qualify the column names with either the table name or the correlation name defined for the table. For example:

```
SELECT T.* FROM Database.Table1 AS T
```

When you use ESQL procedure and function names within a database query, the positioning of these within the call affects how these names are processed. If it is determined that the procedure or function affects the results returned by the query, it is not processed as ESQL and is passed as part of the database call.

This applies when attempting to use a function or procedure name with the column identifiers within the SELECT statement.

For example, if you use a CAST statement on a column identifier specified in the Select clause, this is used during the database query to determine the data type of the data being returned for that column. An ESQL CAST is not performed to that ESQL data type, and the data returned is affected by the database interaction's interpretation of that data type.

If you use a function or procedure on a column identifier specified in the WHERE clause, this is passed directly to the database manager for processing.

The examples in the subsequent topics illustrate how the results sets of external database queries are represented in WebSphere Message Broker. The results of database queries are assigned to fields in a message using a Compute node.

A column function is a function that takes the values of a single column in all the selected rows of a table or message and returns a single scalar result.

Selecting data from database columns

You can configure a Compute, Filter, or Database node to select data from database columns and include it in an output message.

The following example assumes that you have a database table called USERTABLE with two char(6) data type columns (or equivalent), called Column1 and Column2. The table contains two rows:

	Column1	Column2
Row 1	value1	value2
Row 2	value3	value4

Configure the Compute, Filter, or Database node to identify the database in which you have defined the table. For example, if you are using the default database (specified on the data source property of the node), right-click the node, select **Open ESQL**, and code the following ESQL statements in the module for this node:

```
SET OutputRoot = InputRoot;
DELETE FIELD OutputRoot.*[<];
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.Column1, T.Column2 FROM Database.USERTABLE AS T);
```

This ESQL produces the following output message:

```
<Test>
  <Result>
    <Column1>value1</Column1>
    <Column2>value2</Column2>
  </Result>
  <Result>
    <Column1>value3</Column1>
    <Column2>value4</Column2>
  </Result>
</Test>
```

Figure 1. Output message

To trigger the SELECT, send a trigger message with an XML body that is of the following form:

```

<Test>
  <Result>
    <Column1></Column1>
    <Column2></Column2>
  </Result>
  <Result>
    <Column1></Column1>
    <Column2></Column2>
  </Result>
</Test>

```

The exact structure of the XML is not important, but the enclosing tag must be <Test> to match the reference in the ESQL. If the enclosing tag is not <Test>, the ESQL statements result in top-level enclosing tags being formed, which is not valid XML.

If you want to create an output message that includes all the columns of all the rows that meet a particular condition, use the SELECT statement with a WHERE clause:

```

-- Declare and initialize a variable to hold the
--     test vaue (in this case the surname Smith)
DECLARE CurrentCustomer STRING 'Smith';

-- Loop through table records to extract matching information
SET OutputRoot.XML.Invoice[] =
  (SELECT R FROM Database.USERTABLE AS R
   WHERE R.Customer.LastName = CurrentCustomer
  );

```

The message fields are created in the same order as the columns occur in the table.

If you are familiar with SQL in a database environment, you might expect to code SELECT *. This syntax is not accepted by the broker, because you must start all references to columns with a correlation name to avoid ambiguities with declared variables. Also, if you code SELECT I.*, this syntax is accepted by the broker, but the * is interpreted as the first child element, not all elements, as you might expect from other database SQL.

The assignment of the result set of a database into a parser-owned message tree requires the result set to exactly match the message definition. Because the generic XML parser is self-defining, the example creates a new subtree off the Invoice folder, and the parser can parse the new elements in the subtree. If the structure of the result set exactly matches the message definition, the result set can be assigned directly into the OutputRoot message body tree.

If the structure of the result set does not exactly match the MRM message definition, you must first assign the result set into a ROW data type, or an Environment tree that does not have a parser associated with it.

The required data can then be assigned to OutputRoot to build a message tree that conforms to the message definition.

Selecting data from a table in a case-sensitive database system:

If the database system is case sensitive, you must use an alternative approach. This approach is also necessary if you want to change the name of the generated field to something different:

```

SET OutputRoot = InputRoot;
SET OutputRoot.XML.Test.Result[] =
  (SELECT T.Column1 AS Column1, T.Column2 AS Column2
   FROM Database.USERTABLE AS T);

```

This example produces the output message as shown in Figure 1 on page 70. Ensure that references to the database columns (in this example, T.Column1 and T.Column2) are specified in the correct case to match the database definitions exactly. If you do not match the database definitions exactly (for example if you specify T.COLUMN1), the broker generates a runtime error. Column1 and Column2 are used in the SELECT statement to match the columns that you have defined in the database, although you can use any values here; the values do not have to match.

Selecting bitstream data from a database:

These samples show how to retrieve XML bitstream data from a database, and include it in an output message. See “INSERT statement” on page 249 for examples that show how you can insert bitstream data into a database.

In the following example, bitstream data is held in a database column with a BLOB data type. The database table used in the example (TABLE1) is the one created in the “INSERT statement” on page 249 examples, and the table contains the following columns:

- MSGDATA
- MSGCCSID
- MSGENCODING

If the bit stream from the database does not need to be interrogated or manipulated by the message flow, the output message can be constructed in the BLOB domain without any alteration.

In the following example, the message data, along with the MQMD header, is held in a database column with a BLOB data type. To re-create the message tree, including the MQMD header, from the bit stream, you can use a CREATE statement with a PARSE clause and DOMAIN('MQMD'). The output message can then be modified by the message flow:

```

SET Environment.Variables.DBResult = THE( SELECT T.* FROM Database.TABLE1 AS T );
DECLARE resultRef REFERENCE TO Environment.Variables.DBResult;

IF LASTMOVE(resultRef) THEN

  DECLARE outMsg BLOB resultRef.MSGDATA ;
  DECLARE outCCSID INT resultRef.MSGCCSID;
  DECLARE outEncoding INT resultRef.MSGENCODING;
  DECLARE outMsgPriority INT resultRef.MSGPRIORITY;
  DECLARE outMsgSeqNum INT resultRef.MSGSEQNUMBER;

  SET OutputRoot.Properties.CodedCharSetId = outCCSID;
  SET OutputRoot.Properties.Encoding = outEncoding ;

  CREATE LASTCHILD OF OutputRoot DOMAIN('MQMD') PARSE(outMsg, outEncoding, outCCSID);

  SET OutputRoot.MQMD.Version = MQMD_VERSION_2;

  SET OutputRoot.MQMD.Priority = outMsgPriority;
  SET OutputRoot.MQMD.MsgSeqNumber = outMsgSeqNum;

```



```

DECLARE HDRL INT ;
SET HDRL = LENGTH(BITSTREAM(OutputRoot.MQMD));
CREATE FIELD OutputRoot."BLOB"."BLOB";
DECLARE MSGB BLOB;
SET MSGB = SUBSTRING(outMsg FROM HDRL +1);
SET OutputRoot."BLOB"."BLOB" = MSGB;

END IF;

```

If you want to interrogate or manipulate a bit stream extracted from a database, you must re-create the original message tree. To re-create the XML message tree from the bit stream, you can use a CREATE statement with a PARSE clause. The output message can then be modified by the message flow.

For example, you might create a database table by using the following statement:

```

INSERT INTO Database.TABLE1(MSGDATA, MSGENCODING, MSGCCSID)
VALUES (msgBitStream, inEncoding, inCCSID);

```

The following code snippet shows how to re-create the message tree in the XMLNS domain by using the data read from the table:

```

CALL CopyMessageHeaders();
SET Environment.Variables.DBResult = THE( SELECT T.* FROM Database.TABLE1 AS T);
DECLARE resultRef REFERENCE TO Environment.Variables.DBResult;
IF LASTMOVE(resultRef) THEN
  DECLARE outCCSID INT resultRef.MSGCCSID;
  DECLARE outEncoding INT resultRef.MSGENCODING;
  DECLARE outMsg BLOB resultRef.MSGDATA;
  SET OutputRoot.Properties.CodedCharSetId = outCCSID;
  SET OutputRoot.Properties.Encoding = outEncoding;
  CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNS') PARSE(outMsg, outEncoding, outCCSID);
  -- Now modify the message tree fields
  SET OutputRoot.XMLNS.A.B = 4;
  SET OutputRoot.XMLNS.A.E = 5;
END IF;

```

In the following example, the data is held in a database column with a character data type, such as CHAR or VARCHAR. A cast is used to convert the data extracted from the database into BLOB format. If the bitstream data from the database does not need to be interrogated or manipulated by the message flow, the output message can be constructed in the BLOB domain, without any alteration.

```

CALL CopyMessageHeaders();
SET Environment.Variables.DBResult = THE( SELECT T.* FROM Database.TABLE1 AS T);
DECLARE resultRef REFERENCE TO Environment.Variables.DBResult;
IF LASTMOVE(resultRef) THEN
  DECLARE outCCSID INT resultRef.MSGCCSID;
  DECLARE outMsg BLOB CAST(resultRef.MSGDATA AS BLOB CCSID outCCSID);
  SET OutputRoot.Properties.CodedCharSetId = outCCSID;
  SET OutputRoot.Properties.Encoding = resultRef.MSGENCODING;
  SET OutputRoot.BLOB.BLOB = outMsg;
END IF;

```

In the following example, the data is held in a database column with a character data type, such as CHAR or VARCHAR. A cast is used to convert the data extracted from the database into BLOB format. To manipulate or interrogate this data within the message flow, you must re-create the original message tree. In this example, a CREATE statement with a PARSE clause is used to re-create the XML message tree in the XMLNS domain.

```

CALL CopyMessageHeaders();
SET Environment.Variables.DBResult = THE( SELECT T.* FROM Database.TABLE1 AS T);
DECLARE resultRef REFERENCE TO Environment.Variables.DBResult;
IF LASTMOVE(resultRef) THEN

```

```

DECLARE outCCSID INT resultRef.MSGCCSID;
DECLARE outEncoding INT resultRef.MSGENCODING;
DECLARE outMsg BLOB CAST(resultRef.MSGDATA AS BLOB CCSID outCCSID);
SET OutputRoot.Properties.CodedCharSetId = outCCSID;
SET OutputRoot.Properties.Encoding = outEncoding;
CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNS') PARSE(outMsg, outEncoding, outCCSID);
-- Now modify the message tree fields
SET OutputRoot.XMLNS.A.B = 4;
SET OutputRoot.XMLNS.A.E = 5;
END IF;

```

Accessing multiple database tables

You can refer to multiple tables that you have created in the same database. Use the FROM clause on the SELECT statement to join the data from the two tables.

The following example assumes that you have two database tables called USERTABLE1 and USERTABLE2. Both tables have two char(6) data type columns (or equivalent).

USERTABLE1 contains two rows:

	Column1	Column2
Row 1	value1	value2
Row 2	value3	value4

USERTABLE2 contains two rows:

	Column3	Column4
Row 1	value5	value6
Row 2	value7	value8

All tables referenced by a single SELECT function must be in the same database. The database can be either the default (specified on the Data Source property of the node) or another database (specified on the FROM clause of the SELECT function).

Configure the Compute, Database, or Filter node that you are using to identify the database in which you have defined the tables. For example, if you are using the default database, right-click the node, select **Open ESQL**, and code the following ESQL statements in the module for this node:

```

SET OutputRoot.XML.Test.Result[] =
    (SELECT A.Column1 AS FirstColumn,
         A.Column2 AS SecondColumn,
         B.Column3 AS ThirdColumn,
         B.Column4 AS FourthColumn
     FROM Database.USERTABLE1 AS A,
          Database.USERTABLE2 AS B
     WHERE A.Column1 = 'value1' AND
          B.Column4 = 'value8'
    );

```

This code results in the following output message content:

```

<Test>
  <Result>
    <FirstColumn>value1</FirstColumn>
    <SecondColumn>value2</SecondColumn>
    <ThirdColumn>value7</ThirdColumn>
    <FourthColumn>value8</FourthColumn>
  </Result>
</Test>

```

This example shows how to access data from two database tables. You can code more complex FROM clauses to access multiple database tables (although all the tables must be in the same database). You can also refer to one or more message trees, and can use SELECT to join tables with tables, messages with messages, or tables with messages. “Joining data from messages and database tables” on page 96 provides an example of how to merge message data with data in a database table.

If you specify an ESQL function or procedure on the column identifier in the WHERE clause, it is processed as part of the database query and not as ESQL.

Consider the following example:

```

SET OutputRoot.XML.Test.Result =
  THE(SELECT ITEM T.Column1 FROM Database.USERTABLE1 AS T
  WHERE UPPER(T.Column2) = 'VALUE2');

```

This code attempts to return the rows where the value of Column2 converted to uppercase is VALUE2. However, only the database manager can determine the value of T.Column2 for any given row, therefore it cannot be processed by ESQL before the database query is issued, because the WHERE clause determines the rows that are returned to the message flow.

Therefore, the UPPER is passed to the database manager to be included as part of its processing. However, if the database manager cannot process the token within the SELECT statement, an error is returned.

Changing database content

You can use Compute, Database, and Filter nodes to change the contents of a database by updating, inserting, or deleting data.

The following ESQL code includes statements that show all three operations. This code is appropriate for a Database and Filter node; if you create this code for a Compute node, use the correlation name InputRoot in place of Root.

```

IF Root.XMLNS.TestCase.Action = 'INSERT' THEN
  INSERT INTO Database.STOCK (STOCK_ID, STOCK_DESC, STOCK_QTY_HELD,
    BROKER_BUY_PRICE, BROKER_SELL_PRICE, STOCK_HIGH_PRICE, STOCK_HIGH_DATE,
    STOCK_HIGH_TIME) VALUES
    (CAST(Root.XMLNS.TestCase.stock_id AS INTEGER),
    Root.XMLNS.TestCase.stock_desc,
    CAST(Root.XMLNS.TestCase.stock_qty_held AS DECIMAL),
    CAST(Root.XMLNS.TestCase.broker_buy_price AS DECIMAL),
    CAST(Root.XMLNS.TestCase.broker_sell_price AS DECIMAL),
    Root.XMLNS.TestCase.stock_high_price,
    CURRENT_DATE,
    CURRENT_TIME);

ELSEIF Root.XMLNS.TestCase.Action = 'DELETE' THEN

  DELETE FROM Database.STOCK WHERE STOCK.STOCK_ID =
    CAST(Root.XMLNS.TestCase.stock_id AS INTEGER);

ELSEIF Root.XMLNS.TestCase.Action = 'UPDATE' THEN

  UPDATE Database.STOCK as A SET STOCK_DESC = Root.XMLNS.TestCase.stock_desc
    WHERE A.STOCK_ID = CAST(Root.XMLNS.TestCase.stock_id AS INTEGER);

END IF;

```

Checking returns to SELECT

If a SELECT function returns no data, or no further data, this result is handled as a normal situation and no error code is set in SQLCODE, regardless of the setting of the Throw Exception On Database Error and Treat Warnings As Errors properties on the current node.

To recognize that a SELECT function has returned no data, include ESQL that checks what has been returned. You can use various methods:

1. EXISTS

This ESQL returns a Boolean value that indicates if a SELECT function returned one or more values (TRUE), or none (FALSE).

```

IF EXISTS(SELECT T.MYCOL FROM Database.MYTABLE) THEN
  ...

```

2. CARDINALITY

If you expect an array in response to a SELECT, you can use CARDINALITY to calculate how many entries have been received.

```

SET OutputRoot.XMLNS.Testcase.Results[] = (
  SELECT T.MYCOL FROM Database.MYTABLE)
.....
IF CARDINALITY (OutputRoot.XMLNS.Testcase.Results[])> 0 THEN
  .....

```

3. IS NULL

If you have used either THE or ITEM keywords in your SELECT function, a scalar value is returned. If no rows have been returned, the value set is NULL. However, it is possible that the value NULL is contained within the column, and you might want to distinguish between these two cases.

Distinguish between cases by including COALESCE in the SELECT function, for example:

```

SET OutputRoot.XMLNS.Testcase.Results VALUE = THE (
  SELECT ITEM COALESCE(T.MYCOL, 'WAS NULL')
  FROM Database.MYTABLE);

```

If this example returns the character string WAS NULL, it indicates that the column contained NULL, and not that no rows were returned.

In previous releases, an SQLCODE of 100 was set in most cases if no data, or no further data, was returned. An exception was raised by the broker if you chose to handle database errors in the message flow.

Committing database updates

When you create a message flow that interacts with databases, you can choose whether the updates that you make are committed when the current node has completed processing, or when the current invocation of the message flow has terminated.

For each node, select the appropriate option for the Transaction property to specify when its database updates are to be committed:

- Choose Automatic (the default) if you want updates made in this node to be committed or rolled back as part of the whole message flow. The actions that you define in the ESQL module are performed on the message and it continues through the message flow. If the message flow completes successfully, the updates are committed. If the message flow fails, the message and the database updates are rolled back.
- Choose Commit if you want to commit the action of the node on the database, irrespective of the success or failure of the message flow as a whole. The database update is committed when the node processing is successfully completed, that is, after all ESQL has been processed, even if the message flow itself detects an error in a subsequent node that causes the message to be rolled back.

The value that you choose is implemented for the database tables that you have updated. You cannot select a different value for each table.

If you have set Transaction to Commit, the behavior of the message flow and the commitment of database updates can be affected by the use of the PROPAGATE statement.

If you choose to include a PROPAGATE statement in the node's ESQL that generates one or more output message from the node, the processing of the PROPAGATE statement is not considered complete until the entire path that the output message takes has completed. This path might include several other nodes, including one or more output nodes. Only then does the node that issues the PROPAGATE statement receive control back and its ESQL terminate. At that point, a database commit is performed, if appropriate.

If one of the nodes on the propagated path detects an error and throws an exception, the processing of the node in which you have coded the PROPAGATE statement never completes. If the error processing results in a rollback, the message flow and the database update in this node are rolled back. This behavior is consistent with the stated operation of the Commit option, but might not be the behavior that you expect.

Invoking stored procedures

To invoke a procedure that is stored in a database, use the ESQL CALL statement. The stored procedure must be defined by a CREATE PROCEDURE statement that has a Language clause of DATABASE and an EXTERNAL NAME clause that identifies the name of the procedure in the database and, optionally, the database schema to which it belongs.

When you invoke a stored procedure with the CALL statement, the broker ensures that the ESQL definition and the database definition match:

- The external name of the procedure must match a procedure in the database.
- The number of parameters must be the same.
- The type of each parameter must be the same.
- The direction of each parameter (IN, OUT, INOUT) must be the same.

The following restrictions apply to the use of stored procedures:

- Overloaded procedures are not supported. (An overloaded procedure is one that has the same name as another procedure in the same database schema with a different number of parameters, or parameters with different types.) If the broker detects that a procedure has been overloaded, it raises an exception.
- In an Oracle stored procedure declaration, you are not permitted to constrain CHAR and VARCHAR2 parameters with a length, and NUMBER parameters with a precision or scale, or both. Use %TYPE when you declare CHAR, VARCHAR and NUMBER parameters to provide constraints on a formal parameter.

You can also invoke a database stored procedure or a database user-defined function from a Mapping node. See Mapping a target element from database stored procedures or Mapping a target element from database user-defined functions.

Creating a stored procedure in ESQL:

When you define an ESQL procedure that corresponds to a database stored procedure, you can specify either a qualified name (where the qualifier is a database schema) or an unqualified name.

To create a stored procedure:

1. Code a statement similar to this example to create an unqualified procedure:

```
CREATE PROCEDURE myProc1(IN p1 CHAR) LANGUAGE DATABASE EXTERNAL NAME "myProc";
```

The EXTERNAL NAME that you specify must match the definition you have created in the database, but you can specify any name you choose for the corresponding ESQL procedure.

2. Code a statement similar to this example to create a qualified procedure:

```
CREATE PROCEDURE myProc2(IN p1 CHAR) LANGUAGE DATABASE EXTERNAL NAME "Schema1.myProc";
```

3. Code a statement similar to this example to create a qualified procedure in an Oracle package:

```
CREATE PROCEDURE myProc3(IN p1 CHAR) LANGUAGE DATABASE EXTERNAL
NAME "mySchema.myPackage.myProc";
```

For examples of stored procedure definitions in the database, see the “CREATE PROCEDURE statement” on page 220.

Calling a stored procedure:

1. Code a statement similar to this example to invoke an unqualified procedure:

```
CALL myProc1('HelloWorld');
```

Because it is not defined explicitly as belonging to any schema, the myProc1 procedure must exist in the default schema (the name of which is the user name used to connect to the data source) or the command fails.

2. The following example calls the myProc procedure in schema Schema1.

```
CALL myProc2('HelloWorld');
```

3. Code a statement similar to this example to invoke an unqualified procedure with a dynamic schema:

```
DECLARE Schema2 char 'mySchema2';  
CALL myProc1('HelloWorld') IN Database.{ 'Schema2' };
```

This statement calls the myProc1 procedure in database Schema2, overriding the default “username” schema.

Calling a stored procedure that returns two result sets:

To call a stored procedure that takes one input parameter and returns one output parameter and two result sets:

1. Define the procedure with a CREATE PROCEDURE statement that specifies one input parameter, one output parameter, and two result sets:

```
CREATE PROCEDURE myProc1 (IN P1 INT, OUT P2 INT)  
LANGUAGE DATABASE  
DYNAMIC RESULT SETS 2  
EXTERNAL NAME "myschema.myproc1";
```

2. To invoke the myProc1 procedure using a field reference, code:

```
/* using a field reference */  
CALL myProc1(InVar1, OutVar2, Environment.ResultSet1[],  
            OutputRoot.XMLNS.Test.ResultSet2[]);
```

3. To invoke the myProc1 procedure using a reference variable, code:

```
/* using a reference variable*/  
DECLARE cursor REFERENCE TO OutputRoot.XMLNS.Test;  
  
CALL myProc1(InVar1, cursor.OutVar2, cursor.ResultSet1[],  
            cursor.ResultSet2[]);
```

Coding ESQL to handle errors

When you process messages in a message flow, errors can have a number of different causes and the message flow designer must decide how to handle those errors.

Introduction

When you process messages in message flows, errors can have the following causes:

- External causes; for example, the incoming message is syntactically invalid, a database used by the flow has been shut down, or the power supply to the machine on which the broker is running fails.
- Internal causes; for example, an attempt to insert a row into a database table fails because of a constraint check, or a character string that is read from a database cannot be converted to a number because it contains alphabetic characters.

Internal errors can be caused by programs storing invalid data in the database, or by a flaw in the logic of a flow.

The message flow designer must decide how to handle errors.

Using default error-handling

The simplest strategy for handling ESQL errors is to do nothing, and use the broker's default behavior. The default behavior is to cut short the processing of the failing message, and to proceed to the next message. Input and output nodes provide options to control exactly what happens when processing is cut short.

If the input and output nodes are set to transactional mode, the broker restores the state before the message is processed:

1. The input message that has apparently been taken from the input queue is put back.
2. Any output messages that the flow has apparently written to output queues are discarded.

If the input and output nodes are not set to transactional mode:

1. The input message that was taken from the input queue is not put back.
2. Any output messages that the flow has written to output queues remain on the output queues.

Each of these strategies has its advantages. The transactional model preserves the consistency of data, while the non-transactional model maximizes the continuity of message processing. In the transactional model, the failing input message is put back onto the input queue, and the broker attempts to process it again. The most likely outcome of this scenario is that the message continues to fail until the retry limit is reached, at which point the message is placed on a dead letter queue. The reason for the failure to process the message is logged to the system event log (Windows) or syslog (UNIX[®]). Therefore, the failing message holds up the processing of subsequent valid messages, and is left unprocessed by the broker.

Most databases operate transactionally so that all changes that are made to database tables are committed if the processing of the message succeeds, or rolled back if it fails, therefore maintaining the integrity of data. An exception to this situation is if the broker itself, or a database, fails (for example, the power to the computers on which they are running is interrupted). In these cases, changes might be committed in some databases, but not in others, or the database changes might be committed but the input and output messages are not committed. If these possibilities concern you, make the flow coordinated and configure the databases that are involved.

Using customized error handling

The following list contains some general tips for creating customized error handlers.

- If you require something better than default error handling, the first step is to use a handler; see “DECLARE HANDLER statement” on page 240. Create one handler per node to intercept all possible exceptions (or all those that you can predict).
- Having intercepted an error, the error handler can use whatever logic is appropriate to handle it. Alternatively, it can use a THROW statement or node to create an exception, which could be handled higher in the flow logic, or even reach the input node, causing the transaction to be rolled back; see “Throwing an exception” on page 83.
- If a node generates an exception that is not caught by the handler, the flow is diverted to the Failure terminal, if one is connected, or handled by default error-handling if no Failure terminal is connected.

Use Failure terminals to catch unhandled errors. Attach a simple logic flow to the Failure terminal. This logic flow could consist of a database or Compute node that writes a log record to a database (possibly including the message's bit stream), or writes a record to the event log. The flow could also contain an output node that writes the message to a special queue.

The full exception tree is passed to any node that is connected to a Failure terminal; see Exception list tree structure.

- Your error handlers are responsible for logging each error in an appropriate place, such as the system event log.

For a detailed description of the options that you can use to process errors in a message flow, see *Handling errors in message flows*. For examples of what you can do, see “Throwing an exception” on page 83 and “Capturing database state” on page 84.

Writing code to detect errors

The following sections assume that the broker detects the error. It is possible, however, for the logic of the flow to detect an error. For example, when coding the flow logic, you could use the following elements:

- IF statements that are inserted specifically to detect situations that should not occur
- The ELSE clause of a case expression or statement to trap routes through the code that should not be possible

As an example of a flow logic-detected error, consider a field that has a range of possible integer values that indicate the type of message. It would not be good practice to leave to chance what would happen if a message were to arrive in which the field's value did not correspond to any known type of message. One way this situation could occur is if the system is upgraded to support extra types of message, but one part of the system is not upgraded.

Using your own logic to handle input messages that are not valid

Input messages that are syntactically invalid (and input messages that appear to be not valid because of erroneous message format information) are difficult to deal with, because the broker cannot determine the contents of the message. Typically, the best way to deal with these messages is to configure the input node to fully parse and validate the message. However, this configuration applies only to predefined messages; that is, MRM or IDoc.

If the input node is configured in this way, the following results are guaranteed if the input message cannot be parsed successfully:

- The input message never emerges from the node's normal output terminal (it goes to the Failure terminal).
- The input message never enters the main part of the message flow.
- The input message never causes any database updates.
- No messages are written to any output queues.

To deal with a failing message, connect a simple logic flow to the Failure terminal. The only disadvantage to this strategy is that if the normal flow does not require access to all of the message's fields, the forcing of complete parsing of the message affects performance.

Using your own logic to handle database errors

Database errors fall into three categories:

- The database is not working (for example, it is off line).
- The database is working but refuses your request (for example, a lock contention occurs).
- The database is working but it cannot do what you request (for example, read from a non-existent table).

If you require something better than default error handling, the first step is to use a handler (see “DECLARE HANDLER statement” on page 240) to intercept the exception. The handler can determine the nature of the failure from the SQL state that is returned by the database.

A database is not working

If a database is not working at all, and is essential to the processing of messages, there is typically not much that you can do. The handler, having determined the cause, might complete one or more of the following actions:

- Use the RESIGNAL statement to re-throw the original error, therefore allowing the default error handler to take over
- Use a different database
- Write the message to a special output queue

Be careful if you use an approach similar to this technique; the handler absorbs the exception, therefore all changes to other databases, or writes to queues, are committed.

A database refuses your request

The situation when a lock contention occurs is similar to the “Database not working” case because the database will have backed out *all* the database changes that you have made for the current message, not just the failing request. Therefore, unless you are sure that this was the only update, default error-handling is typically the best strategy, except possibly logging the error or passing the message to a special queue.

Impossible requests

An impossible request occurs when the database is working, but cannot complete the requested action. This type of error covers a wide variety of problems.

If, as discussed in the previous example, the database does not have a table of the name that the flow expects, default error-handling is typically the best strategy, except possibly logging the error or passing the message to a special queue.

Many other errors might be handled successfully, however. For example, an attempt to insert a row might fail because there is already such a row and the new row would be a duplicate. Or an attempt to update a row might fail because there is no such row (that is, the update action updated zero rows). In these cases, the handler can incorporate whatever logic you think appropriate. It might insert the missing row, or use the existing one (possibly making sure the values in it are suitable).

If you want an update of zero rows to be reported as an error, you must set the Treat warnings as errors property on the node to true, which is not the default setting.

Using your own logic to handle errors in output nodes

Errors that occur in MQOutput nodes report the nature of the error in the SQL state and give additional information in the *SQL native error* variable. Therefore, if something better than default error handling is required, the first step is to use a handler (see “DECLARE HANDLER statement” on page 240) to intercept the exception. Such a handler typically surrounds only a single PROPAGATE statement.

Using your own logic to handle other errors

Besides those errors covered above, a variety of other errors can occur. For example, an arithmetic calculation might overflow, a cast might fail because of the unsuitability of the data, or an access to a message field might fail because of a type constraint. The broker offers two programming strategies for dealing with these types of error.

- The error causes an exception that is either handled or left to roll back the transaction.
- The failure is recorded as a special value that is tested for later.

In the absence of a type constraint, an attempt to access a non-existent message field results in the value null. Null values propagate through expressions, making the result null. Therefore, if an expression, however complex, does not return a null value, you know that all the values that it needed to calculate its result were not null.

Cast expressions can have a default clause. If there is a default clause, casts fail quietly; instead of throwing an exception, they simply return the default value. The default value could be an innocuous number (for example, zero for an integer), or a value that is clearly invalid in the context (for example, -1 for a customer number). Null might be particularly suitable because it is a value that is different from all others, and it will propagate through expressions without any possibility of the error condition being masked.

Handling errors in other nodes

Exceptions that occur in other nodes (that is, downstream of a PROPAGATE statement) might be caught by handlers in the normal way. Handling such errors intelligently, however, poses a problem: another node was involved in the original error, therefore another node, and not necessarily the originator of the exception, is likely to be involved in handling the error.

To help in these situations, the Database and Compute nodes have four terminals called Out1, Out2, Out3, and Out4. In addition, the syntax of the PROPAGATE statement includes target expression, message source, and control clauses to give more control over these terminals.

Throwing an exception

If you detect an error or other situation in your message flow in which you want message processing to be ended, you can throw an exception in a message flow in two ways.

- Use the ESQL THROW EXCEPTION statement.
 - Include the THROW statement anywhere in the ESQL module for a Compute, Database, or Filter node. Use the options on the statement to code your own data to be inserted into the exception.
- Include a THROW node in your message flow.
 - Set the node properties to identify the source and content of the exception.

By using either statement options or node properties, you can specify a message identifier and values that are inserted into the message text to give additional information and identification to users who interpret the exception. You can specify any message in any catalog that is available to the broker. See Using error logging from a user-defined extension for more information.

The situations in which you might want to throw an exception are determined by the behavior of the message flow; decide when you design the message flow where this action might be appropriate. For example, you might want to examine the content of the input message to ensure that it meets criteria that cannot be detected by the input node (which might check that a particular message format is received).

The following example uses the example Invoice message to show how you can use the ESQL THROW statement. To check that the invoice number is within a particular range, throw an exception for any invoice message received that does not fall in the valid range.

```
--Check for invoice number lower than permitted range
IF Body.Invoice.InvoiceNo < 100000 THEN
    THROW USER EXCEPTION CATALOG 'MyCatalog' MESSAGE 1234 VALUES
    ('Invoice number too low', Body.Invoice.InvoiceNo);

-- Check for invoice number higher than permitted range
ELSEIF Body.InvoiceNo> 500000 THEN
    THROW USER EXCEPTION CATALOG 'MyCatalog' MESSAGE 1235 VALUES
    ('Invoice number too high', Body.Invoice.InvoiceNo);

ELSE DO
    -- invoice number is within permitted range
    -- complete normal processing
ENDIF;
```

Capturing database state

If an error occurs when the broker accesses an external database, you can either let the broker throw an exception during node processing or use ESQL statements to process the exception within the node itself.

Letting the broker throw an exception during node processing is the default action; ESQL processing in the current node is abandoned. The exception is then propagated backwards through the message flow until an enclosing catch node, or the input node for this message flow, is reached. If the exception reaches the input node, an active transaction is rolled back.

Using ESQL statements to process the exception within the node itself requires an understanding of database return codes and a logical course of action to take when an error occurs. To enable this inline database error processing, you must clear the Filter, Database, or Compute node's Throw Exception On Database Error property. If you clear this property, the node sets the database state indicators SQLCODE, SQLSTATE, SQLNATIVEERROR, and SQLERRORTXT, with appropriate information from the database manager instead of throwing an exception.

The indicators contain information only when an error (not a warning) occurs, unless you have selected the Treat Warnings As Errors property. If a database operation is successful, or returns success with information, the indicators contain their default success values.

You can use the values contained in these indicators in ESQL statements to make decisions about the action to take. You can access these indicators with the SQLCODE, SQLSTATE, SQLNATIVEERROR, and SQLERRORTXT functions.

If you are attempting inline error processing, check the state indicators after each database statement is executed to ensure that you catch and assess all errors. When processing the indicators, if you meet an error that you cannot handle inline, you

can raise a new exception either to deal with it upstream in a catch node, or to let it through to the input node so that the transaction is rolled back, for which you can use the `ESQL THROW` statement.

You might want to check for the special case in which a `SELECT` returns no data. This situation is not considered an error and `SQLCODE` is not set, therefore you must test explicitly for it; see “Checking returns to `SELECT`” on page 76.

Using ESQL to access database state indicators

The following ESQL example shows how to use the four database state functions, and how to include the error information that is returned in an exception:

```
DECLARE SQLState1 CHARACTER;
DECLARE SQLErrorText1 CHARACTER;
DECLARE SQLCode1 INTEGER;
DECLARE SQLNativeError1 INTEGER;

-- Make a database insert to a table that does not exist --
INSERT INTO Database.DB2ADMIN.NONEXISTENTTABLE (KEY,QMGR,QNAME)
        VALUES (45,'REG356','my TESTING 2');

--Retrieve the database return codes --
SET SQLState1 = SQLSTATE;
SET SQLCode1 = SQLCODE;
SET SQLErrorText1 = SQLERRORTXT;
SET SQLNativeError1 = SQLNATIVEERROR;

--Use the THROW statement to back out the database and issue a user exception--
THROW USER EXCEPTION MESSAGE 2950 VALUES
( 'The SQL State' , SQLState1 , SQLCode1 , SQLNativeError1 ,
SQLErrorText1 );
```

You do not have to throw an exception when you detect a database error; you might prefer to save the error information returned in the local environment tree, and include a Filter node in your message flow that routes the message to error or success subflows according to the values saved.

The following sample program provides another example of ESQL that uses these database functions:

- Airline Reservations

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Using the SELECT function

The `SELECT` function is a convenient and powerful tool for accessing fields and transforming data in a message tree.

The following topics show by example how to use the `SELECT` function to transform a variety of messages. The examples are based on an XML input message that has been parsed in the XMLNS domain. However, the techniques shown in these topics can be applied to any message tree.

- “Transforming a simple message” on page 86
- “Transforming a complex message” on page 89
- “Returning a scalar value in a message” on page 91
- “Joining data in a message” on page 93
- “Translating data in a message” on page 95

- “Joining data from messages and database tables” on page 96

Transforming a simple message

When you code the ESQL for a Compute node, use the SELECT function to transform simple messages.

This topic provides examples of simple message transformation. Review the examples and modify them for your own use. They are all based on the Invoice message as input.

Consider the following ESQL:

```
SET OutputRoot.XMLNS.Data.Output[] =
  (SELECT R.Quantity, R.Author FROM InputRoot.XMLNS.Invoice.Purchases.Item[] AS R
  );
```

When this ESQL code processes the Invoice message, it produces the following output message:

```
<Data>
  <Output>
    <Quantity>2</Quantity>
    <Author>Neil Bradley</Author>
  </Output>
  <Output>
    <Quantity>1</Quantity>
    <Author>Don Chamberlin</Author>
  </Output>
  <Output>
    <Quantity>1</Quantity>
    <Author>Philip Heller, Simon Roberts</Author>
  </Output>
</Data>
```

Three Output fields are present, one for each Item field, because SELECT creates an item in its result list for each item described by its FROM list. Within each Output field, a Field is created for each field named in the SELECT clause. These fields are in the order in which they are specified within the SELECT clause, not in the order in which they appear in the incoming message.

The R that is introduced by the final AS keyword is known as a correlation name. It is a local variable that represents in turn each of the fields addressed by the FROM clause. The name chosen has no significance. In summary, this simple transform does two things:

1. It discards unwanted fields.
2. It guarantees the order of the fields.

You can perform the same transform with a procedural algorithm:

```
DECLARE i INTEGER 1;
DECLARE count INTEGER CARDINALITY(InputRoot.XMLNS.Invoice.Purchases.Item[]);

WHILE (i <= count)
  SET OutputRoot.XMLNS.Data.Output[i].Quantity = InputRoot.XMLNS.Invoice.Purchases.Item[i].Quantity;
  SET OutputRoot.XMLNS.Data.Output[i].Author = InputRoot.XMLNS.Invoice.Purchases.Item[i].Author;
  SET i = i+1;
END WHILE;
```

These examples show that the SELECT version of the transform is much more concise. It also executes faster.


```

<Data>
  <Output>
    <Header>Start</Header>
    <Book>
      <Quantity>Number of books:2</Quantity>
      <Author>Neil Bradley:Name and Surname</Author>
    </Book>
    <Trailer>End</Trailer>
  </Output>
  <Output>
    <Header>Start</Header>
    <Book>
      <Quantity>Number of books:1</Quantity>
      <Author>Don Chamberlin:Name and Surname</Author>
    </Book>
    <Trailer>End</Trailer>
  </Output>
  <Output>
    <Header>Start</Header>
    <Book>
      <Quantity>Number of books:1</Quantity>
      <Author>Philip Heller, Simon Roberts:Name and Surname</Author>
    </Book>
    <Trailer>End</Trailer>
  </Output>
</Data>

```

As shown above, the AS clauses of the SELECT clause contain a path that describes the full name of the field that is to be created in the result. These paths can also specify (as is normal for paths) the type of field that is to be created. The following example transform specifies the field types. In this case, XML tagged data is transformed to XML attributes:

```

SET OutputRoot.XMLNS.Data.Output[] =
  (SELECT R.Quantity.* AS Book.(XML.Attribute)Quantity,
         R.Author.* AS Book.(XML.Attribute)Author
   FROM InputRoot.XMLNS.Invoice.Purchases.Item[] AS R
  );

```

Using the same Invoice message, the result is:

```

<Data>
  <Output>
    <Book Quantity="2" Author="Neil Bradley"/>
  </Output>
  <Output>
    <Book Quantity="1" Author="Don Chamberlin"/>
  </Output>
  <Output>
    <Book Quantity="1" Author="Philip Heller, Simon Roberts"/>
  </Output>
</Data>

```

Finally, you can use a WHERE clause to eliminate some of the results. In the following example a WHERE clause is used to remove results in which a specific criterion is met. An entire result is either included or excluded:

```

SET OutputRoot.XMLNS.Data.Output[] =
  (SELECT R.Quantity AS Book.Quantity,
         R.Author AS Book.Author
   FROM InputRoot.XMLNS.Invoice.Purchases.Item[] AS R
   WHERE R.Quantity = 2
  );

```


Using the same input message, the result is:

```
<Data>
  <Output>
    <Book>
      <Quantity>2</Quantity>
      <Author>Neil Bradley</Author>
    </Book>
  </Output>
</Data>
```

Transforming a complex message

When you code the ESQL for a Compute node, use the SELECT function for complex message transformation.

This topic provides examples of complex message transformation. Review the examples and modify them for your own use. They are all based on the Invoice message as input.

In this example, Invoice contains a variable number of Items. The transform is shown below:

```
SET OutputRoot.XMLNS.Data.Statement[] =
  (SELECT I.Customer.Title AS Customer.Title,
    I.Customer.FirstName || ' ' || I.Customer.LastName AS Customer.Name,
    COALESCE(I.Customer.PhoneHome, '') AS Customer.Phone,
    (SELECT II.Title AS Desc,
      CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
      II.Quantity AS Qty
    FROM I.Purchases.Item[] AS II
    WHERE II.UnitPrice > 0.0 ) AS Purchases.Article[],
    (SELECT SUM( CAST(II.UnitPrice AS FLOAT) *
      CAST(II.Quantity AS FLOAT) *
      1.6 )
    FROM I.Purchases.Item[] AS II ) AS Amount,
    'Dollars' AS Amount.(XML.Attribute)Currency

  FROM InputRoot.XMLNS.Invoice[] AS I
  WHERE I.Customer.LastName <> 'Brown'
);
```

The output message that is generated is:

```

<Data>
<Statement>
  <Customer>
    <Title>Mr</Title>
    <Name>Andrew Smith</Name>
    <Phone>01962818000</Phone>
  </Customer>
  <Purchases>
    <Article>
      <Desc Category="Computer" Form="Paperback" Edition="2">The XML Companion</Desc>
      <Cost>4.472E+1</Cost>
      <Qty>2</Qty>
    </Article>
    <Article>
      <Desc Category="Computer" Form="Paperback" Edition="2">
        A Complete Guide to DB2 Universal Database</Desc>
      <Cost>6.872E+1</Cost>
      <Qty>1</Qty>
    </Article>
    <Article>
      <Desc Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers Handbook</Desc>
      <Cost>9.5984E+1</Cost>
      <Qty>1</Qty>
    </Article>
  </Purchases>
  <Amount Currency="Dollars">2.54144E+2</Amount>
</Statement>
</Data>

```

This transform has nested SELECT clauses. The outer statement operates on the list of Invoices. The inner statement operates on the list of Items. The AS clause that is associated with the inner SELECT clause expects an array:

```

(SELECT II.Title                AS Desc,
      CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
      II.Quantity                AS Qty
 FROM I.Purchases.Item[] AS II
 WHERE II.UnitPrice > 0.0
 )
-- Note the use of [] in the next expression
AS Purchases.Article[],

```

This statement tells the outer SELECT clause to expect a variable number of Items in each result. Each SELECT clause has its own correlation name: I for the outer SELECT clause and II for the inner one. Each SELECT clause typically uses its own correlation name, but the FROM clause in the inner SELECT clause refers to the correlation name of the outer SELECT clause:

```

(SELECT II.Title                AS Desc,
      CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
      II.Quantity                AS Qty
-- Note the use of I.Purchases.Item in the next expression
 FROM I.Purchases.Item[] AS II
 WHERE II.UnitPrice > 0.0
 ) AS Purchases.Article[],

```

This statement tells the inner SELECT clause to work with the current Invoice's Items. Both SELECT clauses contain WHERE clauses. The outer one uses one criterion to discard certain Customers, and the inner one uses a different criterion to discard certain Items. The example also shows the use of COALESCE to prevent missing input fields from causing the corresponding output field to be missing. Finally, it also uses the column function SUM to add together the value of all Items in each Invoice. Column functions are discussed in "Referencing columns in a database" on page 68.

When the fields Desc are created, the whole of the input Title field is copied: the XML attributes and the field value. If you do not want these attributes in the output message, use the FIELDVALUE function to discard them; for example, code the following ESQL:

```
SET OutputRoot.XMLNS.Data.Statement[] =
  (SELECT I.Customer.Title AS Customer.Title,
    I.Customer.FirstName || ' ' || I.Customer.LastName AS Customer.Name,
    COALESCE(I.Customer.PhoneHome, '') AS Customer.Phone,
    (SELECT FIELDVALUE(II.Title) AS Desc,
      CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
      II.Quantity AS Qty
    FROM I.Purchases.Item[] AS II
    WHERE II.UnitPrice > 0.0 ) AS Purchases.Article[],
    (SELECT SUM( CAST(II.UnitPrice AS FLOAT) *
      CAST(II.Quantity AS FLOAT) *
      1.6 )
    FROM I.Purchases.Item[] AS II ) AS Amount,
    'Dollars' AS Amount.(XML.Attribute)Currency

  FROM InputRoot.XMLNS.Invoice[] AS I
  WHERE I.Customer.LastName <> 'Brown'
  );
```

That code generates the following output message:

```
<Data>
<Statement>
<Customer>
<Title>Mr</Title>
<Name>Andrew Smith</Name>
<Phone>01962818000</Phone>
</Customer>
<Purchases>
<Article>
<Desc>The XML Companion</Desc>
<Cost>4.472E+1</Cost>
<Qty>2</Qty>
</Article>
<Article>
<Desc>A Complete Guide to DB2 Universal Database</Desc>
<Cost>6.872E+1</Cost>
<Qty>1</Qty>
</Article>
<Article>
<Desc>JAVA 2 Developers Handbook</Desc>
<Cost>9.5984E+1</Cost>
<Qty>1</Qty>
</Article>
</Purchases>
<Amount Currency="Dollars">2.54144E+2</Amount>
</Statement>
</Data>
```

Returning a scalar value in a message

Use a SELECT statement to return a scalar value by including both the THE and ITEM keywords.

For example:

```
1 + THE(SELECT ITEM T.a FROM Body.Test.A[] AS T WHERE T.b = '123')
```

Use of the ITEM keyword:

The following example shows the use of the ITEM keyword to select one item and create a single value.

```
SET OutputRoot.MQMD = InputRoot.MQMD;  
  
SET OutputRoot.XMLNS.Test.Result[] =  
  (SELECT ITEM T.UnitPrice FROM InputBody.Invoice.Purchases.Item[] AS T);
```

When the Invoice message is received as input, the ESQL shown generates the following output message:

```
<Test>  
  <Result>27.95</Result>  
  <Result>42.95</Result>  
  <Result>59.99</Result>  
</Test>
```

When the ITEM keyword is specified, the output message includes a list of scalar values. Compare this message to the one that is produced if the ITEM keyword is omitted, in which a list of fields (name-value pairs) is generated:

```
<Test>  
  <Result>  
    <UnitPrice>27.95</UnitPrice>  
  </Result>  
  <Result>  
    <UnitPrice>42.95</UnitPrice>  
  </Result>  
  <Result>  
    <UnitPrice>59.99</UnitPrice>  
  </Result>  
</Test>
```

Effects of the THE keyword:

The THE keyword converts a list containing one item to the item itself.

The two previous examples both specified a list as the source of the SELECT in the FROM clause (the field reference has [] at the end to indicate an array), so typically the SELECT function generates a list of results. Because of this behavior, you must specify a list as the target of the assignment (thus the "Result[]" as the target of the assignment). However, you often know that the WHERE clause that you specify as part of the SELECT returns TRUE for only one item in the list. In this case use the THE keyword.

The following example shows the effect of using the THE keyword:

```
SET OutputRoot.MQMD = InputRoot.MQMD;  
  
SET OutputRoot.XMLNS.Test.Result =  
  THE (SELECT T.Publisher, T.Author FROM InputBody.Invoice.Purchases.Item[]  
      AS T WHERE T.UnitPrice = 42.95);
```

The THE keyword means that the target of the assignment becomes OutputRoot.XMLNS.Test.Result (the "[]" is not permitted). Its use generates the following output message:

```
<Test>
  <Result>
    <Publisher>Morgan Kaufmann Publishers</Publisher>
    <Author>Don Chamberlin</Author>
  </Result>
</Test>
```

Selecting from a list of scalars:

Consider the following sample input message:

```
<Test>
  <A>1</A>
  <A>2</A>
  <A>3</A>
  <A>4</A>
  <A>5</A>
</Test>
```

If you code the following ESQL statements to process this message:

```
SET OutputRoot.XMLNS.Test.A[] =
  (SELECT ITEM A from InputBody.Test.A[]
   WHERE CAST(A AS INTEGER) BETWEEN 2 AND 4);
```

the following output message is generated:

```
<A>2</A>
<A>3</A>
<A>4</A>
```

Joining data in a message

The FROM clause of a SELECT function is not restricted to having one item. Specifying multiple items in the FROM clause produces the typical Cartesian product joining effect, in which the result includes an item for all combinations of items in the two lists.

Using the FROM clause in this way produces the same joining effect as standard SQL.

The Invoice message includes a set of customer details, payment details, and details of the purchases that the customer makes. Code the following ESQL to process the input Invoice message:

```
SET OutputRoot.XMLNS.Items.Item[] =
  (SELECT D.LastName, D.Billing,
         P.UnitPrice, P.Quantity
   FROM InputBody.Invoice.Customer[] AS D,
         InputBody.Invoice.Purchases.Item[] AS P);
```

The following output message is generated:

```

<Items>
  <Item>
    <LastName>Smith</LastName>
    <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>S0213JR</PostCode>
    </Billing>
    <UnitPrice>27.95</UnitPrice>
    <Quantity>2</Quantity>
  </Item>
  <Item>
    <LastName>Smith</LastName>
    <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>S0213JR</PostCode>
    </Billing>
    <UnitPrice>42.95</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
  <Item>
    <LastName>Smith</LastName>
    <Billing>
      <Address>14 High Street</Address>
      <Address>Hursley Village</Address>
      <Address>Hampshire</Address>
      <PostCode>S0213JR</PostCode>
    </Billing>
    <UnitPrice>59.99</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
</Items>

```

Three results are produced, giving the number of descriptions in the first list (one) multiplied by the number of prices in the second (three). The results systematically work through all the combinations of the two lists. You can see this by looking at the *LastName* and *UnitPrice* fields selected from each result:

```

LastName Smith   UnitPrice 27.95
LastName Smith   UnitPrice 42.95
LastName Smith   UnitPrice 59.99

```

You can join data that occurs in a list and a non-list, or in two non-lists, and so on. For example:

```

OutputRoot.XMLNS.Test.Result1[] =
  (SELECT ... FROM InputBody.Test.A[], InputBody.Test.b);
OutputRoot.XMLNS.Test.Result1 =
  (SELECT ... FROM InputBody.Test.A, InputBody.Test.b);

```

The location of the [] in each case is significant. Any number of items can be specified in the FROM clause, not just one or two. If any of the items specify [] to indicate a list of items, the SELECT function returns a list of results (the list might contain only one item, but the SELECT function can return a list of items).

The target of the assignment must specify a list (so must end in []), or you must use the “THE function” on page 329 if you know that the WHERE clause guarantees that only one combination is matched.

Translating data in a message

You can translate data from one form to another.

A typical example of the requirement to translate data is if the items are known in one message by names, and in another message by numbers. For example:

Type Name	Type Code
Confectionary	2000
Newspapers	3000
Hardware	4000

Consider the following input message:

```
<Data>
  <Items>
    <Item>
      <Cat>1000</Cat>
      <Description>Milk Chocolate Bar</Description>
      <Type>Confectionary</Type>
    </Item>
    <Item>
      <Cat>1001</Cat>
      <Description>Daily Newspaper</Description>
      <Type>NewsPapers</Type>
    </Item>
    <Item>
      <Cat>1002</Cat>
      <Description>Kitchen Sink</Description>
      <Type>Hardware</Type>
    </Item>
  </Items>
  <TranslateTable>
    <Translate>
      <Name>Confectionary</Name>
      <Number>2000</Number>
    </Translate>
    <Translate>
      <Name>NewsPapers</Name>
      <Number>3000</Number>
    </Translate>
    <Translate>
      <Name>Hardware</Name>
      <Number>4000</Number>
    </Translate>
  </TranslateTable>
</Data>
```

This message has two sections; the first section is a list of items in which each item has a catalogue number and a type; the second section is a table for translating between descriptive type names and numeric type codes. Include a Compute node with the following transform:

```
SET OutputRoot.XMLNS.Result.Items.Item[] =
  (SELECT M.Cat, M.Description, T.Number As Type
   FROM
     InputRoot.XMLNS.Data.Items.Item[]           As M,
     InputRoot.XMLNS.Data.TranslateTable.Translate[] As T
   WHERE M.Type = T.Name
  );
```

The following output message is generated:

```

<Result>
  <Items>
    <Item>
      <Cat>1000</Cat>
      <Description>Milk Chocolate Bar</Description>
      <Type>2000</Type>
    </Item>
    <Item>
      <Cat>1001</Cat>
      <Description>Daily Newspaper</Description>
      <Type>3000</Type>
    </Item>
    <Item>
      <Cat>1002</Cat>
      <Description>Kitchen Sink</Description>
      <Type>4000</Type>
    </Item>
  </Items>
</Result>

```

In the result, each type name has been converted to its corresponding code. In this example, both the data and the translate table were in the same message tree, although this is not a requirement. For example, the translate table could be coded in a database, or might have been set up in LocalEnvironment by a previous Compute node.

Joining data from messages and database tables

You can use SELECT functions that interact with both message data and databases.

You can also nest a SELECT function that interacts with one type of data within a SELECT clause that interacts with the other type.

Consider the following input message, which contains invoice information for two customers:

```

<Data>
  <Invoice>
    <CustomerNumber>1234</CustomerNumber>
    <Item>
      <PartNumber>1</PartNumber>
      <Quantity>9876</Quantity>
    </Item>
    <Item>
      <PartNumber>2</PartNumber>
      <Quantity>8765</Quantity>
    </Item>
  </Invoice>
  <Invoice>
    <CustomerNumber>2345</CustomerNumber>
    <Item>
      <PartNumber>2</PartNumber>
      <Quantity>7654</Quantity>
    </Item>
    <Item>
      <PartNumber>1</PartNumber>
      <Quantity>6543</Quantity>
    </Item>
  </Invoice>
</Data>

```

Consider the following database tables, Prices and Addresses, and their contents:

PARTNO	PRICE
1	+2.50000E+001
2	+6.50000E+00

PARTNO	STREET	CITY	COUNTRY
1234	22 Railway Cuttings	East Cheam	England
2345	The Warren	Watership Down	England

If you code the following ESQL transform:

```
-- Create a valid output message
SET OutputRoot.MQMD = InputRoot.MQMD;

-- Select suitable invoices
SET OutputRoot.XMLNS.Data.Statement[] =
  (SELECT I.CustomerNumber           AS Customer.Number,
         A.Street                    AS Customer.Street,
         A.City                      AS Customer.Town,
         A.Country                   AS Customer.Country,

         -- Select suitable items
         (SELECT II.PartNumber AS PartNumber,
              II.Quantity  AS Quantity,
              PI.Price     AS Price
          FROM Database.db2admin.Prices AS PI,
              I.Item[]      AS II
          WHERE II.PartNumber = PI.PartNo ) AS Purchases.Item[]

  FROM Database.db2admin.Addresses AS A,
       InputRoot.XMLNS.Data.Invoice[] AS I

  WHERE I.CustomerNumber = A.PartNo
  );
```

the following output message is generated. The input message is augmented with the price and address information from the database table:

```

<Data>
  <Statement>
    <Customer>
      <Number>1234</Number>
      <Street>22 Railway Cuttings</Street>
      <Town>East Cheam</Town>
      <Country>England</Country>
    </Customer>
    <Purchases>
      <Item>
        <PartNumber>1</PartNumber>
        <Quantity>9876</Quantity>
        <Price>2.5E+1</Price>
      </Item>
      <Item>
        <PartNumber>2</PartNumber>
        <Quantity>8765</Quantity>
        <Price>6.5E+1</Price>
      </Item>
    </Purchases>
  </Statement>
  <Statement>
    <Customer>
      <Number>2345</Number>
      <Street>The Warren</Street>
      <Town>Watership Down</Town>
      <Country>England</Country>
    </Customer>
    <Purchases>
      <Item>
        <PartNumber>1</PartNumber>
        <Quantity>6543</Quantity>
        <Price>2.5E+1</Price></Item>
      <Item>
        <PartNumber>2</PartNumber>
        <Quantity>7654</Quantity>
        <Price>6.5E+1</Price>
      </Item>
    </Purchases>
  </Statement>
</Data>

```

You can nest the database SELECT clause within the message SELECT clause. In most cases, the code is not as efficient as the previous example, but you might find that it is better if the messages are small and the database tables are large.

```

-- Create a valid output message
SET OutputRoot.MQMD = InputRoot.MQMD;

-- Select suitable invoices
SET OutputRoot.XMLNS.Data.Statement[] =
  (SELECT I.CustomerNumber
      AS Customer.Number,

      -- Look up the address
      THE ( SELECT
            A.Street,
            A.City AS Town,
            A.Country
          FROM Database.db2admin.Addresses AS A
          WHERE A.PartNo = I.CustomerNumber
        )
      AS Customer,

      -- Select suitable items
      (SELECT
        II.PartNumber AS PartNumber,
        II.Quantity AS Quantity,

        -- Look up the price
        THE (SELECT ITEM P.Price
            FROM Database.db2admin.Prices AS P
            WHERE P.PartNo = II.PartNumber
          )
        AS Price

      FROM I.Item[] AS II
      ) AS Purchases.Item[]

  FROM InputRoot.XMLNS.Data.Invoice[] AS I
);

```

Manipulating messages in the XML domains

The following topics contain instructions on manipulating messages in the XMLNSC, XMLNS, and XML domains.

- “Working with XML messages”
- “Manipulating messages in the XMLNSC domain” on page 109
- “Manipulating messages in the XMLNS domain” on page 122
- “Manipulating messages in the XML domain” on page 132

For information about dealing with MRM XML messages, see “Manipulating messages in the MRM domain” on page 132.

Working with XML messages

The following topics provide information about typical tasks for processing XML messages.

Some of this information is available publicly in Web pages and online tutorials. If you are new to XML, you will find it useful to also read about the XML standard.

- “Constructing an XML message tree” on page 100
- “Working with namespaces” on page 100
- “Working with binary data” on page 101
- “XMLNSC: Working with CDATA” on page 103
- “XMLNSC: Working with XML messages and bit streams” on page 104
- “Working with large XML messages” on page 105

For details about XML Schema, see XML Schema Part 0: Primer on the World Wide Web Consortium (W3C) Web site.

Constructing an XML message tree:

When constructing an XML message tree, consider the order of fields in the tree.

Order of fields in the message tree

When you create an XML output message in a Compute node, the order of your lines of ESQL code is important, because the message elements are created in the order that you code them.

Consider the following XML message:

```
<Order>
  <ItemNo>1</ItemNo>
  <Quantity>2</Quantity>
</Order>
```

If you want to add a DocType Declaration to this, insert the DocType Declaration before you copy the input message to the output message.

For example:

```
SET OutputRoot.XMLNS.(XML.XmlDecl) = '';
SET OutputRoot.XMLNS.(XML.XmlDecl).(XML.Version) = '1.0';
SET OutputRoot.XMLNS.(XML.DocTypeDecl)Order = '';
SET OutputRoot.XMLNS.(XML.DocTypeDecl).(XML.SystemId) = 'NewDtdName.dtd';
SET OutputRoot = InputRoot;
-- more ESQL --
```

If you put the last statement to copy the input message before the XML-specific statements, the following XML is generated for the output message.

```
<Order>
  <ItemNo>1</ItemNo>
  <Quantity>2</Quantity>
</Order>
<?xml version="1.0"?>
```

This is not well-formed XML and causes an error when it is written from the message tree to a bit stream in the output node.

Setting the field type

If you copy a message tree from input to output without changing the domain, most of the syntax elements will be created by the parser (XMLNSC or XMLNS) and their field types will be correct. However, if you construct your message tree from a database query, or from another parser's message tree, you must ensure that you identify each syntax element correctly by using its field type. You can find full details of the field type constants used by XMLNSC and XMLNS in the following topics:

- “XMLNSC: Using field types” on page 111
- XML constructs

Working with namespaces:

The following example shows how to use ESQL to work with namespaces.

Namespace constants are declared at the start of the main module so that you can use prefixes in the ESQL statements instead of the full URIs of the namespaces. The namespace constants affect only the ESQL; they do not control the prefixes that are generated in the output message. The prefixes in the generated output message are controlled by namespace declarations in the message tree. You can include namespace declarations in the tree using the XML.NamespaceDecl field type. These elements are then used to generate namespace declarations in the output message.

When the output message is generated, if the parser encounters a namespace for which it has no corresponding namespace declaration, a prefix is automatically generated using prefixes of the form NSn where n is a positive integer.

```
CREATE COMPUTE MODULE xmlns_doc_flow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
CALL CopyMessageHeaders();
-- Declaration of namespace constants --
These are only used by ESQL

DECLARE sp1 NAMESPACE 'http://www.ibm.com/space1';
DECLARE sp2 NAMESPACE 'http://www.ibm.com/space2';
DECLARE sp3 NAMESPACE 'http://www.ibm.com/space3';

-- Namespace declaration for prefix 'space1'
SET OutputRoot.XMLNS.message.(XML.NamespaceDecl)xmlns:space1 = 'http://www.ibm.com/space1';
SET OutputRoot.XMLNS.message.sp1:data1 = 'Hello!';

-- Default namespace declaration ( empty prefix )
SET OutputRoot.XMLNS.message.sp2:data2.(XML.NamespaceDecl)xmlns = 'http://www.ibm.com/space2';
SET OutputRoot.XMLNS.message.sp2:data2.sp2:subData1 = 'Hola!';
SET OutputRoot.XMLNS.message.sp2:data2.sp2:subData2 = 'Guten Tag!';
SET OutputRoot.XMLNS.message.sp3:data3 = 'Bonjour!';
RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders()
BEGIN
DECLARE I INTEGER 1;
DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
WHILE I < J DO SET OutputRoot.*[I] = InputRoot.*[I];
    SET I = I + 1;
END WHILE;
END;
END MODULE;
```

When this ESQL is processed, the following output message is generated:

```
<message xmlns:space1="http://www.ibm.com/space1">
  <space1:data1>Hello!</space1:data1>
  <data2 xmlns="http://www.ibm.com/space2">
    <subData1>Hola!</subData1>
    <subData2>Guten Tag!</subData2>
  </data2>
  <NS1:data3 xmlns="http://www.ibm.com/space3">Bonjour!</NS1:data3>
</message>
```

Working with binary data:

If you need to include binary data or non-valid characters in your XML documents, the safest method is to encode the data as a binary string.

Binary encodings for XML

There are two methods of encoding binary data in an XML document.

```
hexBinary: <nonXMLChars>0001020304050607080B0C0E0F</nonXMLChars>
```

```
base64Binary: <nonXMLChars>AAECAwQFBgcICwwODw==</nonXMLChars>
```

The base64Binary encoding makes better use of the available XML characters, and on average a base64-encoded binary field is 2/3 the size of its hexBinary equivalent. Base64Binary is widely used by the MIME format and by various XML-based standards.

You might prefer to use the simpler hexBinary encoding if you are sending data to applications that cannot decode base64 data, and if the increase in message size is not important.

Parsing binary data

The most straightforward way to parse any binary data is to use the XMLNSC parser with a message set:

1. Locate or construct an XML Schema that describes your input XML.
2. Import the XML Schema to create a message definition file.
3. In your message flow, set the node properties as follows:
 - On the Default page, set *Message Domain* to XMLNSC and *Message Set* to the name of your message set.
 - On the Validation page, set *Validation* to Content and Value.
 - In the XMLNSC properties, select the check box option *Build tree using XML Schema types*.

The XMLNSC parser automatically decodes your hexBinary or base64Binary data, being guided by the simple type of the element or attribute that contains the binary data. The message tree will contain the decoded BLOB value.

If you are using the XMLNS domain, you must parse the binary data as a string. It appears in the message tree as a CHARACTER value. If the data is encoded as hexBinary, you can use the ESQL CAST function to convert it to a BLOB value. If the data is encoded as base64Binary, the easiest approach is to call a static Java method from ESQL to decode the base64 data into a BLOB value.

Generating binary data

You can generate binary data in your output XML in either hexBinary or base64Binary encoding.

For hexBinary, use the ESQL CAST statement to convert your BLOB data to a hexBinary string.

For base64Binary, you have two options:

- Call a static Java method to encode your BLOB data as base64.
- Use the XMLNSC parser, and modify the type field on the syntax element, as shown in this example:

```
-- ESQL code to generate base64-encoded binary data
DECLARE myBLOB BLOB;
-- Populate myBLOB with your binary data
```

```

CREATE LASTCHILD OF OutputRoot.XMLNSC.message
  TYPE BITOR(XMLNSC.Attribute, XMLNSC.base64Binary)
  NAME myBase64Element
  VALUE myBLOB;

```

Note that setting the field type to XMLNSC.base64Binary does not change the logical value in the message tree. In your message flow, it is still a BLOB, and if you ask for its string representation, it is reported as a hexBinary string. However, when the message tree is converted to a bit stream (in an output node, or by a call to ASBITSTREAM) the base64 conversion is performed automatically, and the output XML contains the correct base64 string.

XMLNSC: Working with CData:

A CData section can be used to embed an XML document within another XML document.

What is a CData section?

An XML element can contain text content:

```
<element>text content</element>
```

However, some characters cannot appear in that content. In particular, '<' and '&' both have special meaning to an XML parser. If they are included in the text content of an element, they change the meaning of the XML document.

For example, this is a badly formed XML document:

```
<element><text><content></element>
```

There are two ways to make the XML well-formed:

1. Use character entities:

```
<element>&lt;text&gt;&lt;content&gt;</element>
```

2. Use a CData section:

```
<element><![CDATA[<text><content>]]></element>
```

What can you use a CData section for?

In a CData section, you can include XML markup in the value of an element. However, non-valid XML characters cannot be included. Binary data also cannot be included in a CData section.

The most common use for CData is to embed one XML document within another.

For example:

```

<outer>
  <embedXML>
    <![CDATA[<innerMsg></innerMsg>]]>
  </embedXML>
</outer>

```

You can even embed a badly-formed XML document in this way, because the XML parser does not attempt to parse the content of a CData section.

```

<outer>
  <embedXML>
    <![CDATA[<badXML></wrongClosingTag>]]>
  </embedXML>
</outer>

```

The following items are not valid within a CDATA section:

- Non-valid XML characters (see <http://www.w3.org/TR/2006/REC-xml-20060816/#charsets>)
- The text string ']]>' (because this terminates the CDATA section)

Because of these restrictions, do not use a CDATA section to include arbitrary text in your XML document, and do not try to use a CDATA section to hold binary data (unless it is encoded as hexBinary or base64Binary).

How do you add a CDATA section to an output XML message?

Consider the following input message :

```
<TestCase>
  <Folder>
    <Field1>Value1</Field1>
    <Field2>Value2</Field2>
    <Field3>Value3</Field3>
  </Folder>
</TestCase>
```

The following ESQL shows how to serialize a whole message:

```
DECLARE wholeMsgBlob BLOB
  ASBITSTREAM(InputRoot.XMLNSC,
              InputRoot.Properties.Encoding,
              InputRoot.Properties.CodedCharSetId );
DECLARE wholeMsgChar CHAR
  CAST(wholeMsgBlob AS CHAR CCSID InputRoot.Properties.CodedCharSetId);
SET OutputRoot.XMLNSC.Output.(XMLNSC.CDataField)Field1 = wholeMsgChar;
```

This example serializes the InputRoot.XMLNSC.TestCase.Folder portion of the message tree.

If the output message tree were examined before an MQOutput node, this would show :

```
(0x01000010):XML      = (
  (0x01000000):Output = (
    (0x01000000):Field1 = (
      (0x02000001): = '<TestCase><Folder><Field1>Value1</Field1><Field2>Value2</Field2>
        <Field3>Value3</Field3></Folder><Folder2><Field1>Value1</Field1>
          <Field2>Value2</Field2><Field3>Value3</Field3></Folder2></TestCase>'
      )
    )
  )
)
```

As can be seen, each CDATA section contains a single scalar value that is the character representation of the portion of the XML message that is required.

This tree produces the following XML output message :

```
<Output>
  <Field1><![CDATA[<TestCase><Folder><Field1>Value 1</Field1>
    <Field2>Value 2</Field2>
    <Field3>Value 3</Field3></Folder>
    <Folder2><Field1>Value 1</Field1>
    <Field2>Value 2</Field2>
    <Field3>Value 3</Field3></Folder2>
  </Testcase>]]</Field1>
</Output>
```

XMLNSC: Working with XML messages and bit streams:

Use the ASBITSTREAM function and the CREATE statement to manage XML message content.

The ASBITSTREAM function

If you code the ASBITSTREAM function with the parser mode option set to RootBitStream to parse a message tree to a bit stream, the result is an XML document that is built from the children of the target element in the normal way. This algorithm is identical to that used to generate the normal output bit stream. Because the target element is not included in the output bit stream, you must ensure that the children of the element follow the constraints for an XML document.

One constraint is that there must be only one body element in the message. You can use a well-formed bit stream obtained in this way to re-create the original tree using a CREATE statement with a PARSE clause.

If you code the ASBITSTREAM function with the parser mode option set to FolderBitStream to parse a message tree to a bit stream, the generated bit stream is an XML document built from the target element and its children. Any DocTypeDecl or XmlDecl elements are ignored, and the target element itself is included in the generated bit stream.

The advantage of this mode is that the target element becomes the body element of the document, and that body element can have multiple elements nested within it. Use this mode to obtain a bit stream description of arbitrary sub-trees owned by an XML parser. You can use bit streams obtained in this way to re-create the original tree using a CREATE statement with a PARSE clause, and a mode of FolderBitStream.

For further information about the ASBITSTREAM function, and some examples of its use, see “ASBITSTREAM function” on page 316.

The CREATE statement with a PARSE clause

If you code a CREATE statement with a PARSE clause with the parser mode option set to RootBitStream to parse a bit stream to a message tree, the expected bit stream is a normal XML document. A field in the tree is created for each field in the document. This algorithm is identical to that used when parsing a bit stream from an input node. In particular, an element named 'XML', 'XMLNS', or 'XMLNSC' is created as the root element of the tree, and all the content in the message is created as children of that root.

If you code a CREATE statement with a PARSE clause with the parser mode option set to FolderBitStream to parse a bit stream to a message tree, the expected bit stream is a normal XML document. Any content outside the body element (such as an XML declaration or doctype) is discarded. The first element created during the parse corresponds to the body of the XML document, and from there the parse proceeds as normal.

For further information about the CREATE statement, and examples of its use, see “CREATE statement” on page 201.

Working with large XML messages:

When an input bit stream is parsed and a logical tree is created, the tree representation of an XML message is typically bigger, and in some cases much bigger, than the corresponding bit stream.

The reasons for this expansion include:

- The addition of the pointers that link the objects together
- Translation of character data into Unicode, which can double the size
- The inclusion of field names that might have been implicit in the bit stream
- The presence of control data that is associated with the broker's operation

Manipulating a large message tree can require a lot of storage. If you design a message flow that handles large messages that are made up of repeating structures, you can code ESQL statements that help to reduce the storage load on the broker. These statements support both random and sequential access to the message, but assume that you do not need access to the whole message at one time.

These ESQL statements cause the broker to perform limited parsing of the message, and to keep in storage at one time, only that part of the message tree that reflects a single record. If your processing requires you to retain information from record to record (for example, to calculate a total price from a repeating structure of items in an order), you can either declare, initialize, and maintain ESQL variables, or you can save values in another part of the message tree; for example, in `LocalEnvironment`.

This technique reduces the memory that is used by the broker to that needed to hold the full input and output bit streams, plus that needed for the message trees of just one record. This technique also provides memory savings when even a small number of repeats is encountered in the message. The broker uses partial parsing and the ability to parse specified parts of the message tree, to and from the corresponding part of the bit stream.

To use these techniques in your Compute node, apply these general techniques:

- Copy the body of the input message as a bit stream to a special folder in the output message. The copy creates a modifiable copy of the input message that is not parsed and that therefore uses a minimum amount of memory.
- Avoid any inspection of the input message, which avoids the need to parse the message.
- Use a loop and a reference variable to step through the message one record at a time. For each record:
 - Use normal transforms to build a corresponding output subtree in a second special folder.
 - Use the `ASBITSTREAM` function to generate a bit stream for the output subtree. The generated bit stream is stored in a `BitStream` element that is placed in the position in the output subtree that corresponds to its required position in the final bit stream.
 - Use the `DELETE` statement to delete both the current input and output record message trees when you have completed their manipulation.
 - When you have completed the processing of all records, detach the special folders so that they do not appear in the output bit stream.

You can vary these techniques to suit the processing that is required for your messages.

The following ESQL code provides an example of one implementation, and is a modification of the ESQL example in “Transforming a complex message” on page 89. It uses a single SET statement with nested SELECT functions to transform a message that contains nested, repeating structures.

```
-- Copy the MQMD header
SET OutputRoot.MQMD = InputRoot.MQMD;

-- Create a special folder in the output message to hold the input tree
-- Note : SourceMessageTree is the root element of an XML parser
CREATE LASTCHILD OF OutputRoot.XMLNS.Data DOMAIN 'XMLNS' NAME 'SourceMessageTree';

-- Copy the input message to a special folder in the output message
-- Note : This is a root to root copy which will therefore not build trees
SET OutputRoot.XMLNS.Data.SourceMessageTree = InputRoot.XMLNS;

-- Create a special folder in the output message to hold the output tree
CREATE FIELD OutputRoot.XMLNS.Data.TargetMessageTree;

-- Prepare to loop through the purchased items
DECLARE sourceCursor REFERENCE TO OutputRoot.XMLNS.Data.SourceMessageTree.Invoice;
DECLARE targetCursor REFERENCE TO OutputRoot.XMLNS.Data.TargetMessageTree;
DECLARE resultCursor REFERENCE TO OutputRoot.XMLNS.Data;
DECLARE grandTotal FLOAT 0.0e0;

-- Create a block so that it's easy to abandon processing
ProcessInvoice: BEGIN
-- If there are no Invoices in the input message, there is nothing to do
IF NOT LASTMOVE(sourceCursor) THEN
    LEAVE ProcessInvoice;
END IF;

-- Loop through the invoices in the source tree
InvoiceLoop : LOOP
-- Inspect the current invoice and create a matching Statement
SET targetCursor.Statement = THE (SELECT 'Monthly' AS (XML.Attribute)Type,
    'Full' AS (0x03000000)Style[1],
    I.Customer.FirstName AS Customer.Name,
    I.Customer.LastName AS Customer.Surname,
    (SELECT
        FIELDVALUE(II.Title) AS Title,
        CAST(II.UnitPrice AS FLOAT) * 1.6 AS Cost,
        II.Quantity AS Qty
    FROM I.Purchases.Item[] AS II
    WHERE II.UnitPrice > 0.0) AS Purchases.Article[],
    (SELECT
        SUM( CAST(II.UnitPrice AS FLOAT) *
            CAST(II.Quantity AS FLOAT) *
            1.6
        )
    FROM I.Purchases.Item[] AS II) AS Amount,
    'Dollars' AS Amount.(XML.Attribute)Currency
    FROM sourceCursor AS I
    WHERE I.Customer.LastName <> 'White');

-- Turn the current Statement into a bit stream
DECLARE StatementBitStream BLOB
ASBITSTREAM(targetCursor.Statement OPTIONS FolderBitStream);
-- If the SELECT produced a result
-- (that is, it was not filtered out by the WHERE clause),
-- process the Statement
IF StatementBitStream IS NOT NULL THEN
-- create a field to hold the bit stream in the result tree
CREATE LASTCHILD OF resultCursor
    Type XML.BitStream
    NAME 'StatementBitStream'
    VALUE StatementBitStream;
```

```

-- Add the current Statement's Amount to the grand total
-- Note that the cast is necessary because of the behavior
-- of the XML syntax element
  SET grandTotal = grandTotal
    + CAST(targetCursor.Statement.Amount AS FLOAT);
END IF;

-- Delete the real Statement tree leaving only the bit stream version
DELETE FIELD targetCursor.Statement;

-- Step onto the next Invoice,
-- removing the previous invoice and any
-- text elements that might have been
-- interspersed with the Invoices

REPEAT
  MOVE sourceCursor NEXTSIBLING;
  DELETE PREVIOUS SIBLING OF sourceCursor;
  UNTIL (FIELDNAME(sourceCursor) = 'Invoice')
    OR (LASTMOVE(sourceCursor) = FALSE)
  END REPEAT;

-- If there are no more invoices to process, abandon the loop
IF NOT LASTMOVE(sourceCursor) THEN
  LEAVE InvoiceLoop;
END IF;

END LOOP InvoiceLoop;
END ProcessInvoice;

-- Remove the temporary source and target folders
DELETE FIELD OutputRoot.XMLNS.Data.SourceMessageTree;
DELETE FIELD OutputRoot.XMLNS.Data.TargetMessageTree;

-- Finally add the grand total
SET resultCursor.GrandTotal = grandTotal;

```

This ESQL code produces the following output message:

```

<Data>
  <Statement Type="Monthly" Style="Full">
    <Customer>
      <Name>Andrew</Name>
      <Surname>Smith</Surname>
      <Title>Mr</Title>
    </Customer>
    <Purchases>
      <Article>
        <Title>The XML Companion</Title>
        <Cost>4.472E+1</Cost>
        <Qty>2</Qty>
      </Article>
      <Article>
        <Title>A Complete Guide to DB2 Universal Database</Title>
        <Cost>6.872E+1</Cost>
        <Qty>1</Qty>
      </Article>
      <Article>
        <Title>JAVA 2 Developers Handbook</Title>
        <Cost>9.5984E+1</Cost>
        <Qty>1</Qty>
      </Article>
    </Purchases>
    <Amount Currency="Dollars">2.54144E+2</Amount>
  </Statement>
  <GrandTotal>2.54144E+2</GrandTotal>
</Data>

```

Manipulating messages in the XMLNSC domain

This topic provides information about on how to write ESQL for processing messages that belong to the XMLNSC domain, and that are parsed by the XMLNSC parser. Refer to XMLNSC parser for background information.

The following topics provide detailed information about the structure of the message tree that the XMLNSC parser builds, and the field types that it uses.

- “XMLNSC: The XML declaration”
- “XMLNSC: The inline DTD” on page 110
- “XMLNSC: The message body” on page 111
- “XMLNSC: XML Schema support” on page 119

Information for users who are migrating from XML, XMLNS, or MRM XML is available in “Migrating to XMLNSC” on page 121.

Further information about processing XML messages can be found in “Working with XML messages” on page 99.

XMLNSC: The XML declaration:

The XML declaration is represented in the message tree by a syntax element with field type XMLNSC.XMLDeclaration.

If an XML declaration is created by the XMLNSC parser, its name is 'XmlDeclaration'. However, when a message tree is being produced, the name is not important: the XMLNSC parser recognizes this syntax element by its field type only. The following example shows a typical declaration:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

The XML Declaration has three optional attributes; Version, Standalone, and Encoding. The XMLNSC parser does not define special field types for these attributes. Instead, they are identified by their name, and by their position as a child of the XML Declaration element.

ESQL example code to create an XML declaration

To construct the XML declaration that is shown in the previous example, code the following ESQL statements:

```
CREATE FIRSTCHILD OF OutputRoot.XMLNSC TYPE XMLNSC.XmlDeclaration;
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.(XMLNSC.Attribute)Version = '1.0';
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.(XMLNSC.Attribute)Encoding = 'UTF-8';
SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.(XMLNSC.Attribute)StandAlone = 'yes';
```

The first line is optional; if it is omitted, the XMLNSC.XMLDeclaration element is automatically created when it is referenced by the second line.

Java example code to create an XML declaration

To construct the XML declaration that is shown in the previous example, write the following Java code:

```
//Create the XML domain root node
MElement xmlRoot =
    root.createElementAsLastChild(MbXMLNSC.PARSER_NAME);
```

```

//Create the XML declaration parent node
MbElement xmlDecl =
    xmlRoot.createElementAsFirstChild(MbXMLNSC.XML_DECLARATION);

xmlDecl.setName("XmlDeclaration");

MbElement version =
    xmlDecl.CreateElementAsFirstChild(MbXMLNSC.ATTRIBUTE, "Version", "1.0");
MbElement encoding =
    xmlDecl.CreateElementAsFirstChild(MbXMLNSC.ATTRIBUTE, "Encoding", "utf-8");
MbElement standalone =
    xmlDecl.CreateElementAsFirstChild(MbXMLNSC.ATTRIBUTE, "Standalone", "yes");

```

Note: In both the ESQL example and the Java example, 'Version', 'StandAlone', and 'Encoding' can all be written in lowercase.

XMLNSC: The inline DTD:

When parsing an XML document that has an inline DTD, the XMLNSC parser does not put the DTD information into the message tree. However, using ESQL you can add XML entity definitions to the message tree, and these are used when the message tree is produced by the XMLNSC parser.

ESQL example code for entity definition and entity reference

This example assumes that InputRoot.XMLNSC has been created from the following XML message:

```
<BookInfo dtn="BookInfo" edn="author" edv="A.N.Other"/>
```

The following output message is generated:

```
<!DOCTYPE BookInfo [<!ENTITY author "A.N.Other">]>
<BookInfo><entref>&author;</entref></BookInfo>
```

The ESQL to create the output message is:

```

DECLARE cursor REFERENCE TO InputRoot.XMLNSC.BookInfo;
-- Create <!DOCTYPE BOOKInfo ...
SET OutputRoot.XMLNSC.(XMLNSC.DocumentType)* NAME = cursor.dtn;
-- Create <!ENTITY author "A.N.Other" > ...
SET
OutputRoot.XMLNSC.(XMLNSC.DocumentType)* NAME = 'author' VALUE = cursor.edv;
-- Create the entity reference
SET OutputRoot.XMLNSC.BookInfo.(XMLNSC.EntityReference)entref = 'author';

```

Java example code to create an XML declaration

To construct the XML declaration that is shown above, the following Java is required:

```

//Create the XML domain root node
MbElement xmlRoot =
    root.createElementAsLastChild(MbXMLNSC.PARSER_NAME);
//Create the XML declaration parent node
MbElement xmlDecl =
    xmlRoot.createElementAsFirstChild(MbXMLNSC.XML_DECLARATION);

xmlDecl.setName("XmlDeclaration");

MbElement version =
    xmlDecl.CreateElementAsFirstChild(MbXMLNSC.ATTRIBUTE, "Version", "1.0");
MbElement encoding =

```

```
xmlDecl.CreateElementAsFirstChild(MbXMLNSC.ATTRIBUTE, "Encoding", "utf-8");
MbElement standalone =
xmlDecl.CreateElementAsFirstChild(MbXMLNSC.ATTRIBUTE, "Standalone", "Yes");
```

XMLNSC: The message body:

The XMLNSC parser builds a message tree from the body of an XML document.

The following topics describe how the XMLNSC parser builds a message tree from the body of an XML document:

- “XMLNSC: Using field types”
- “XMLNSC: Attributes and elements” on page 113
- “XMLNSC: Namespace declarations” on page 114
- “XMLNSC: Element values and mixed content” on page 116
- “XMLNSC: Comments and Processing Instructions” on page 118

XMLNSC: Using field types:

The XMLNSC parser sets the field type on every syntax element that it creates.

The field type indicates the type of XML construct that the element represents. The XMLNSC parser uses the field type when writing a message tree. The field type can be set by using ESQL or Java to control the output XML. The field types that are used by the XMLNSC parser must be referenced by using constants with names that are prefixed by 'XMLNSC.'

Tip: Field type constants that have the prefix 'XML.' are for use with the XMLNS and XML parsers only, and are not valid with the XMLNSC or MRM parsers.

Field types for creating syntax elements

Use the following field type constants to create syntax elements in the message tree. The XMLNSC parser uses these values when creating a message tree from an input message.

XML construct	XMLNSC Field Type constant	Value
Simple Element	XMLNSC.Field XMLNSC.CDataField	0x03000000 0x03000001
Attribute	XMLNSC.SingleAttribute XMLNSC.Attribute	0x03000101 0x03000100
Mixed content	XMLNSC.Value XMLNSC.CDataValue	0x02000000 0x02000001
Namespace Declaration	XMLNSC.SingleNamespaceDecl XMLNSC.NamespaceDecl	0x03000102 0x03000103
Complex element	XMLNSC.Folder	0x01000000
Inline DTD	XMLNSC.DocumentType	0x01000300
XML declaration	XMLNSC.XmlDeclaration	0x01000400
Entity reference	XMLNSC.EntityReference	0x02000100
Entity definition	XMLNSC.SingleEntityDefinition XMLNSC.EntityDefinition	0x03000301 0x03000300
Comment	XMLNSC.Comment	0x03000400
Processing Instruction	XMLNSC.ProcessingInstruction	0x03000401

Field types for path expressions (generic field types)

Use the field type constants that are listed below when querying the message tree by using a path expression; for example:

```
SET str = FIELDVALUE(InputRoot.e1.(XMLNSC.Attribute)attr1)
```

It is good practice to specify field types when querying a message tree built by the XMLNSC parser. This makes your ESQL code more specific and more readable, and it avoids incorrect results in some cases. However, care is required when choosing which field type constant to use. When you use the XMLNSC parser, use a generic field type constant when querying the message tree. This allows your path expression to tolerate variations in the input XML.

The generic field type constants are listed below:

XML construct	XMLNSC Field Type constant	Purpose
Tag	XMLNSC.Element	Matches any tag, whether it contains child tags (XMLNSC.Folder) or a value (XMLNSC.Field)
Element	XMLNSC.Field	Matches a tag which contains normal text, CDATA, or a mixture of both. Does not match tags which contain child tags.
Attribute	XMLNSC.Attribute	Matches single-quoted and double-quoted attributes
Mixed content	XMLNSC.Value	Matches normal text, CDATA, or a mixture of both
XML Declaration	XMLNSC.NamespaceDecl	Matches single- and double-quoted declarations

If you write

```
InputRoot.e1.(XMLNSC.DoubleAttribute)attrName
```

your path expression does not match a single-quoted attribute. If you use the generic field type constant *XMLNSC.Attribute*, your message flow works with either single-quoted or double-quoted attributes.

Note that you should always use the field type constants and not their numeric values.

Field types for controlling output format

The following field types are provided for XML Schema and base64 support. Do not use these field type constants in path expressions; use them in conjunction with *XMLNSC.Attribute* and *XMLNSC.Field* to indicate the required output format for DATE and BLOB values. See “XMLNSC: XML Schema support” on page 119 for further information.

XMLNSC Field Type constant	Purpose	Value
XMLNSC.gYear	The value must be a DATE. If the field type includes this value, the DATE value is produced by using the XML Schema gYear format.	0x00000010

XMLNSC Field Type constant	Purpose	Value
XMLNSC.gYearMonth	The value must be a DATE. If the field type includes this value, the DATE value is produced by using the XML Schema gYearMonth format.	0x00000040
XMLNSC.gMonth	The value must be a DATE. If the field type includes this value, the DATE value is produced by using the XML Schema gMonth format.	0x00000020
XMLNSC.gMonthDay	The value must be a DATE. If the field type includes this value, the DATE value is produced by using the XML Schema gMonthDay format.	0x00000050
XMLNSC.gDay	The value must be a DATE. If the field type includes this value, the DATE value is produced by using the XML Schema gDay format.	0x00000030
XMLNSC.base64Binary	The value must be a BLOB. The value is produced with base64 encoding.	0x00000060
XMLNSC.List	The element must be XMLNSC.Attribute or XMLNSC.Field. If the field type includes this value, the values of all child elements in the message tree are produced as a space-separated list.	0x00000070

Field types for direct output

Use the following field types to produce pre-constructed segments of an XML document. Character escaping is not done; therefore, take extra care not to construct a badly-formed output document. Use these constants only after carefully exploring alternative solutions.

XMLNSC Field Type constant	Purpose	Value
XMLNSC.Bitstream	The value of this syntax element must be a BLOB. The value is written directly to the output bit stream. For more information about its usage, see “Working with large XML messages” on page 105.	0x03000200
XMLNSC.AsisElementContent	The value of this syntax element must be CHARACTER. The value is written directly to the output bit stream. No character substitutions are performed. Use this element with care.	0x03000600

XMLNSC: Attributes and elements:

The XMLNSC parser uses field types to represent attributes and elements.

Use the field type constants that are listed below when creating your own syntax elements in the message tree.

Table 1. Specific field type constants

XML Construct	XMLNSC Field Type constant	Value
Complex element	XMLNSC.Folder	0x01000000
Simple element	XMLNSC.Field	0x02000000
	XMLNSC.CDataField	0x02000001
Attribute	XMLNSC.SingleAttribute	0x03000100
	XMLNSC.Attribute	0x03000101

When accessing elements and attributes in the message tree, use generic field type constants which match all of the alternative values. Because there is only one type of Folder element, it is safe to use XMLNSC.Folder when querying the message tree.

Table 2. Generic field type constants

XML Construct	XMLNSC Field Type constant	Purpose
Element	XMLNSC.Field	Matches elements that contain normal text, CDATA, or a mixture of both
Attribute	XMLNSC.Attribute	Matches both single-quoted and double-quoted attributes

ESQL code examples

The following examples use this XML message:

```
<root id="12345">
  <id>ABCDE</id>
</root>
```

Note that the message contains an attribute and an element with the same name.

Example 1 : Query the value of an XML element

```
SET value = FIELDVALUE(InputRoot.XMLNSC.root.(XMLNSC.Field)id)
```

The result is that value is set to 'ABCDE'.

Example 2 : Query the value of an XML attribute

```
SET value = FIELDVALUE(InputRoot.XMLNSC.root.(XMLNSC.Attribute)id)
```

The result is that value is set to '12345'.

Example 3 : Create the example message by using ESQL

```
CREATE LASTCHILD OF OutputRoot.XMLNSC Type 'XMLNSC.Folder' Name 'root';
-- Note : XMLNSC.Attribute could be used here as well.
SET OutputRoot.XMLNSC.root.(XMLNSC.Attribute)id = '12345';
SET OutputRoot.XMLNSC.root.(XMLNSC.Field)id = 'ABCDE';
```

The first line is optional because the element 'root' is created automatically by the following line if it does not already exist.

XMLNSC: Namespace declarations:

The XMLNSC parser provides full support for namespaces.

The XMLNSC parser sets the correct namespace on every syntax element that it creates while parsing a message, and stores namespace declarations in the message tree. The parser uses the namespace declarations to select the appropriate prefixes when outputting the message tree.

The XMLNSC parser uses the following field types to represent namespace declarations. Use the field type constants that are listed below when you create namespace declarations in the message tree.

Table 3. Specific field type constants

XML construct	XMLNSC field type constant	Value
Namespace declaration	XMLNSC.SingleNamespaceDecl	0x03000102
	XMLNSC.DoubleNamespaceDecl	0x03000103

When accessing elements and attributes in the message tree, do not use the constants that are listed in the previous table; instead, use the generic field type constant that matches both of the values in the table above.

Table 4. Generic field type constants

XML construct	XMLNSC field type constant	Purpose
Namespace declaration	XMLNSC.NamespaceDecl	Matches both single-quoted and double-quoted namespace declarations

ESQL code examples

Example 1 : Declaring a non-empty prefix

```
DECLARE space1 NAMESPACE 'namespace1';
SET OutputRoot.XMLNSC.space1:root.(XMLNSC.NamespaceDecl)xmlns:ns1 = space1;
SET OutputRoot.XMLNSC.space1:root.space1:example = 'ABCDE';
```

This creates the following XML message:

```
<ns1:root xmlns:ns1="namespace1">
  <ns1:example>ABCDE</ns1:example>
</ns1:root>
```

Note that the NAMESPACE constant space1 is just a local variable in the ESQL; it does not affect the namespace prefix ns1 that is defined by the NameSpaceDecl element and appears in the output message.

However, as shown here, space1 can be used to initialize the NameSpaceDecl for ns1. This avoids the need to duplicate the namespace URI ('namespace1' in this example), which in practice is typically a much longer string.

Example 2 : Declaring an empty prefix

```
DECLARE space1 NAMESPACE 'namespace1';
SET OutputRoot.XMLNSC.space1:root.(XMLNSC.NamespaceDecl)xmlns = space1;
SET OutputRoot.XMLNSC.space1:root.space1:example = 'ABCDE';
```

This creates the following XML message:

```
<root xmlns="namespace1">
  <example>ABCDE</example>
</root>
```

Note that the syntax elements root and example must have a non-empty namespace. The default namespace declaration means that any child element without a prefix is in the namespace namespace1.

Example 3 : Example of incorrect usage

```
DECLARE space1 NAMESPACE 'namespace1';
SET OutputRoot.XMLNSC.root.(XMLNSC.NamespaceDecl)xmlns = space1;
SET OutputRoot.XMLNSC.root.example = 'ABCDE';
```

This example causes the XMLNSC parser to issue the message BIP5014 when it attempts to create the message tree. The elements `root` and `example` are both within the scope of the default namespace declaration; therefore, in ESQL, these elements must be qualified by a namespace prefix bound to that namespace.

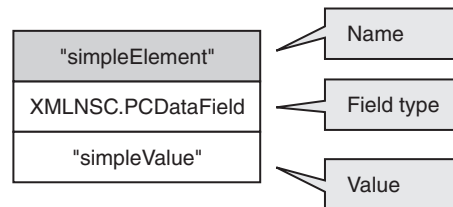
XMLNSC: Element values and mixed content:

The XMLNSC parser is a compact parser; therefore, an element with single content is parsed as a single syntax element. When an element has both child elements and some text, the text is called *mixed content*.

Element with simple content

The following XML fragment with a single content is parsed as a single syntax element:

```
<simpleElement>simpleValue</simpleElement>
```



The value of this element can be queried with this ESQL:

```
SET val = FIELDVALUE(InputRoot.XMLNSC.(XMLNSC.Field)simpleElement);
```

To generate an element with simple content in the output:

```
SET OutputRoot.XMLNSC.(PCDataField)simpleElement VALUE = 'simpleValue';
```

Note that `XMLNSC.Field` is used when querying the message tree, but `XMLNSC.PCDataField` is specified when constructing the output message. `XMLNSC.PCDataField` can be used to query the message tree; however, that would not work if the input message used a `CData` section, as shown below:

```
<simpleElement><![CDATA[simpleValue]]></simpleElement>
```

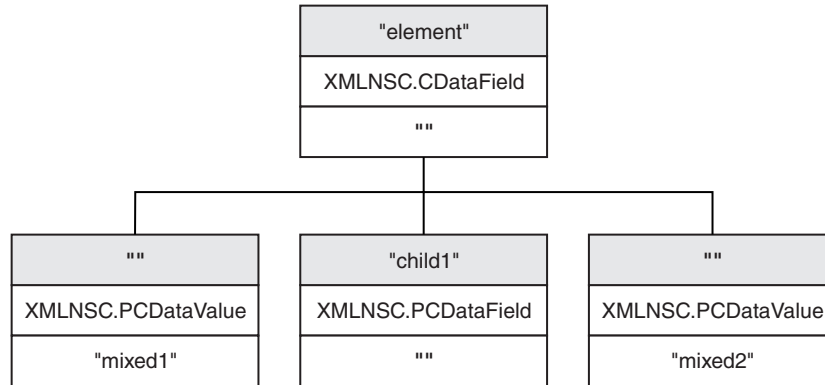
Element with mixed content

If an element has child elements, it is typically a 'folder', and does not have a value. When an element has both child elements and some text, the text is called 'mixed content'.

```
<element>mixed1<child/>mixed2</element>
```

By default, mixed content is discarded because it is typically just formatting white space and has no business meaning. Mixed content can be preserved if you select the Retain mixed content check box on the Parser Options page of the node properties.

If mixed content is being preserved, the XMLNSC parser creates a `Value` child element for each distinct item of mixed content.



The mixed content can be queried with this ESQL:

```
SET mixed1 = FIELDVALUE(InputRoot.XMLNSC.(element).*[1]);
```

The ESQL to construct the above XML fragment is:

```
CREATE ref REFERENCE TO OutputRoot.XMLNSC.element;
CREATE FIRSTCHILD OF ref TYPE XMLNSC.PCDataValue VALUE 'mixed1';
CREATE LASTCHILD OF ref NAME 'child1' TYPE XMLNSC.PCDataField VALUE '';
CREATE LASTSTCHILD OF ref TYPE XMLNSC.PCDataValue VALUE 'mixed2';
```

The following ESQL enables the *Retain mixed content* option:

```
DECLARE X BLOB;
-- assume that X contains an XML document
CREATE LASTCHILD OF OutputRoot
  PARSE(X OPTIONS XMLNSC.MixedContentRetainAll);
```

Element containing a CData section

A CData section is an XML notation that allows XML markup characters to be included in the content of an element.

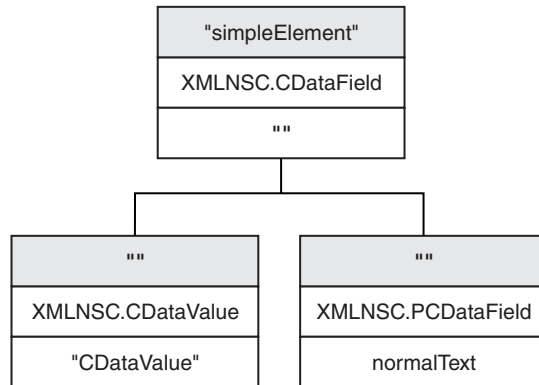
The following two XML fragments are identical in their meaning:

```
<simpleElement>simpleValue</simpleElement>
<simpleElement><![CDATA[simpleValue]]></simpleElement>
```

If the CData section is the only text content, the XMLNSC parser remembers that the input document contained a CData section by setting the field type to XMLNSC.CDataField instead of XMLNSC.PCDataField.

If the CData section is not the only text content, it is created as a child value element, with other child value elements representing the remaining text content. An example is shown below:

```
<simpleElement><![CDATA[CDataValue]]>normalText</simpleElement>
```



See “XMLNSC: Working with CData” on page 103 for more information about the correct use of CData in XML documents.

XMLNSC: Comments and Processing Instructions:

The XMLNSC parser discards comments and processing instructions because both comments and processing instructions are auxiliary information with no business meaning.

Comments

Comments can be preserved if you select the *Retain comments* check box on the Parser Options page of the node properties.

If *Retain comments* is selected, each comment in the input document is represented by a single syntax element with field type XMLNSC.Comment. The *Retain comments* option can also be accessed by using the following ESQL:

```

DECLARE X BLOB;
-- assume that X contains an XML document
CREATE LASTCHILD OF OutputRoot.XMLNSC
  PARSE(X DOMAIN XMLNSC
    NAME preserveComments
    OPTIONS XMLNSC.CommentsRetainAll);

-- do it again, this time discarding comments
CREATE LASTCHILD OF OutputRoot.XMLNSC
  PARSE(X DOMAIN XMLNSC
    NAME discardComments
    OPTIONS XMLNSC.CommentsRetainNone);
  
```

Processing Instructions

Processing instructions can be preserved if you select the *Retain processing instructions* check box on the Parser Options page of the node properties.

If *Retain processing instructions* is selected, each processing instruction the input document is represented by a single syntax element with field type XMLNSC.ProcessingInstruction. The *Retain processing instructions* option can also be accessed by using the following ESQL:

```

DECLARE X BLOB;
-- assume that X contains an XML document
CREATE LASTCHILD OF OutputRoot.XMLNSC
  PARSE(X DOMAIN XMLNSC
  
```

```

NAME preserveProcessingInstructions
OPTIONS XMLNSC.ProcessingInstructionsRetainAll);

-- do it again, this time discarding processing instructions
CREATE LASTCHILD OF OutputRoot.XMLNSC
  PARSE(X DOMAIN XMLNSC
    NAME discardProcessingInstructions
    OPTIONS XMLNSC.ProcessingInstructionsRetainNone);

```

XMLNSC: XML Schema support:

Use the XMLNSC parser to parse and validate by using an XML Schema.

For information about how to configure the XMLNSC parser to use an XML Schema, see XMLNSC parser.

The following topics describe how the XMLNSC parser uses the field type to hold information about XML Schema simple types. This behavior enables the parser to write dates and binary elements in the same form in which they were parsed, and according to the XML Schema specification. It also allows the message flow developer to write dates, lists, and binary data in the correct XML Schema format.

- “XMLNSC: base64 support”
- “XMLNSC: XML Schema date formatting”
- “XMLNSC: XML List type support” on page 120

XMLNSC: base64 support:

The XMLNSC parser can produce binary data in base64-encoded format.

If Validation is set to Content and Value, and Build tree using schema types is enabled, the XMLNSC parser automatically decodes base64 data and creates a BLOB value in the message tree. When producing a message tree, the XMLNSC parser will 'base64-encode' a BLOB if the field type includes the constant *XMLNSC.base64Binary*.

ESQL code example to output base64 data

```

DECLARE Base64Data BLOB '0102030405060708090A0B0C0D0E0F';
-- Add in the base64Binary field type
DECLARE base64FieldType INTEGER XMLNSC.Field + XMLNSC.base64Binary;
CREATE LASTCHILD OF OutputRoot DOMAIN 'XMLNSC' NAME 'XMLNSC';
CREATE LASTCHILD OF OutputRoot.XMLNSC TYPE base64FieldType NAME 'myBinaryData' VALUE Base64Data;

```

Result : <myBinaryData>AQIDBAUGBwgJCgsMDQ4P</myBinaryData>

Note that this example does not depend on validation. The XMLNSC parser can produce base64 binary data even if Validation is set to None.

XMLNSC: XML Schema date formatting:

The XMLNSC parser can parse and write all of the XML Schema simple types.

The rarely used types gYear, gYearMonth, gMonth, gMonthDay, and gDay do not map directly to a message broker data type. For these simple types, the XMLNSC parser adds one of the following constant values to the field type. This behavior allows the parser to produce the data for output in the same format as it was received.

Field types for controlling date format

The following field types are provided for XML Schema date format support. Do not use these field type constants in path expressions. Use them in conjunction with constants XMLNSC.Attribute and XMLNSC.Field to indicate the required output format for DATE values.

XMLNSC Field Type constant	Purpose	Value
XMLNSC.gYear	Value must be a DATE. If the field type includes this value, the DATE value is written by using the XML Schema gYear format.	0x00000010
XMLNSC.gYearMonth	Value must be a DATE. If the field type includes this value, the DATE value is written by using the XML Schema gYearMonth format.	0x00000040
XMLNSC.gMonth	Value must be a DATE. If the field type includes this value, the DATE value is written by using the XML Schema gMonth format.	0x00000020
XMLNSC.gMonthDay	Value must be a DATE. If the field type includes this value, the DATE value is written by using the XML Schema gMonthDay format.	0x00000050
XMLNSC.gDay	Value must be a DATE. If the field type includes this value, the DATE value is written by using the XML Schema gDay format.	0x00000030

ESQL code example

```
DECLARE gYear DATE '2007-01-01';
-- Add in the gYear field type
DECLARE gYearFieldType INTEGER XMLNSC.Field + XMLNSC.gYear;
CREATE LASTCHILD OF OutputRoot DOMAIN 'XMLNSC' NAME 'XMLNSC';
CREATE LASTCHILD OF OutputRoot.XMLNSC TYPE gYearFieldType NAME 'gYear' VALUE gYear;
```

Result : <gYear>2007</gYear>

XMLNSC: XML List type support:

The XMLNSC parser can automatically parse a space-separated list of values into individual syntax elements in the message tree if you select certain options.

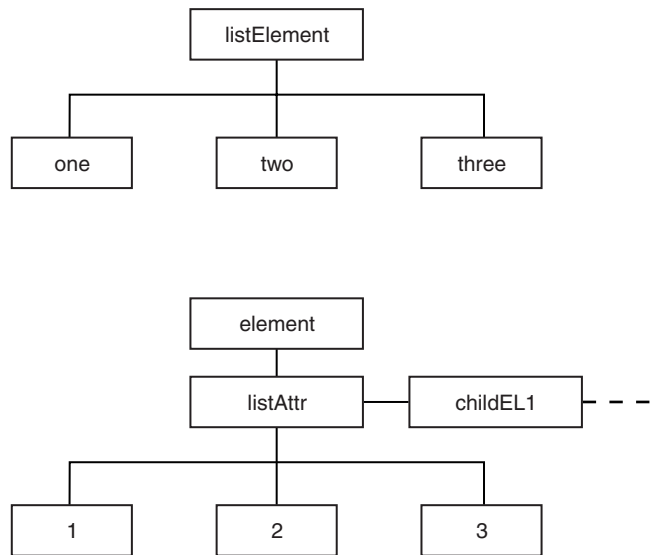
An element or an attribute can have multiple values separated by spaces, as shown in the examples below:

```
<listElement>one two three</listElement>
<element listAttribute="1 2 3"><childEL1/></element>
```

If your XML schema specifies a list type for an element or an attribute, and Validation is set to Content and Value, and Build tree using schema types is

enabled, the XMLNSC parser automatically parses the space-separated list of values into individual syntax elements in the message tree. The resulting message tree looks like this:

and for an attribute with a list value it looks like this:



ESQL code examples

Access the individual values in a list

```
SET val = InputRoot.XMLNSC.listElement.*[1];
```

Result : val = 'one'

```
SET val = InputRoot.XMLNSC.element.(XMLNSC.Attribute)listAttr.*[3];
```

Result : val='3'

Create a list element in the message tree

```
CREATE LASTCHILD OF OutputRoot.XMLNSC
  Name 'listElement'
  Type XMLNSC.List;
DECLARE listEl REFERENCE TO OutputRoot.XMLNSC.listElement;
DECLARE listValType INTEGER XMLNSC.PCDataValue;
CREATE LASTCHILD OF listEl TYPE listValType VALUE 'one';
CREATE LASTCHILD OF listEl TYPE listValType VALUE 'two';
CREATE LASTCHILD OF listEl TYPE listValType VALUE 'three';
```

Migrating to XMLNSC:

The XMLNSC parser now offers the best combination of features and performance for most applications.

Reasons to migrate

If your message flow uses the XMLNS or XML domain, you might want to migrate a message flow to XMLNSC to take advantage of the XML schema validation. If your message flow uses the MRM domain, you might want to migrate to XMLNSC to obtain standards-compliant validation, and a large reduction in CPU usage.

Migrating from the XMLNS or XML domain

The XMLNSC parser differs from the XMLNS parser in the following ways:

- The XMLNSC parser builds a compact message tree.
- It uses different field type constants.
- It discards inline DTDs

In most cases, the compact message tree has no effect on ESQL paths or XPath expressions. Typically, a simple message tree query produces the same results in XMLNSC as in the XMLNS or XML domain. Merely changing the correlation name from XMLNS to XMLNSC is usually sufficient, but care must be taken with complex XPath expressions that navigate to the value of an element, then to its parent in a single query. These might produce different results in the XMLNSC domain.

The field type constants that are used by the XMLNSC parser are completely different from those used by XMLNS or XML. Every occurrence of XML.Attribute, XML.XmlDecl, for example, must be changed to use the equivalent XMLNSC field type constant.

The discarding of inline DTDs only affects message flows that process the DTD.

Migrating from MRM XML

The XMLNSC parser differs from the MRM XML parser in the following ways:

- The XMLNSC parser uses field types to identify the XML constructs in the message tree. The MRM parser distinguishes attributes from elements by matching the message tree against the message definition.
- When writing a message tree, the XMLNSC parser selects namespace prefixes by detecting and using xmlns attributes in the message tree. The MRM XML parser uses a table in the message set properties.
- The MRM parser does not include the root tag of the XML document in the message tree.

Migrating a message flow from MRM to XMLNSC typically requires extensive changes to your message flow. However, the migration usually delivers a large reduction in CPU usage, and allows much more accurate control of the output XML.

Manipulating messages in the XMLNS domain

This topic provides information about how to write ESQL for processing messages that belong to the XMLNS domain, and that are parsed by the XMLNS parser. Most of the information in this topic, and the topics mentioned below, also applies to the XML domain, unless it refers to features that are not supported in the XML domain.

Refer to XMLNS parser for background information.

The following topics provided detailed information about the structure of the message tree that the XMLNS parser builds, and the field types that it uses.

- “XMLNS: The XML declaration” on page 123
- “XMLNS: The DTD” on page 124
- “XMLNS: The XML message body” on page 125

More information about processing XML messages can be found in “Working with XML messages” on page 99.

XMLNS: The XML declaration:

The beginning of an XML message can contain an XML declaration.

An example of a declaration is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

The XML Declaration is represented by the following types of syntax element:

- “XML.XMLDecl”
- “XML.version”
- “XML.encoding”
- “XML.standalone” on page 124

“XMLNS: XML declaration example” on page 124 includes another example of an XML declaration and the tree structure that it forms.

XML.XMLDecl:

The XML Declaration is represented in the message tree by a syntax element with field type 'XML.XMLDecl'.

If the XML declaration is created by the XMLNS parser its name is 'XMLDecl'. However, when a message tree is being written, the name is not important; only the field type is used by the parser.

An example of a declaration is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

XML.version:

The XML version attribute in the XML declaration is represented in the message tree by a syntax element with field type 'XML.version'.

The value of the XML version attribute is the value of the version attribute. It is always a child of an XML.XmlDecl syntax element. In the example below, the version element contains the string value 1.0:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

XML.encoding:

The encoding attribute is represented by a syntax element with field type 'XML.encoding', and is always a child of an XML.XmlDecl syntax element.

In the example shown below, the encoding attribute has a value of UTF-8.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">
<s1>.....</s1>
```

You cannot specify WebSphere MQ encodings in this element.

In your ESQL, (XML,"Encoding") must include quotation marks, because Encoding is a reserved word.

XML.standalone:

The XML standalone element defines the existence of an externally-defined DTD. In the message tree it is represented by a syntax element with field type XML.standalone.

The value of the XML standalone element is the value of the standalone attribute in the XML declaration. It is always a child of an XML.XmlDecl syntax element. The only valid values for the standalone element are yes and no. An example is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<!DOCTYPE s1 PUBLIC "http://www.ibm.com/example.dtd" "example.dtd">  
<s1>.....</s1>
```

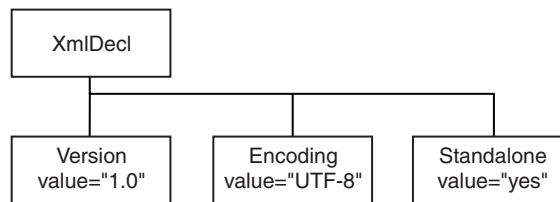
A value of no indicates that this XML document is not standalone, and depends on an externally-defined DTD. A value of yes indicates that the XML document is self-contained. However, because the current release of WebSphere Message Broker does not resolve externally-defined DTDs, the setting of standalone is irrelevant, and is ignored.

XMLNS: XML declaration example:

The following example shows an XML declaration in an XML document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

The following figure shows the tree structure that the XMLNS parser creates for this declaration:



XMLNS: The DTD:

The document type declaration (DTD) of an XML message is represented by a syntax element with field type XML.DocTypeDecl, and its children. These comprise the DOCTYPE construct.

Only internal (inline) DTD subsets are represented in the syntax element tree. An inline DTD is a DTD that is declared within the XML document itself. It can be a complete DTD definition, or it can extend the definition in an external DTD.

External DTD subsets (identified by the SystemID or PublicId elements described below) can be referenced in the message, but those referenced are not resolved by the broker.

The following field type constants can be used to reference the various parts of a DTD in the message tree:

- XML DocTypeDecl
- XML NotationDecl
- XML entities
- XML ElementDef
- XML AttributeList
- XML AttributeDef
- XML DocTypePI
- XML WhiteSpace and DocTypeWhiteSpace
- XML DocTypeComment

XML DTD example shows an example of an XML DTD.

See XML document type declaration for more information about handling an inline DTD.

XMLNS: The XML message body:

Every XML message must have a body. The body consists of a hierarchy of XML elements and other XML constructs that represent the message data.

The XMLNS parser assigns a field type to every syntax element that it creates. The value of the field type indicates the XML construct that it represents. In the following topics, each field type is discussed, with a series of example XML fragments.

The following common element types are discussed:

- “XML.element” on page 127
- “XML.Attribute” on page 128
- “XML.content” on page 128

“XML message body example” on page 132 provides an example of an XML message body and the tree structure that is created from it using the syntax elements types that are listed above.

More complex XML messages might use some of the following syntax element types:

- “XML.CDataSection” on page 128
- “XML.EntityReferenceStart and XML.EntityReferenceEnd” on page 129
- “XML.comment” on page 130
- “XML.ProcessingInstruction” on page 130
- “XML.AsisElementContent” on page 130
- “XML.BitStream” on page 131

Accessing attributes and elements:

The XMLNS parser sets the field type on every message tree element that it creates.

The field type indicates the type of XML construct that the element represents. The field types used by the XMLNS parser can be referenced using constants with

names prefixed with 'XML.' Field type constants with prefix 'XML.' are for use with the XMLNS and XML parsers only; they do not work with the XMLNSC or MRM parsers.

	XMLNS Field Type constant
Tag	XML.Element
Attribute	XML.Attribute XML.Attr

By using the field type in this way, the XMLNS parser can distinguish between an element and an attribute that have the same name.

Example XML

```
<parent id="12345">
  <id>ABCDE</id>
</parent>
```

Example ESQL

```
SET value = FIELDVALUE(InputRoot.XMLNS.parent.(XML.Element)id)
Result : value is 'ABCDE'
```

```
SET value = FIELDVALUE(InputRoot.XMLNS.parent.(XML.Attr)id)
Result : value is '12345'
```

Example using SELECT to access multiple attributes

In the example Invoice message, the element Title within each Item element has three attributes: Category, Form, and Edition. For example, the first Title element contains:

```
<Title Category="Computer" Form="Paperback" Edition="2">The XML Companion</Title>
```

The element InputRoot.XML.Invoice.Purchases.Item[1].Title has four children in the logical tree: Category, Form, Edition, and the element value, which is "The XML Companion".

If you want to access the attributes for this element, you can code the following ESQL. This extract of code retrieves the attributes from the input message and creates them as elements in the output message. It does not process the value of the element itself in this example.

```
-- Set the cursor to the first XML.Attribute of the Title.
-- Note the * after (XML.Attribute) meaning any name, because the name might not be known
DECLARE cursor REFERENCE TO InputRoot.XMLNS.Invoice.Purchases.Item[1].Title.(XML.Attribute)*;
WHILE LASTMOVE(cursor) DO
  -- Create a field with the same name as the XML.Attribute
  -- and set its value to the value of the XML.Attribute
  SET OutputRoot.XML.Data.Attributes.{FIELDNAME(cursor)} = FIELDVALUE(cursor);
  -- Move to the next sibling of the same TYPE to avoid the Title value
  -- which is not an XML.Attribute
  MOVE cursor NEXTSIBLING REPEAT TYPE;
END WHILE;
```

When this ESQL is processed by the Compute node, the following output message is generated:

```
<Data>
  <Attributes>
    <Category>Computer</Category>
```

```

    <Form>Paperback</Form>
    <Edition>2</Edition>
  </Attributes>
</Data>

```

You can also use a SELECT statement:

```

SET OutputRoot.XMLNS.Data.Attributes[] =
  (SELECT FIELDVALUE(I.Title) AS title,
        FIELDVALUE(I.Title.(XML.Attribute)Category) AS category,
        FIELDVALUE(I.Title.(XML.Attribute)Form) AS form,
        FIELDVALUE(I.Title.(XML.Attribute)Edition) AS edition
   FROM InputRoot.XML.Invoice.Purchases.Item[] AS I);

```

This statement generates the following output message:

```

<Data>
  <Attributes>
    <title>The XML Companion</title>
    <category>Computer</category>
    <form>Paperback</form>
    <edition>2</edition>
  </Attributes>
  <Attributes>
    <title>A Complete Guide to DB2 Universal Database</title>
    <category>Computer</category>
    <form>Paperback</form>
    <edition>2</edition>
  </Attributes>
  <Attributes>
    <title>JAVA 2 Developers Handbook</title>
    <category>Computer</category>
    <form>Hardcover</form>
    <edition>0</edition>
  </Attributes>
</Data>

```

You can qualify the SELECT with a WHERE statement to narrow down the results to obtain the same output message as the one that is generated by the WHILE statement. This second example shows that you can create the same results with less, and less complex, ESQL.

```

SET OutputRoot.XMLNS.Data.Attributes[] =
  (SELECT FIELDVALUE(I.Title.(XML.Attribute)Category) AS category,
        FIELDVALUE(I.Title.(XML.Attribute)Form) AS form,
        FIELDVALUE(I.Title.(XML.Attribute)Edition) AS edition
   FROM InputRoot.XML.Invoice.Purchases.Item[] AS I
   WHERE I.Title = 'The XML Companion');

```

This statement generates the following output message:

```

<Data>
  <Attributes>
    <Category>Computer</Category>
    <Form>Paperback</Form>
    <Edition>2</Edition>
  </Attributes>
</Data>

```

XML.element: This syntax element represents an XML element (a tag). The name of the syntax element corresponds to the name of the XML element in the message. This element can have many children in the message tree, including attributes, elements, and content.

XML.tag is supported as an alternative to XML.element for compatibility with earlier versions of WebSphere Message Broker. Use XML.element in any new message flows that you create.

XML.Attribute: **Parsing**

The XMLNS parser uses this field type for syntax elements that represent an XML attribute. The name and value of the syntax element correspond to the name and value of the XML attribute that is represented. Attribute elements have no children, and must always be children of an element.

Writing

When the XMLNS parser generates a bit stream from a message tree, occurrences of ampersand (&), less than (<), greater than (>), double quotation mark ("), and apostrophe ('), within the attribute value, are replaced by the predefined XML entities &, <, >, ", and '.

The XML.attr field type constant is also supported for compatibility with earlier versions of WebSphere Message Broker.

XML.content: The XMLNS parser uses this syntax element to represent character data (including an XML attribute). The name and value of the syntax element correspond to the name and value of the XML attribute that is represented. Attribute elements have no children, and must always be children of an element.

Writing

When the XMLNS parser generates a bit stream from a message tree, occurrences of ampersand (&), less than (<), greater than (>), double quotation mark ("), and apostrophe ('), within the attribute value, are replaced by the predefined XML entities &, <, >, ", and '.

XML.CDataSection:

CData sections in the XML message are represented by a syntax element with field type XML.CdataSection.

The content of the CDataSection element is the value of the CDataSection element without the <![CDATA[that marks its beginning, and the]]> that marks its end.

For example, the following CData section:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

is represented by a CDataSection element with a string value of:

```
"<greeting>Hello, world!</greeting>"
```

Unlike Content, occurrences of <, >, &, " and ' are not translated to their XML character entities (<, >, and &) when the CDataSection is produced.

When to use XML.CDataSection

A CData section is often used to embed one XML message within another. By using a CData section, you ensure that the XML reserved characters (<, >, and &) are not replaced with XML character entities.

XML.AsisElementContent also allows the production of unmodified character data, but XML.CDataSection is typically a better choice because it protects the outer message from errors in the embedded message.

Parsing the contents of a CDataSection

A common requirement is to parse the contents of a CData section to create a message tree, which you can achieve by using the ESQL statement CREATE with the PARSE clause; see "XMLNSC: Working with XML messages and bit streams" on page 104.

XML.NamespaceDecl:

A namespace declaration is represented by a syntax element with field type XML.NamespaceDecl.

Namespace declarations take one of two forms in the message tree:

- **Declaration using a namespace prefix**

```
<ns1:e1 xmlns:ns1="namespace1"/>
```

In the message tree, the syntax element for this namespace declaration is shown in the following table:

Namespace	http://www.w3.org/2000/xmlns/
Name	ns1
Value	namespace1

- **Declaration of a default namespace**

A default namespace declaration is an xmlns attribute that defines an empty prefix

```
<e1 xmlns="namespace1"/>
```

In the message tree, the syntax element for this namespace declaration is shown in the following table:

Namespace	""
Name	xmlns
Value	namespace1

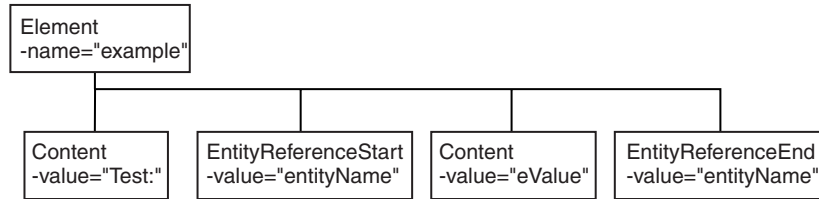
Note that, in both cases, element 'e1' is in namespace 'namespace1'.

XML.EntityReferenceStart and XML.EntityReferenceEnd:

When an entity reference is encountered in the XML message, both the expanded form and the original entity name are stored in the syntax element tree. The name of the entity is stored as the value of the EntityReferenceStart and EntityReferenceEnd syntax elements, and any syntax elements between contain the entity expansion.

The following examples show the XML entity references in an XML document, and in tree structure form.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE example [ <!ENTITY entityName "eValue"> ]>
<example>Test: &entityName;</example>
```



The XML declaration and the document type declaration are not shown here. Refer to “XMLNS: The XML declaration” on page 123 and “XMLNS: The DTD” on page 124 for details of those sections of the syntax element tree.

XML.comment:

An XML.comment that is encountered outside the document type declaration is represented by a syntax element with field type XML.comment. The value of the element is the comment text from the XML message.

If the value of the element contains the character sequence -->, the sequence is replaced with the text -->. This ensures that the contents of the comment cannot prematurely terminate the comment. Occurrences of the following characters are not translated to their escape sequences:

< > & " ' '

An example of the XML comment in an XML document is shown below:

```
<example><!-- This is a comment --></example>
```

XML.ProcessingInstruction:

A processing instruction that is encountered outside the document type declaration is represented by a syntax element with field type XML.ProcessingInstruction.

This is a name-value element; the name of the syntax element is the processing instruction target name, and the value of the syntax element is the character data of the processing instruction. The value of the syntax element must not be empty. The name cannot be XML in either uppercase or lowercase.

If the value of the element contains the character sequence ?>, the sequence is replaced with the text ?>. This ensures that the content of the processing instruction cannot prematurely terminate the processing instruction. Occurrences of the following characters are not translated to their escape sequences:

< > & " ' '

An example of the XML processing instruction in an XML document is shown below:

```
<example><?target This is a PI.?></example>
```

XML.AsisElementContent:

Use the special field type XML.AsisElementContent to precisely control generated XML.

XML.AsisElementContent is a special field type. Use the field type in a message flow to precisely control the XML that is generated in an output message, without

the safeguards of the Element, Attribute, and Content syntax elements. The XMLNS parser never creates elements with this field type.

Try to avoid using `AsisElementContent`; there is typically a safer alternative approach. If you do use `AsisElementContent`, it is your responsibility to ensure that the output message is well-formed XML.

You might choose to use `AsisElementContent` if, for example, you want to suppress the usual behavior in which occurrences of ampersand (&), less than (<), greater than (>), quotation mark ("), and apostrophe (') are replaced by the predefined XML entities `&`, `<`, `>`, `"`, and `'`.

The following example illustrates the use of `AsisElementContent`. The statement:

```
Set OutputRoot.XMLNS.(XML.Element)Message.(XML.Content) = '<rawMarkup>';
```

generates the following XML in an output message:

```
<Message>&lt;rawMarkup&gt;</Message>
```

However, the statement:

```
Set OutputRoot.XMLNS.(XML.Element)Message.(XML.AsisElementContent) = '<rawMarkup>';
```

generates the following XML in an output message:

```
<Message><rawMarkup></Message>
```

This shows that the value of an `AsisElementContent` syntax element is not modified before it is written to the output message.

XML.BitStream:

`BitStream` is a specialized element designed to aid the processing of very large messages.

`XML.Bitstream` is a special field type. When writing an XML message, the value of the `BitStream` element is written directly into the message, and the name is not important. The `BitStream` element might be the only element in the message tree.

The value of the element must be of type `BLOB`; any other data type generates an error when writing the element. Ensure that the content of the element is appropriate for use in the output message; pay special attention to the `CCSID` and the encoding of the XML text in the `BLOB`.

Use of the `BitStream` element is similar to the use of the `AsisElementContent` element, except that the `AsisElementContent` type converts its value into a string, whereas the `BitStream` element uses its `BLOB` value directly. This is a specialized element designed to aid the processing of very large messages.

The following ESQL excerpts demonstrate some typical use of this element. First, declare the element:

```
DECLARE StatementBitStream BLOB
```

Initialize the contents of `StatementBitStream` from an appropriate source, such as an input message. If the source field is not of type `BLOB`, use the `CAST` statement to convert the contents to `BLOB`. Then create the new field in the output message; for example:

```

CREATE LASTCHILD OF resultCursor
  Type XML.BitStream
  NAME 'StatementBitStream'
  VALUE StatementBitstream;

```

XML message body example:

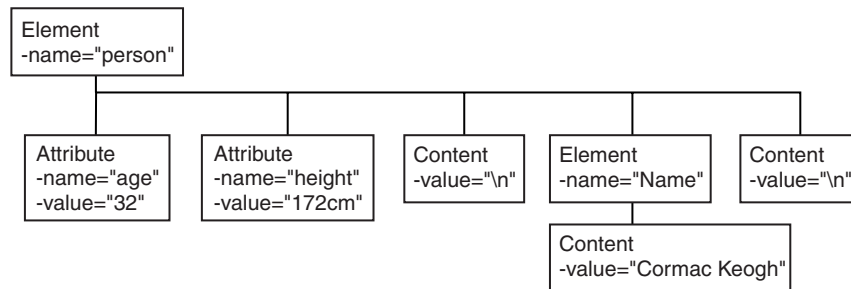
The XMLNS parser creates a message tree that represents an XML document.

The following example shows the message tree that the XMLNS parser creates for the following snippet from a simple XML document:

```

<Person age="32" height="172cm">
  <Name>Cormac Keogh</Name>
</Person>

```



Manipulating messages in the XML domain

The XML parser is very similar to the XMLNS parser, but it has no support for namespaces or opaque parsing. For information about how to work with the XML parser, refer to “Manipulating messages in the XMLNS domain” on page 122.

Manipulating messages in the MRM domain

How to use messages that have been modeled in the MRM domain, and that are parsed by the MRM parser.

The following topics show you how to deal with messages that have been modeled in the MRM domain, and that are parsed by the MRM parser. The physical formats associated with the message models do not affect this information unless specifically stated. Use this information in conjunction with the information about manipulating message body content; see “Manipulating message body content” on page 28.

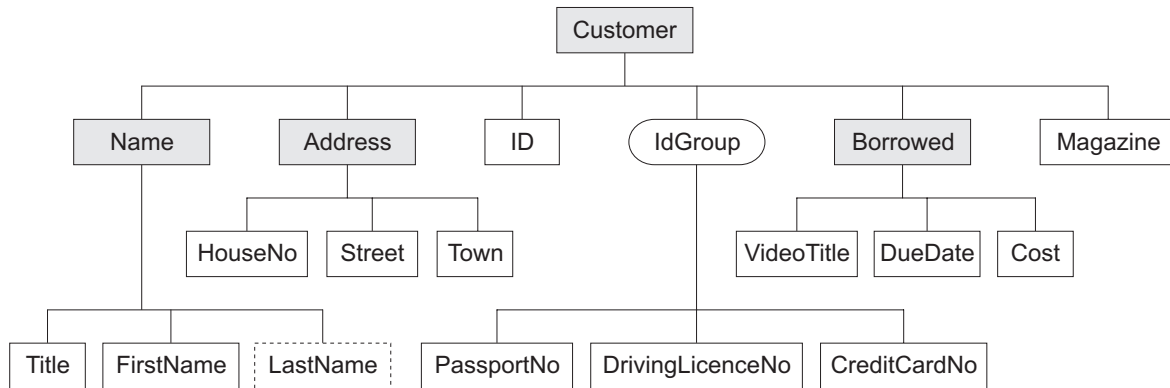
- “Accessing elements in a message in the MRM domain” on page 133
- “Accessing multiple occurrences of an element in a message in the MRM domain” on page 134
- “Accessing attributes in a message in the MRM domain” on page 135
- “Accessing elements within groups in a message in the MRM domain” on page 136
- “Accessing mixed content in a message in the MRM domain” on page 138
- “Accessing embedded messages in the MRM domain” on page 139
- “Accessing the content of a message in the MRM domain with namespace support enabled” on page 140
- “Querying null values in a message in the MRM domain” on page 141
- “Setting null values in a message in the MRM domain” on page 142
- “Working with MRM messages and bit streams” on page 142
- “Handling large MRM messages” on page 145

The structure of the "Customer" message is shown in the following sample:

- Video Rental

The message is used in the samples in the topics listed previously to show ESQL that manipulates the objects that can be defined in a message model.

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.



The message includes a variety of structures that demonstrate how you can classify metadata to the MRM. Within an MRM message set, you can define the following objects: messages, types, groups, elements, and attributes. Folder icons that represent each of these types of objects are displayed for each message definition file in the Broker Application Development perspective.

Each message definition file can contribute to a namespace; in this sample, each namespace is completely defined by a single message definition file. You can combine several message definition files to form a complete message dictionary, which you can then deploy to a broker.

The video sample has three message definition files:

Customer.mxsd

Resides in the no target namespace

Address.mxsd

Resides in the namespace <http://www.ibm.com/AddressDetails>

Borrowed.mxsd

Resides in the namespace <http://www.ibm.com/BorrowedDetails>

Look at the video rental message structure sample for detailed information about the objects that are defined in this message model:

- Video Rental

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Accessing elements in a message in the MRM domain

You can use ESQL to manipulate the logical tree that represents a message in the message flow. This topic describes how to access data for elements in a message in the MRM domain.

You can populate an element with data with the SET statement:

```
SET OutputRoot.MRM.Name = UPPER(InputRoot.MRM.Name);
```

The field reference on the left hand side of the expression refers to the element called Name within the MRM message domain. This statement takes the input value for the Name field, converts it to uppercase, and assigns the result to the same element in the output message.

The Name element is defined in the noTarget namespace. No namespace prefix is specified in front of the Name part of the field reference in the example above. If you have defined an MRM element in a namespace other than the noTarget namespace, you must also specify a namespace prefix in the statement. For example:

```
DECLARE brw NAMESPACE 'http://www.ibm.com/Borrowed';  
SET OutputRoot.MRM.brw:Borrowed.VideoTitle = 'MRM Greatest Hits';
```

For more information about using namespaces with messages in the MRM domain, see “Accessing the content of a message in the MRM domain with namespace support enabled” on page 140.

Accessing multiple occurrences of an element in a message in the MRM domain

You can use specific ESQL code to set the value of one occurrence of an element that has multiple occurrences in a message. You can also use arrow notation to indicate the direction of search when searching for multiple occurrences of an element.

You can access MRM domain elements following the general guidance given in “Accessing known multiple occurrences of an element” on page 33 and “Accessing unknown multiple occurrences of an element” on page 34. Further information specific to MRM domain messages is provided in this topic.

Consider the following statements:

```
DECLARE brw NAMESPACE 'http://www.ibm.com/Borrowed';  
  
SET OutputRoot.MRM.brw:Borrowed[1].VideoTitle = 'MRM Greatest Hits Volume 1';  
SET OutputRoot.MRM.brw:Borrowed[2].VideoTitle = 'MRM Greatest Hits Volume 2';
```

The above SET statements operate on two occurrences of the element Borrowed. Each statement sets the value of the child VideoTitle. The array index indicates which occurrence of the repeating element you are interested in.

When you define child elements of a complex type (which has its Composition property set to Sequence) in a message set, you can add the same element to the complex type more than once. These instances do not have to be contiguous, but you must use the same method (array notation) to refer to them in ESQL.

For example, if you create a complex type with a Composition of Sequence that contains the following elements:

- StringElement1
- IntegerElement1
- StringElement1

use the following ESQL to set the value of StringElement1:

```

SET OutputRoot.MRM.StringElement1[1] =
    'This is the first occurrence of StringElement1';
SET OutputRoot.MRM.StringElement1[2] =
    'This is the second occurrence of StringElement1';

```

You can also use the arrow notation (the greater than (>) and less than (<) symbols) to indicate the direction of search and the index to be specified:

```

SET OutputRoot.MRM.StringElement1[>] =
    'This is the first occurrence of StringElement1';
SET OutputRoot.MRM.StringElement1[<2] =
    'This is the last but one occurrence of
StringElement1';
SET OutputRoot.MRM.StringElement1[<1] =
    'This is the last occurrence of StringElement1';

```

Refer to “Accessing known multiple occurrences of an element” on page 33 and “Accessing unknown multiple occurrences of an element” on page 34 for additional detail.

Accessing attributes in a message in the MRM domain

When an MRM message is parsed into a logical tree, attributes and the data that they contain are created as name-value pairs in the same way that MRM elements are. The ESQL that you code to interrogate and update the data held in attributes refers to the attributes in a similar manner.

Consider the Video Rental sample MRM message. The attribute LastName is defined as a child of the Name element in the Customer message. Here is an example input XML message:

```

<Customer xmlns:addr="http://www.ibm.com/AddressDetails"
xmlns:brw="http://www.ibm.com/BorrowedDetails">
  <Name LastName="Bloggs">
    <Title>Mr</Title>
    <FirstName>Fred</FirstName>
  </Name>
  <addr:Address>
    <HouseNo>13</HouseNo>
    <Street>Oak Street</Street>
    <Town>Southampton</Town>
  </addr:Address>
  <ID>P</ID>
  <PassportNo>J123456TT</PassportNo>
  <brw:Borrowed>
    <VideoTitle>Fast Cars</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.50</Cost>
  </brw:Borrowed>
  <brw:Borrowed>
    <VideoTitle>Cut To The Chase</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.00</Cost>
  </brw:Borrowed>
  <Magazine>0</Magazine>
</Customer>

```

When the input message is parsed, values are stored in the logical tree as shown in the following section of user trace:

```

(0x0100001B):MRM = (
  (0x01000013):Name = (
    (0x0300000B):LastName = 'Bloggs'
    (0x0300000B):Title = 'Mr'
    (0x0300000B):FirstName = 'Fred'
  )
  (0x01000013)http://www.ibm.com/AddressDetails:Address = (
    (0x0300000B):HouseNo = 13
    (0x0300000B):Street = 'Oak Street'
    (0x0300000B):Town = 'Southampton'
  )
  (0x0300000B):ID = 'P'
  (0x0300000B):PassportNo = 'J123456TT'
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Fast Cars'
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.50
  )
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Cut To The Chase '
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.00
  )
  (0x0300000B):Magazine = FALSE

```

The following ESQL changes the value of the LastName attribute in the output message:

```
SET OutputRoot.MRM.Name.LastName = 'Smith';
```

Be aware of the ordering of attributes when you code ESQL. When attributes are parsed, the logical tree inserts the corresponding name-value before the MRM element's child elements. In the previous example, the child elements Title and FirstName appear in the logical message tree after the attribute LastName. In the Broker Application Development perspective, the Outline view displays attributes after the elements. When you code ESQL to construct output messages, you must define name-value pairs for attributes before any child elements.

The following sample shows the structure of the Customer message:

- Video Rental

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Accessing elements within groups in a message in the MRM domain

When an input message is parsed, structures that you have defined as groups in your message set are not represented in the logical tree, but its children are. If you want to refer to or update values for elements that are children of a groups, do not include the group in the ESQL statement. Groups do not have tags that appear in instance messages, and do not appear in user trace of the logical message tree.

Consider the following Video message:


```

<Customer xmlns:addr="http://www.ibm.com/AddressDetails"
xmlns:brw="http://www.ibm.com/BorrowedDetails">
  <Name LastName="Bloggs">
    <Title>Mr</Title>
    <FirstName>Fred</FirstName>
  </Name>
  <addr:Address>
    <HouseNo>13</HouseNo>
    <Street>Oak Street</Street>
    <Town>Southampton</Town>
  </addr:Address>
  <ID>P</ID>
  <PassportNo>J123456TT</PassportNo>
  <brw:Borrowed>
    <VideoTitle>Fast Cars</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.50</Cost>
  </brw:Borrowed>
  <brw:Borrowed>
    <VideoTitle>Cut To The Chase</VideoTitle>
    <DueDate>2003-05-23T01:00:00</DueDate>
    <Cost>3.00</Cost>
  </brw:Borrowed>
  <Magazine>0</Magazine>
</Customer>

```

When the input message is parsed, values are stored in the logical tree as shown in the following section of user trace:

```

(0x0100001B):MRM = (
  (0x01000013):Name = (
    (0x0300000B):LastName = 'Bloggs'
    (0x0300000B):Title = 'Mr'
    (0x0300000B):FirstName = 'Fred'
  )
  (0x01000013)http://www.ibm.com/AddressDetails:Address = (
    (0x0300000B):HouseNo = 13
    (0x0300000B):Street = 'Oak Street'
    (0x0300000B):Town = 'Southampton'
  )
  (0x0300000B):ID = 'P'
  (0x0300000B):PassportNo = 'J123456TT'
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Fast Cars'
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.50
  )
  (0x01000013)http://www.ibm.com/BorrowedDetails:Borrowed = (
    (0x0300000B):VideoTitle = 'Cut To The Chase '
    (0x0300000B):DueDate = TIMESTAMP '2003-05-23 00:00:00'
    (0x0300000B):Cost = 3.00
  )
  (0x0300000B):Magazine = FALSE

```

Immediately following the element named ID, the MRM message definition uses a group which has a *Composition* of Choice. The group is defined with three children: PassportNo, DrivingLicenceNo, and CreditCardNo. The choice composition dictates that instance documents must use only one of these three possible alternatives. The example shown above uses the PassportNo element.

When you refer to this element in ESQL statements, you do not specify the group to which the element belongs. For example:

```
SET OutputRoot.MRM.PassportNo = 'J999999TT';
```

If you define messages within message sets that include XML and TDS physical formats, you can determine from the message data which option of a choice has been taken, because the tags in the message represent one of the choice's options. However, if your messages have CWF physical format, or are non-tagged TDS messages, it is not clear from the message data, and the application programs processing the message must determine which option of the choice has been selected. This is known as *unresolved choice handling*. For further information, see the description of the value of Choice in Complex type logical properties.

Accessing mixed content in a message in the MRM domain

When you define a complex type in a message model, you can optionally specify its content to be mixed. This setting, in support of mixed content in XML Schema, allows you to manipulate data that is included between elements in the message.

Consider the following example:

```
<MRM>
  <Mess1>
    abc
    <Elem1>def</Elem1>
    ghi
    <Elem2>jkl</Elem2>
    mno
    <Elem3>pqr</Elem3>
  </Mess1>
</MRM>
```

The strings abc, ghi, and mno do not represent the value of a particular element (unlike def, for example, which is the value of element Elem1). The presence of these strings means that you must model Mess1 with mixed content. You can model this XML message in the MRM using the following objects:

Message

The message *Name* property is set to Mess1 to match the XML tag.

The *Type* property is set to tMess1.

Type The complex type *Name* property is set to tMess1.

The *Composition* property is set to OrderedSet.

The complex type has mixed content.

The complex type contains the following objects:

Element

The *Name* property is set to Elem1 to match the XML tag.

The *Type* property is set to simple type xsd:string.

Element

The *Name* property is set to Elem2 to match the XML tag.

The *Type* property is set to simple type xsd:string.

Element

The *Name* property is set to Elem3 to match the XML tag.

The *Type* property is set to simple type xsd:string.

If you code the following ESQL:

```

SET OutputRoot.MRM.*[1] = InputBody.Elem3;
SET OutputRoot.MRM.Elem1 = InputBody.*[5];
SET OutputRoot.MRM.*[3] = InputBody.Elem2;
SET OutputRoot.MRM.Elem2 = InputBody.*[3];
SET OutputRoot.MRM.*[5] = InputBody.Elem1;
SET OutputRoot.MRM.Elem3 = InputBody*[1];

```

the mixed content is successfully mapped to the following output message:

```

<MRM>
  <Mess1>
    pqr
    <Elem1>mno</Elem1>
    jk1
    <Elem2>ghi</Elem2>
    def
    <Elem3>abc</Elem3>
  </Mess1>
</MRM>

```

Accessing embedded messages in the MRM domain

If you have defined a multipart message, you have at least one message embedded within another. Within the overall complex type that represents the outer messages, you can model the inner message in two ways.

You can model the inner message in the following ways:

- An element (named `E_outer1` in the following example) with its *Type* property set to a complex type that has been defined with its *Composition* property set to `Message`
- A complex type with its *Composition* property set to `Message` (named `t_Embedded` in the following example)

The ESQL that you need to write to manipulate the inner message varies depending on which of the above models you have used. For example, assume that you have defined:

- An outer message `M_outer` that has its *Type* property set to `t_Outer`.
- An inner message `M_inner1` that has its *Type* set to `t_Inner1`
- An inner message `M_inner2` that has its *Type* set to `t_Inner2`
- Type `t_Outer` that has its first child element named `E_outer1` and its second child defined as a complex type named `t_Embedded`
- Type `t_Embedded` that has its *Composition* property set to `Message`
- Type `t_Inner1` that has its first child element named `E_inner11`
- Type `t_Inner2` that has its first child element named `E_inner21`
- Type `t_outer1` that has its *Composition* property set to `Message`
- Element `E_outer1` that has its *Type* property set to `t_outer1`

If you want to set the value of `E_inner11`, code the following ESQL:

```
SET OutputRoot.MRM.E_outer1.M_inner1.E_inner11 = 'FRED';
```

If you want to set the value of `E_inner21`, code the following ESQL:

```
SET OutputRoot.MRM.M_inner2.E_inner21 = 'FRED';
```

If you copy message headers from the input message to the output message, and your input message type contains a path, only the outermost name in the path is copied to the output message type.

When you configure a message flow to handle embedded messages, you can specify the path of a message type in either an MQRFH2 header (if one is present in the input message) or in the input node *Message Type* property in place of a name (for example, for the message modeled above, the path could be specified as M_Outer/M_Inner1/M_Inner2 instead of just M_Outer).

If you have specified that the input message has a physical format of either CWF or XML, any message type prefix is concatenated in front of the message type from the MQRFH2 or input node, giving a final path to use (for more information refer to Multipart messages). The MRM uses the first item in the path as the outermost message type, then progressively works inwards when it finds a complex type with its *Composition* property set to Message.

If you have specified that the input message has a physical format of TDS, a different process that uses message keys is implemented. This is described in TDS format: Multipart messages.

For more information about path concatenations, see Message set properties.

Accessing the content of a message in the MRM domain with namespace support enabled

Use namespaces where appropriate for messages that are parsed by the MRM parser.

When you want to access elements of a message and namespaces are enabled, you must include the namespace when you code the ESQL reference to the element. If you do not do so, the broker searches the no target namespace. If the element is not found in the no target namespace, the broker searches all other known namespaces in the message dictionary (that is, within the deployed message set). For performance and integrity reasons, specify namespaces wherever they apply.

The most efficient way to refer to elements when namespaces are enabled is to define a namespace constant, and use this in the appropriate ESQL statements. This technique makes your ESQL code much easier to read and maintain.

Define a constant using the DECLARE NAMESPACE statement:

```
DECLARE ns01 NAMESPACE 'http://www.ns01.com'  
.  
.  
SET OutputRoot.MRM.Element1 = InputBody.ns01:Element1;
```

ns01 is interpreted correctly as a namespace because of the way that it is declared.

You can also use a CHARACTER variable to declare a namespace:

```
DECLARE ns02 CHARACTER 'http://www.ns02.com'  
.  
.  
SET OutputRoot.MRM.Element2 = InputBody.{ns02}:Element2;
```

If you use this method, you must surround the declared variable with braces to ensure that it is interpreted as a namespace.

If you are concerned that a CHARACTER variable might get changed, you can use a CONSTANT CHARACTER declaration:

```
DECLARE ns03 CONSTANT CHARACTER 'http://www.ns03.com'  
.  
.  
SET OutputRoot.MRM.Element3 = InputBody.{ns03}:Element3;
```

You can declare a namespace, constant, and variable within a module or function. However, you can declare only a namespace or constant in schema scope (that is, outside a module scope).

The following sample provides further examples of the use of namespaces:

- Video Rental

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

Querying null values in a message in the MRM domain

You can use an ESQL statement to compare an element to NULL.

If you want to compare an element to NULL, code the statement:

```
IF InputRoot.MRM.Elem2.Child1 IS NULL THEN  
  DO:  
    -- more ESQL --  
END IF;
```

If nulls are permitted for this element, this statement tests whether the element exists in the input message, or whether it exists and contains the MRM-supplied null value. The behavior of this test depends on the physical format:

- For an XML element, if the XML tag or attribute is not in the bit stream, this test returns TRUE.
- For an XML element, if the XML tag or attribute is in the bit stream and contains the MRM null value, this test returns TRUE.
- For an XML element, if the XML tag or attribute is in the bit stream and does not contain the MRM null value, this test returns FALSE.
- For a delimited TDS element, if the element has no value between the previous delimiter and its delimiter, this test returns TRUE.
- For a delimited TDS element, if the element has a value between the previous delimiter and its delimiter that is the same as the MRM-defined null value for this element, this test returns TRUE.
- For a delimited TDS element, if the element has a value between the previous delimiter and its delimiter that is not the MRM-defined null value, this test returns FALSE.
- For a CWF or fixed length TDS element, if the element's value is the same as the MRM-defined null value for this element, this test returns TRUE.
- For a CWF or fixed length TDS element, if the element's value is not the same as the MRM-defined null value, this test returns FALSE.

If you want to determine if the field is missing, rather than present but with null value, you can use the ESQL CARDINALITY function.

Setting null values in a message in the MRM domain

You can use implicit or explicit null processing to set the value of an element to NULL in an output message.

To set a value of an element in an output message, you normally code an ESQL statement similar to the following:

```
SET OutputRoot.MRM.Elem2.Child1 = 'xyz';
```

or its equivalent statement:

```
SET OutputRoot.MRM.Elem2.Child1 VALUE = 'xyz';
```

If you set the element to a non-null value, these two statements give identical results. However, if you want to set the value to null, these two statements do not give the same result:

1. If you set the element to NULL using the following statement, the element is deleted from the message tree:

```
SET OutputRoot.MRM.Elem2.Child1 = NULL;
```

The content of the output bit stream depends on the physical format:

- For an XML element, neither the XML tag or attribute nor its value are included in the output bit stream.
- For a Delimited TDS element, neither the tag (if appropriate) nor its value are included in the output bit stream. The absence of the element is typically conveyed by two adjacent delimiters.
- For a CWF or Fixed Length TDS element, the content of the output bit stream depends on whether you have set the *Default Value* property for the element. If you have set this property, the default value is included in the bit stream. If you have not set the property, an exception is raised.

This is called implicit null processing.

2. If you set the value of this element to NULL as follows:

```
SET OutputRoot.MRM.Elem2.Child1 VALUE = NULL;
```

the element is not deleted from the message tree. Instead, a special value of NULL is assigned to the element. The content of the output bit stream depends on the settings of the physical format null-handling properties.

This is called explicit null processing.

Setting a complex element to NULL deletes that element and all its children.

Working with MRM messages and bit streams

When you use the ASBITSTREAM function or the CREATE FIELD statement with a PARSE clause you must consider various restrictions.

The ASBITSTREAM function

If you code the ASBITSTREAM function with the parser mode option set to *RootBitStream*, to parse a message tree to a bit stream, the result is an MRM document in the format specified by the message format that is built from the children of the target element in the normal way.

The target element must be a predefined message defined within the message set, or can be a self-defined message if you are using an XML physical format. This algorithm is identical to that used to generate the normal output bit stream. A well-formed bit stream obtained in this way can be used to re-create the original tree by using a CREATE statement with a PARSE clause.

If you code the ASBITSTREAM function with the parser mode option set to *FolderBitStream*, to parse a message tree to a bit stream, the generated bit stream is an MRM element built from the target element and its children. Unlike *RootBitStream* mode the target element does not have to represent a message; it can represent a predefined element within a message or self-defined element within a message.

So that the MRM parser can correctly parse the message, the path from the message to the target element within the message must be specified in the *Message Type*. The format of the path is the same as that used by message paths except that the message type prefix is not used.

For example, suppose the following message structure is used:

```
Message
  elem1
    elem11
    elem12
```

To serialize the subtree representing element `elem12` and its children, specify the message path `'message/elem1/elem12'` in the *Message Type*.

If an element in the path is qualified by a namespace, specify the namespace URI between `{}` characters in the message path. For example if element `elem1` is qualified by namespace `'http://www.ibm.com/temp'`, specify the message path as `'message/{http://www.ibm.com/temp}elem1/elem12'`

This mode can be used to obtain a bit stream description of arbitrary sub-trees owned by an MRM parser. When in this mode, with a physical format of XML, the XML bit stream generated is not enclosed by the 'Root Tag Name' specified for the Message in the Message Set. No XML declaration is created, even if not suppressed in the message set properties.

Bit streams obtained in this way can be used to re-create the original tree by using a CREATE statement with a PARSE clause (by using a mode of *FolderBitStream*).

The CREATE statement with a PARSE clause

If you code a CREATE statement with a PARSE clause, with the parser mode option set to *RootBitStream*, to parse a bit stream to a message tree, the expected bit stream is a normal MRM document. A field in the tree is created for each field in the document. This algorithm is identical to that used when parsing a bit stream from an input node

If you code a CREATE statement with a PARSE clause, with the parser mode option set to *FolderBitStream*, to parse a bit stream to a message tree, the expected bit stream is a document in the format specified by the Message Format, which is either specified directly or inherited. Unlike *RootBitStream* mode the root of the document does not have to represent an MRM message; it can represent a predefined element within a message or self-defined element within a message.

So that the MRM parser can correctly parse the message the path from the message to the target element within the message must be specified in the *Message Type*. The format of the message path is the same as that used for the ASBITSTREAM function described above.

Example of using the ASBITSTREAM function and CREATE statement with a PARSE clause in FolderBitStream mode

The following ESQL uses the message definition described above. The ESQL serializes part of the input tree by using the ASBITSTREAM function, then uses the CREATE statement with a PARSE clause to re-create the subtree in the output tree. The Input message and corresponding Output message are shown below the ESQL.

```

CREATE COMPUTE MODULE DocSampleFlow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyMessageHeaders();

  -- Set the options to be used by ASBITSTREAM and CREATE ... PARSE
  -- to be FolderBitStream and enable validation
  DECLARE parseOptions INTEGER BITOR(FolderBitStream, ValidateContent,
                                     ValidateValue, ValidateLocalError);

  -- Serialise the elem12 element and its children from the input bitstream
  -- into a variable
  DECLARE subBitStream BLOB
  ASBITSTREAM(InputRoot.MRM.elem1.elem12
              OPTIONS parseOptions
              SET 'DocSample'
              TYPE 'message/elem1/elem12'
              FORMAT 'XML1');

  -- Set the value of the first element in the output tree
  SET OutputRoot.MRM.elem1.elem11 = 'val11';

  -- Parse the serialized sub-tree into the output tree
  IF subBitStream IS NOT NULL THEN
    CREATE LASTCHILD OF OutputRoot.MRM.elem1
      PARSE ( subBitStream
              OPTIONS parseOptions
              SET 'DocSample'
              TYPE 'message/elem1/elem12'
              FORMAT 'XML1');
  END IF;

  -- Convert the children of elem12 in the output tree to uppercase
  SET OutputRoot.MRM.elem1.elem12.elem121 =
    UCASE(OutputRoot.MRM.elem1.elem12.elem121);

  SET OutputRoot.MRM.elem1.elem12.elem122 =
    UCASE(OutputRoot.MRM.elem1.elem12.elem122);

  -- Set the value of the last element in the output tree
  SET OutputRoot.MRM.elem1.elem13 = 'val13';

  RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
  DECLARE I INTEGER 1;
  DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
  WHILE I < J DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I = I + 1;
  END;

```



```
    END WHILE;  
    END;  
  
END MODULE;
```

Input message :

```
<message>  
  <elem1>  
    <elem11>value11</elem11>  
    <elem12>  
      <elem121>value121</elem121>  
      <elem122>value122</elem122>  
    </elem12>  
    <elem13>value13</elem13>  
  </elem1>  
</message>
```

Output message :

```
<message>  
  <elem1>  
    <elem11>val11</elem11>  
    <elem12>  
      <elem121>VALUE121</elem121>  
      <elem122>VALUE122</elem122>  
    </elem12>  
    <elem13>val13</elem13>  
  </elem1>  
</message>
```

Handling large MRM messages

When an input bit stream is parsed, and a logical tree created, the tree representation of an MRM message is typically larger, and in some cases much larger, than the corresponding bit stream.

The reasons for this large size include:

- The addition of the pointers that link the objects together
- Translation of character data into Unicode that can double the original size
- The inclusion of field names that can be contained implicitly within the bit stream
- The presence of control data that is associated with the broker's operation

Manipulation of a large message tree can, therefore, demand a great deal of storage. If you design a message flow that handles large messages made up of repeating structures, you can code specific ESQL statements that help to reduce the storage load on the broker. These statements support both random and sequential access to the message, but assume that you do not need access to the whole message at one time.

These ESQL statements cause the broker to perform limited parsing of the message, and to keep only that part of the message tree that reflects a single record in storage at a time. If your processing requires you to retain information from record to record (for example, to calculate a total price from a repeating structure of items in an order), you can either declare, initialize, and maintain ESQL variables, or you can save values in another part of the message tree, for example LocalEnvironment.

This technique reduces the memory used by the broker to that needed to hold the full input and output bit streams, plus that required for one record's trees. It

provides memory savings when even a small number of repeats is encountered in the message. The broker makes use of partial parsing, and the ability to parse specified parts of the message tree, to and from the corresponding part of the bit stream.

To use these techniques in your Compute node apply these general techniques:

- Copy the body of the input message as a bit stream to a special folder in the output message. This creates a modifiable copy of the input message that is not parsed and which therefore uses a minimum amount of memory.
- Avoid any inspection of the input message; this avoids the need to parse the message.
- Use a loop and a reference variable to step through the message one record at a time. For each record:
 - Use normal transforms to build a corresponding output subtree in a second special folder.
 - Use the ASBITSTREAM function to generate a bit stream for the output subtree that is stored in a *BitStream* element, placed in the position in the tree, that corresponds to its required position in the final bit stream.
 - Use the DELETE statement to delete both the current input and the output record message trees when you complete their manipulation.
 - When you complete the processing of all records, detach the special folders so that they do not appear in the output bit stream.

You can vary these techniques to suit the processing that is required for your messages. The following ESQL provides an example of one implementation.

The ESQL is dependant on a message set called `LargeMessageExample` that has been created to define messages for both the Invoice input format and the Statement output format. A message called `AllInvoices` has been created that contains a global element called `Invoice` that can repeat one or more times, and a message called `Data` that contains a global element called `Statement` that can repeat one or more times.

The definitions of the elements and attributes have been given the correct data types, therefore, the CAST statements used by the ESQL in the XML example are no longer required. An XML physical format with name `XML1` has been created in the message set which allows an XML message corresponding to these messages to be parsed by the MRM.

When the Statement tree is serialized using the ASBITSTREAM function the *Message Set*, *Message Type*, and *Message Format* are specified as parameters. The *Message Type* parameter contains the path from the message to the element being serialized which, in this case, is `Data/Statement` because the Statement element is a direct child of the Data message.

The input message to the flow is the same Invoice example message used in other parts of the documentation except that it is contained between the tags:

```
<AllInvoices> .... </AllInvoices>
```

```
CREATE COMPUTE MODULE LargeMessageExampleFlow_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyMessageHeaders();
  -- Create a special folder in the output message to hold the input tree
  -- Note : SourceMessageTree is the root element of an MRM parser
  CREATE LASTCHILD OF OutputRoot.MRM DOMAIN 'MRM' NAME 'SourceMessageTree';
```

```

-- Copy the input message to a special folder in the output message
-- Note : This is a root to root copy which will therefore not build trees
SET OutputRoot.MRM.SourceMessageTree = InputRoot.MRM;

-- Create a special folder in the output message to hold the output tree
CREATE FIELD OutputRoot.MRM.TargetMessageTree;

-- Prepare to loop through the purchased items
DECLARE sourceCursor REFERENCE TO OutputRoot.MRM.SourceMessageTree.Invoice;
DECLARE targetCursor REFERENCE TO OutputRoot.MRM.TargetMessageTree;
DECLARE resultCursor REFERENCE TO OutputRoot.MRM;
DECLARE grandTotal  FLOAT    0.0e0;

-- Create a block so that it's easy to abandon processing
ProcessInvoice: BEGIN
-- If there are no Invoices in the input message, there is nothing to do
IF NOT LASTMOVE(sourceCursor) THEN
    LEAVE ProcessInvoice;
END IF;

-- Loop through the invoices in the source tree
InvoiceLoop : LOOP
-- Inspect the current invoice and create a matching Statement
SET targetCursor.Statement =
    THE (
        SELECT
            'Monthly'                AS Type,
            'Full'                    AS Style,
            I.Customer.FirstName       AS Customer.Name,
            I.Customer.LastName       AS Customer.Surname,
            I.Customer.Title          AS Customer.Title,
            (SELECT
                FIELDVALUE(II.Title)  AS Title,
                II.UnitPrice * 1.6     AS Cost,
                II.Quantity            AS Qty
            FROM I.Purchases.Item[] AS II
            WHERE II.UnitPrice > 0.0    ) AS Purchases.Article[],
            (SELECT
                SUM( II.UnitPrice *
                    II.Quantity *
                    1.6                )
            FROM I.Purchases.Item[] AS II
            'Dollars'                  AS Amount.Currency
            FROM sourceCursor AS I
            WHERE I.Customer.LastName <> 'White'
        );

-- Turn the current Statement into a bit stream
-- The SET parameter is set to the name of the message set
-- containing the MRM definition
-- The TYPE parameter contains the path from the from the message
-- to element being serialized
-- The FORMAT parameter contains the name of the physical format
-- name defined in the message
DECLARE StatementBitStream BLOB
ASBITSTREAM(targetCursor.Statement
    OPTIONS FolderBitStream
    SET 'LargeMessageExample'
    TYPE 'Data/Statement'
    FORMAT 'XML1');

-- If the SELECT produced a result (that is, it was not filtered
-- out by the WHERE clause), process the Statement
IF StatementBitStream IS NOT NULL THEN
-- create a field to hold the bit stream in the result tree
-- The Type of the element is set to MRM.BitStream to indicate

```

```

-- to the MRM Parser that this is a bitstream
CREATE LASTCHILD OF resultCursor
  Type MRM.BitStream
  NAME 'Statement'
  VALUE StatementBitStream;

-- Add the current Statement's Amount to the grand total
SET grandTotal = grandTotal + targetCursor.Statement.Amount;
END IF;

-- Delete the real Statement tree leaving only the bit stream version
DELETE FIELD targetCursor.Statement;

-- Step onto the next Invoice, removing the previous invoice and any
-- text elements that might have been interspersed with the Invoices
REPEAT
  MOVE sourceCursor NEXTSIBLING;
  DELETE PREVIOUSIBLING OF sourceCursor;
UNTIL (FIELDNAME(sourceCursor) = 'Invoice')
  OR (LASTMOVE(sourceCursor) = FALSE)
END REPEAT;

-- If there are no more invoices to process, abandon the loop
IF NOT LASTMOVE(sourceCursor) THEN
  LEAVE InvoiceLoop;
END IF;

END LOOP InvoiceLoop;
END ProcessInvoice;

-- Remove the temporary source and target folders
DELETE FIELD OutputRoot.MRM.SourceMessageTree;
DELETE FIELD OutputRoot.MRM.TargetMessageTree;

-- Finally add the grand total
SET resultCursor.GrandTotal = grandTotal;

-- Set the output MessageType property to be 'Data'
SET OutputRoot.Properties.MessageType = 'Data';

RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
  DECLARE I INTEGER 1;
  DECLARE J INTEGER CARDINALITY(InputRoot.*[I]);
  WHILE I < J DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I = I + 1;
  END WHILE;
END;

END MODULE;

```

Manipulating messages in the JMS domains

This topic provides information specific to dealing with messages that belong to the JMSMap and JMSStream domains. These messages are parsed by the generic XML parser.

Messages that belong to the JMS domains are processed by the XML parser, so you can follow the guidance provided for XML messages in “Manipulating messages in the XML domain” on page 132, in conjunction with the information in “Manipulating message body content” on page 28.

The JMSMap and JMSStream domains support MapMessage and StreamMessage messages. Other kinds of JMS message are supported by other domains. For further information about using JMS messages with WebSphere Message Broker, see WebSphere Broker JMS Transport.

Manipulating messages in the IDOC domain

Use ESQL from a Compute node to copy the incoming IDoc to the outgoing IDoc, and manipulate the message.

A valid IDoc message flows out of SAP and is sent to the MQSeries® link for R/3.

When this IDoc has been committed successfully to the outbound WebSphere MQ queue, the input node of the message flow reads it from that queue and generates the syntax element tree.

The Compute node manipulates this syntax element tree and, when it has finished, passes the output message to subsequent nodes in the message flow. When the message reaches the output node, the IDOC parser is called to rebuild the bit stream from the tree.

The message flow must create an output message in a similar format to the input message.

See Field names of the IDOC parser structures for the field names in the DC (Control Structure) and DD (Data Structure) that are recognized by the IDOC parser

Use the following ESQL example from a Compute node:

```
SET OutputRoot = InputRoot;
SET OutputRoot.IDOC.DC[1].tabnam = 'EDI_DC40 ';
SET OutputRoot.IDOC.DD[2].sdatatag.MRM.maktx = 'Buzzing all day';
```

The first line of the code copies the incoming IDoc to the outgoing IDoc.

The second line sets the *tablename* of the first DC.

The third line uses the second DD segment, which in this example is of type E2MAKTM001, and sets the *maktx* field.

Accessing fields of the IDoc using ESQL

Use the ESQL editor Content Assist to fill in the SAP-defined fields of the IDoc.

After the *sdatatag* tag in an ESQL statement, the next tag is *MRM*, which you must enter manually, followed by the field name that is to be manipulated. Specify the name of the field within the message segment here, not the name of the message segment.

For example, the following code sets the segment name of the IDoc:

```
SET OutputRoot.IDOC.DD[I].segnam = 'E2MAKTM001';
```

The following example sets the *msgfn* field within the E2MAKTM001 segment:

```
SET OutputRoot.IDOC.DD[I].sdatatag.MRM.msgfn = '006';
```

Manipulating messages in the MIME domain

A MIME message does not need to be received over a particular transport. For example, a message can be received over HTTP by using an HTTPInput node, or over WebSphere MQ by using an MQInput node. The MIME parser is used to process a message if the message domain is set to MIME in the input node properties, or if you are using WebSphere MQ, and the MQRFH2 header has a message domain of MIME.

This topic explains how to deal with messages that belong to the MIME domain, and are parsed by the MIME parser. Use this information in conjunction with the information in “Manipulating message body content” on page 28.

You can manipulate the logical tree using ESQL before passing the message on to other nodes in the message flow. A message flow can also create a MIME domain tree using ESQL. When a MIME domain message reaches an output node, the MIME parser is called to rebuild the bit stream from the logical tree.

The following examples show how to manipulate MIME messages:

- “Creating a new MIME tree”
- “Modifying an existing MIME tree” on page 151
- “Managing Content-Type” on page 151

Creating a new MIME tree

A message flow often receives, modifies, and returns a MIME message. In this case, you can work with the valid MIME tree that is created when the input message is parsed. If a message flow receives input from another domain, such as XMLNS, and returns a MIME message, you must create a valid MIME tree. Use the following ESQL example in a Compute node to create the top-level structure for a single-part MIME tree:

```
CREATE FIELD OutputRoot.MIME TYPE Name;  
DECLARE M REFERENCE TO OutputRoot.MIME;  
CREATE LASTCHILD OF M TYPE Name NAME 'Data';
```

The message flow must also ensure that the MIME Content-Type is set correctly, as explained in “Managing Content-Type” on page 151. The flow must then add the message data into the MIME tree. The following ESQL examples show how you can do this. In each case, a Data element is created with the domain BLOB.

- A bit stream from another part of the tree is used. This example shows how a bit stream could be created from an XML message that is received by the message flow. The flow then invokes the BLOB parser to store the data under the Data element.

```
DECLARE partData BLOB ASBITSTREAM(InputRoot.XMLNS);  
CREATE LASTCHILD OF M.Data DOMAIN('BLOB') PARSE(partData);
```

- Instead of parsing the bit stream, create the new structure, then attach the data to it, as shown in this ESQL example:

```
DECLARE partData BLOB ASBITSTREAM(InputRoot.XMLNS);  
CREATE LASTCHILD OF M.Data DOMAIN('BLOB') NAME 'BLOB';  
CREATE LASTCHILD OF M.Data.BLOB NAME 'BLOB' VALUE partData;
```

Both of these approaches create the same tree structure. The first approach is better because explicit knowledge of the tree structure that the BLOB parser requires is not built into the flow.

More commonly, the Compute node must build a tree for a multipart MIME document. The following ESQL example shows how you can do this, including setting the top-level Content-Type property.

```

DECLARE part1Data BLOB ASBITSTREAM(InputRoot.XMLNS, InputProperties.Encoding, InputProperties.CodedCharSetId);
SET OutputRoot.Properties.ContentType = 'multipart/related; boundary=myBoundary';

CREATE FIELD OutputRoot.MIME TYPE Name;
DECLARE M REFERENCE TO OutputRoot.MIME;
CREATE LASTCHILD OF M TYPE Name NAME 'Parts';
CREATE LASTCHILD OF M.Parts TYPE Name NAME 'Part';
DECLARE P1 REFERENCE TO M.Parts.Part[1];
CREATE FIELD P1."Content-Type" TYPE NameValue VALUE 'text/plain';
CREATE FIELD P1."Content-Id" TYPE NameValue VALUE 'part one';
CREATE LASTCHILD OF P1 TYPE Name NAME 'Data';
CREATE LASTCHILD OF P1.Data DOMAIN('BLOB') PARSE(part1Data);

CREATE LASTCHILD OF M.Parts TYPE Name NAME 'Part';
DECLARE P2 REFERENCE TO M.Parts.Part[2];
CREATE FIELD P2."Content-Type" TYPE NameValue VALUE 'text/plain';
CREATE FIELD P2."Content-Id" TYPE NameValue VALUE 'part two';
CREATE LASTCHILD OF P2 TYPE Name NAME 'Data';
CREATE LASTCHILD OF P2.Data DOMAIN('BLOB') PARSE(part2Data);

```

Modifying an existing MIME tree

This ESQL example adds a new MIME part to an existing multipart MIME message. If the message is not multipart, it is not modified.

```

SET OutputRoot = InputRoot;

-- Check to see if the MIME message is multipart or not.
IF LOWER(InputProperties.ContentType) LIKE 'multipart/%'
THEN
  CREATE LASTCHILD OF OutputRoot.MIME.Parts NAME 'Part';

  DECLARE P REFERENCE TO OutputRoot.MIME.Parts.[<];
  CREATE FIELD P."Content-Type" TYPE NameValue VALUE 'text/xml';
  CREATE FIELD P."Content-ID" TYPE NameValue VALUE 'new part';
  CREATE LASTCHILD OF P TYPE Name NAME 'Data';

  -- This is an artificial way of creating some BLOB data.
  DECLARE newBlob BLOB '4f6e652074776f2074687265650d0a';
  CREATE LASTCHILD OF P.Data DOMAIN('BLOB') PARSE(newBlob);
END IF;

```

Managing Content-Type

When you create a new MIME message tree, or when you modify the value of the MIME boundary string, make sure that the MIME Content-Type header is set correctly by setting the ContentType value in the broker Properties subtree. The following example shows how to set the ContentType value for a MIME part with simple content:

```
SET OutputRoot.Properties.ContentType = 'text/plain';
```

Do not set the Content-Type value directly in the MIME tree or HTTP trees because the value is ignored or used inconsistently.

Manipulating messages in the BLOB domain

How to deal with messages that belong to the BLOB domain, and that are parsed by the BLOB parser.

You cannot manipulate the contents of a BLOB message, because it has no predefined structure. However, you can refer to its contents using its known position within the bit stream, and process the message with a minimum of knowledge about its contents.

The BLOB message body parser does not create a tree structure in the same way that other message body parsers do. It has a root element BLOB, that has a child element, also called BLOB, that contains the data.

You can refer to message content using substrings if you know the location of a particular piece of information within the BLOB data. For example, the following expression identifies the tenth byte of the message body:

```
InputBody.BLOB.BLOB[10]
```

The following expression references 10 bytes of the message data starting at offset 10:

```
SUBSTRING(InputBody.BLOB.BLOB from 10 for 10)
```

You can use the Mapping node to map to and from a predefined BLOB message, and to map to and from items of BLOB data.

Simple example to write a string in the output message

The following simple example allows you to write some character data in your ESQL (for example, if you have read some character fields from a database) out as a BLOB:

```
CALL CopyMessageHeaders();  
-- CALL CopyEntireMessage();  
DECLARE mystring CHARACTER;  
SET mystring='hello';  
SET OutputRoot.BLOB.BLOB=CAST (mystring AS BLOB CCSID 1208);
```

Using the CALL statement to call a user-written routine

The ESQL CALL statement calls routines that have been created and implemented in different ways.

A routine is a user-defined function or procedure that has been defined by one of the following statements:

- CREATE FUNCTION
- CREATE PROCEDURE

You can use the CALL statement to call a routine that has been implemented in any of the following ways:

- ESQL
- Java
- As a stored procedure in a database
- As a built-in (broker-provided) function

You can use CALL to call built-in (broker-provided) functions and user-defined SQL functions, but typically you would use their names directly in expressions .

For details of the syntax and parameters of the CALL statement, see “CALL statement” on page 196. For an example of the use of CALL, see the examples in “CREATE PROCEDURE statement” on page 220.

Calling an ESQL routine

A routine is called as an ESQL method if the routine's definition specifies a LANGUAGE clause of ESQL or if the routine is a built-in function. An exact one-to-one matching of the data types and directions of each parameter, between the definition and the CALL, is required. An ESQL routine is allowed to return any ESQL data type, excluding List and Row.

Calling a Java routine

A routine is called as a Java method if the routine's definition specifies a LANGUAGE clause of JAVA. An exact one-to-one matching of the data types and directions of each parameter, between the definition and the CALL, is required. If the Java method has a void return type, the INTO clause cannot be used because no value exists to return.

A Java routine can return any data type in the “ESQL-to-Java data-type mapping table” on page 171, excluding List and Row.

Calling a database stored procedure

A routine is called as a database stored procedure if the routine's definition has a LANGUAGE clause of DATABASE.

When a call is made to a database stored procedure, the broker searches for a definition (created by a CREATE PROCEDURE statement) that matches the procedure's local name. The broker then uses the following sequence to resolve the name by which the procedure is known in the database and the database schema to which it belongs:

1. If the CALL statement specifies an IN clause, the name of the data source, the database schema, or both, is taken from the IN clause.
2. If the name of the data source is not provided by an IN clause on the CALL statement, it is taken from the DATASOURCE attribute of the node.
3. If the database schema is not provided by an IN clause on the CALL statement, but is specified on the EXTERNAL NAME clause of the CREATE PROCEDURE statement, it is taken from the EXTERNAL NAME clause.
4. If no database schema is specified on the EXTERNAL NAME clause of the CREATE PROCEDURE statement, the database's user name is used as the schema name. If a matching procedure is found, the routine is called.

The chief use of the CALL statement's IN clause is that it allows the data source, the database schema, or both, to be chosen dynamically at run time. (The EXTERNAL SCHEMA clause also allows the database schema which contains the stored procedure to be chosen dynamically, but it is not as flexible as the IN clause and is retained only for compatibility with earlier versions. Its use in new applications is deprecated.)

If the called routine has any DYNAMIC RESULT SETS specified in its definition, the number of expressions in the CALL statement's *ParameterList* must match the number of parameters to the routine, plus the number of DYNAMIC RESULT SETS. For example, if the routine has three parameters and two DYNAMIC RESULT SETS, the CALL statement must pass five parameters to the called routine.

The parameters passed for the two DYNAMIC RESULT SETS must be list parameters; that is, they must be field references qualified with array brackets []; for example, Environment.ResultSet1[].

A database stored procedure is allowed to return any ESQL data type, excluding Interval, List, and Row.

Accessing broker properties from ESQL

You can access broker properties, at run time, from the ESQL modules in your message flow nodes.

You can use broker properties on the right side of regular SET statements. For example:

```
DECLARE mybroker CHARACTER;  
SET mybroker = BrokerName;
```

where BrokerName is the broker property that contains the name of the broker on which the message flow is running. However, you cannot use broker properties on the left side of SET statements. This restriction exists because, at run time, broker properties are constants: they cannot be modified, therefore their values cannot be changed by SET statements. If a program tries to change the value of a broker property, the error message Cannot assign to a symbolic constant is issued.

Broker properties:

- Are grouped by broker, execution group, flow, and node.
- Are case sensitive. Their names always start with an uppercase letter.
- Return NULL if they do not contain a value.

If your ESQL code already contains a variable with the same name as one of the broker properties, your variable takes precedence; that is, your variable masks the broker property. To access the broker property, use the form SQL.<broker_property_name>. For example: SQL.BrokerName.

“Broker properties that are accessible from ESQL and Java” on page 377 shows the broker, flow, and node properties that are accessible from ESQL and indicates which properties are also accessible from Java.

Configuring a message flow at deployment time with user-defined properties

Use user-defined properties (UDPs) to configure message flows at deployment and run time, without modifying program code. You can give a UDP an initial value when you declare it in your program, or when you use the Message Flow editor to create or modify a message flow.

For an overview of user-defined properties, see User-defined properties.

See the “DECLARE statement” on page 235 for an example of how to code a UDP statement.

In ESQL, you can define UDPs at the module or schema level.

After a UDP has been defined by the Message Flow editor, you can modify its value before you deploy it.

To configure UDPs:

1. Switch to the Broker Administration perspective or Broker Application Development perspective.
2. Double-click the broker archive (BAR) file in the Navigator view. The contents of the BAR file are shown in the Manage and Configure page of the Broker Archive editor.
3. Click the **Manage and Configure** tab. This tab shows the message flows in your broker archive; expand a flow to show the individual nodes that it contains.
4. Click the message flow that you are interested in. The UDPs that are defined in that message flow are displayed with their values in the **Properties** view.
5. If the value of the UDP is unsuitable for your current environment or task, change it to the value that you want. The value of the UDP is set at the flow level, and is the same for all eligible nodes that are contained in the flow. If a subflow includes a UDP that has the same name as a UDP in the main flow, the value of the UDP in the subflow is not changed.
6. Save your broker archive.

Now you are ready to deploy the message flow. See [Deploying a broker archive file](#).

Part 2. Reference

ESQL reference	159	RETURN statement	266
Syntax diagrams: available types	160	SET statement	268
ESQL data types in message flows	160	THROW statement	270
ESQL BOOLEAN data type.	160	UPDATE statement	272
ESQL datetime data types	160	WHILE statement	275
ESQL NULL data type	166	ESQL functions: reference material, organized by	
ESQL numeric data types	166	function type	275
ESQL REFERENCE data type	169	Calling ESQL functions	278
ESQL ROW data type	169	ESQL database state functions	279
ESQL string data types	170	ESQL datetime functions	283
ESQL-to-Java data-type mapping table	171	ESQL numeric functions	288
ESQL-to-XPath mapping table	172	ESQL string manipulation functions	303
XPath property editors	173	ESQL field functions	316
ESQL variables	173	ESQL list functions	326
ESQL field reference overview.	174	Complex ESQL functions	329
Namespace	176	Miscellaneous ESQL functions	371
Index	177	ESQL constants	377
Type	178	Broker properties that are accessible from ESQL	
Field references summary	179	and Java	377
Target field references	180	Special characters, case sensitivity, and comments	
The effect of setting a field to NULL	181	in ESQL	380
ESQL operators.	181	ESQL reserved keywords	381
ESQL simple comparison operators	181	ESQL non-reserved keywords	382
ESQL complex comparison operators	182	Example message	385
ESQL logical operators	185		
ESQL numeric operators.	186		
ESQL string operator	187		
Rules for ESQL operator precedence.	187		
ESQL statements	188		
ATTACH statement	189		
BEGIN ... END statement	191		
BROKER SCHEMA statement	193		
CALL statement	196		
CASE statement	199		
CREATE statement	201		
CREATE FUNCTION statement	209		
CREATE MODULE statement	218		
CREATE PROCEDURE statement	220		
DECLARE statement	235		
DECLARE HANDLER statement	240		
DELETE FROM statement	243		
DELETE statement	245		
DETACH statement	246		
EVAL statement	246		
FOR statement	248		
IF statement	248		
INSERT statement	249		
ITERATE statement	253		
LEAVE statement	254		
LOG statement	255		
LOOP statement	256		
MOVE statement	257		
PASSTHRU statement	259		
PROPAGATE statement	261		
REPEAT statement	264		
RESIGNAL statement.	265		

ESQL reference

SQL is the industry standard language for accessing and updating database data and ESQL is a language derived from SQL Version 3, particularly suited to manipulating both database and message data.

This section covers the following topics:

“Syntax diagrams: available types” on page 160

This describes the formats that are available for viewing ESQL syntax diagrams.

“ESQL data types in message flows” on page 160

This describes the valid data types for ESQL.

“ESQL field reference overview” on page 174

This topic describes the syntax of field references.

“Special characters, case sensitivity, and comments in ESQL” on page 380

This describes the special characters you use when writing ESQL statements.

“ESQL operators” on page 181

This describes the operators that are available.

“ESQL reserved keywords” on page 381

This lists the reserved keywords which you cannot use for variable names.

“ESQL non-reserved keywords” on page 382

This lists the keywords that are not reserved, as well as those reserved for future releases, which you can use if you choose.

“ESQL functions: reference material, organized by function type” on page 275

This topic lists the functions available in ESQL, and what they do.

“ESQL constants” on page 377

This topic lists the constants available in ESQL, and what they do.

“ESQL statements” on page 188

This topic lists the different statement types available in ESQL, and what they do.

“Calling ESQL functions” on page 278

This topic describes all the ESQL functions in detail.

“ESQL variables” on page 173

This topic describes the types of ESQL variable and their lifetimes.

“Broker properties that are accessible from ESQL and Java” on page 377

This topic lists the broker attributes that can be accessed from ESQL code.

An XML format message that is used in many of the ESQL examples in these topics is shown in “Example message” on page 385.

For information about how you can use ESQL statements and functions to configure Compute, Database, and Filter nodes, see “Writing ESQL” on page 25.

Syntax diagrams: available types

The syntax for commands and ESQL statements and functions is presented in the form of a diagram. The diagram tells you what you can do with the command, statement, or function and indicates relationships between different options and, sometimes, different values of an option. There are two types of syntax diagrams: railroad diagrams and dotted decimal diagrams. Railroad diagrams are a visual format suitable for sighted users. Dotted decimal diagrams are text-based diagrams that are more helpful for blind or partially-sighted users.

To select which type of syntax diagram you use, click the appropriate button above the syntax diagram in the topic that you are viewing.

The following topics describe how to interpret each type of diagram:

- How to read railroad diagrams
- How to read dotted decimal diagrams

ESQL data types in message flows

All data that is referred to in message flows must be one of the defined types.

The defined types are:

- “ESQL BOOLEAN data type”
- “ESQL datetime data types”
- “ESQL NULL data type” on page 166
- “ESQL numeric data types” on page 166
- “ESQL REFERENCE data type” on page 169
- “ESQL ROW data type” on page 169
- “ESQL string data types” on page 170

ESQL BOOLEAN data type

The BOOLEAN data type holds a Boolean value.

The Boolean value can have the following values:

- TRUE
- FALSE
- UNKNOWN

Boolean literals consist of the keywords TRUE, FALSE, and UNKNOWN. The literals can appear in uppercase or lowercase. For further information about UNKNOWN, see the “IF statement” on page 248.

ESQL datetime data types

ESQL supports several data types that handle datetime values.

The following data types are collectively known as **datetime** data types:

- “ESQL DATE data type” on page 161
- “ESQL TIME data type” on page 161
- “ESQL GMTTIME data type” on page 161
- “ESQL TIMESTAMP data type” on page 161
- “ESQL GMTTIMESTAMP data type” on page 162
- “ESQL INTERVAL data type” on page 162

For information about datetime functions see “ESQL datetime functions” on page 283.

ESQL DATE data type

The DATE data type holds a Gregorian calendar date (year, month, and day).

The format of a DATE literal is the word DATE followed by a space, followed by a date in single quotation marks in the form 'yyyy-mm-dd'. For example:

```
DECLARE MyDate DATE;  
SET MyDate = DATE '2000-02-29';
```

Do not omit leading zeros from the year, month, and day.

ESQL TIME data type

The TIME data type holds a time of day in hours, minutes, seconds, and fractions of a second. The format of a TIME literal is the word TIME followed by a space, followed by a time in single quotation marks in the form 'hh:mm:ss.ffffff'. For example:

```
DECLARE MyTime TIME;  
SET MyTime = TIME '11:49:23.656';
```

Each of the hour, minute, and second fields in a TIME literal must always be two digits; the optional fractional seconds field can be up to 6 digits in length.

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL GMTTIME data type

The GMTTIME data type is similar to the TIME data type, except that its values are interpreted as values in Greenwich Mean Time. GMTTIME literals are defined in a similar way to TIME values. For example:

```
DECLARE MyGetGmttime GMTTIME;  
SET MyGetGmttime = GMTTIME '12:00:00';
```

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL TIMESTAMP data type

The TIMESTAMP data type holds a DATE and a TIME in years, months, days, hours, minutes, seconds, and fractions of a second. The format of a TIMESTAMP literal is the word TIMESTAMP followed by a space, followed by a time stamp in single quotation marks in the form 'yyyy-MM-dd HH:mm:ss.SSSSSS'. For example:

```
DECLARE MyTimeStamp TIMESTAMP;  
SET MyTimeStamp = TIMESTAMP '1999-12-31 23:59:59';
```

The year field must always be four digits in length. The month, day, hour, and minute fields must always be two digits. (Do not omit leading zeros.) The optional fractional seconds field can be 0 - 6 digits long.

For a description of the characters used when formatting a time stamp in the ESQL CAST function, see “Formatting and parsing dateTimes as strings” on page 339

The PutTime reported by WebSphere MQ on z/OS and other times or time stamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL GMTTIMESTAMP data type

The GMTTIMESTAMP data type is similar to the TIMESTAMP data type, except that the values are interpreted as values in Greenwich Mean Time. GMTTIMESTAMP values are defined in a similar way to TIMESTAMP values, for example:

```
DECLARE MyGetGMTTimeStamp GMTTIMESTAMP;  
SET MyGetGMTTimeStamp = GMTTIMESTAMP '1999-12-31 23:59:59.999999';
```

The PutTime reported by WebSphere MQ on z/OS and other times or timestamps can be inconsistent if the CVT field is not set correctly. For details about when this problem can occur, and how to solve it, see The PutTime that is reported by WebSphere MQ on z/OS, and other times or timestamps are inconsistent.

ESQL INTERVAL data type

The INTERVAL data type holds an interval of time.

This data type has a number of subtypes:

- YEAR
- YEAR TO MONTH
- MONTH
- DAY
- DAY TO HOUR
- DAY TO MINUTE
- DAY TO SECOND
- HOUR
- HOUR TO MINUTE
- HOUR TO SECOND
- MINUTE
- MINUTE TO SECOND
- SECOND

All these subtypes describe intervals of time and all can take part in the full range of operations of the INTERVAL type; for example, addition and subtraction operations with values of type DATE, TIME, or TIMESTAMP.

Use the CAST function to convert from one subtype to another, except for intervals described in years and months, or months, which cannot be converted to those described in days, hours, minutes, and seconds.

The split between months and days arises because the number of days in each month varies. An interval of one month and a day is not meaningful, and cannot be sensibly converted into an equivalent interval in numbers of days only.

An interval literal is defined by the syntax:

```
INTERVAL <interval string> <interval qualifier>
```

The format of interval string and interval qualifier are defined by the table below.

Interval qualifier	Interval string format	Example
YEAR	'<year>' or '<sign> <year>'	'10' or '-10'
YEAR TO MONTH	'<year>-<month>' or '<sign> <year>-<month>'	'2-06' or '- 2-06'
MONTH	'<month>' or '<sign> <month>'	'18' or '-18'
DAY	'<day>' or '<sign> <day>'	'30' or '-30'
DAY TO HOUR	'<day> <hour>' or '<sign> <day> <hour>'	'1 02' or '-1 02'
DAY TO MINUTE	'<day> <hour>:<minute>' or '<sign> <day> <hour>:<minute>'	'1 02:30' or '-1 02:30'
DAY TO SECOND	'<day> <hour>:<minute>:<second>' or '<sign> <day> <hour>:<minute>:<second>'	'1 02:30:15' or '-1 02:30:15.333'
HOUR	'<hour>' or '<sign> <hour>'	'24' or '-24'
HOUR TO MINUTE	'<hour>:<minute>' or '<sign> <hour>:<minute>'	'1:30' or '-1:30'
HOUR TO SECOND	'<hour>:<minute>:<second>' or '<sign> <hour>:<minute>:<second>'	'1:29:59' or '-1:29:59.333'
MINUTE	'<minute>' or '<sign> <minute>'	'90' or '-90'
MINUTE TO SECOND	'<minute>:<second>' or '<sign> <minute>:<second>'	'89:59' or '-89:59'
SECOND	'<second>' or '<sign> <second>'	'15' or '-15.7'

Where an interval contains both a year and a month value, a hyphen is used between the two values. In this instance, the month value must be within the range [0, 11]. If an interval contains a month value and no year value, the month value is unconstrained.

A space is used to separate days from the rest of the interval.

If an interval contains more than one of HOUR, MINUTE, and SECOND, a colon is needed to separate the values and all except the leftmost are constrained as follows:

HOUR

0-23

MINUTE

0-59

SECOND

0-59.999...

The largest value of the left-most value in an interval is +/- 2147483647.

Some examples of valid interval values are:

- 72 hours
- 3 days: 23 hours
- 3600 seconds
- 90 minutes: 5 seconds

Some examples of invalid interval values are:

- 3 days: 36 hours
A day field is specified, so the hours field is constrained to [0,23].
- 1 hour: 90 minutes
An hour field is specified, so minutes are constrained to [0,59].

Here are some examples of interval literals:

```
INTERVAL '1' HOUR
INTERVAL '90' MINUTE
INTERVAL '1-06' YEAR TO MONTH
```

Representation of ESQL datetime data types

When your application sends a message to a broker, the way in which the message data is interpreted depends on the content of the message itself and the configuration of the message flow. If your application sends a message to be interpreted either by the generic XML parser, or the MRM parser, that is tailored by an XML physical format, the application can include date or time data that is represented by any of the XML Schema primitive datetime data types.

The XML Schema data type of each piece of data is converted to an ESQL data type, and the element that is created in the logical message tree is of the converted type. If the datetime data in an input message does not match the rules of the chosen schema data type, the values that the parser writes to the logical message tree are modified even if the message is in the MRM domain and you have configured the message flow to validate the input message. (Validation is not available for generic XML messages.)

This has the following effect on the subfields of the input datetime data:

- If any of the subfields of the input message are missing, a default value is written to the logical message tree. This default is substituted from the full timestamp that refers to the beginning of the current epoch: 1970-01-01 00:00:00.
- If the input message contains information for subfields that are not present in the schema, the additional data is discarded. If this occurs, no exception is raised, even if a message in the MRM domain is validated.
- After the data is parsed, it is cast to one of three ESQL datetime data types. These are DATE, TIME, and TIMESTAMP.
 - If a datetime value contains only date subfields, it is cast to an ESQL DATE.
 - If a datetime value contains only time subfields, it is cast to an ESQL TIME.
 - If a datetime value contains both date and time subfields, it is cast to an ESQL TIMESTAMP.

The following examples illustrate these points.

Input data XML Schema data type	Schema rules	Input value in the bit stream	Value written to the logical tree (ESQL data type in brackets)
xsd:dateTime	CCYY-MM-DDThh:mm:ss	2002-12-31T23:59:59	2002-12-31 23:59:59 (TIMESTAMP)
		--24	1970-01-24 (DATE)
		23:59:59	23:59:59 (TIME)
xsd:date	CCYY-MM-DD	2002-12-31	2002-12-31 (DATE)
		2002-12-31T23:59:59	2002-12-31 (DATE)
		-06-24	1970-06-24 (DATE)
xsd:time	hh:mm:ss	14:15:16	14:15:16 (TIME)
xsd:gDay	---DD	---24	1970-01-24 (DATE)
xsd:gMonth	--MM	--12	1970-12-01 (DATE)
xsd:gMonthDay	--MM-DD	--12-31	1970-12-31 (DATE)

Input data XML Schema data type	Schema rules	Input value in the bit stream	Value written to the logical tree (ESQL data type in brackets)
xsd:gYear	CCYY	2002	2002-01-01 (DATE)
xsd:gYearMonth	CCYY-MM	2002-12	2002-12-01 (DATE)

Validation with missing subfields: When you consider which schema datetime data type to use, consider that, if the message is in the MRM domain, and you configure the message flow to validate messages, missing subfields can cause validation exceptions.

The schema data types Gday, gMonth, gMonthDay, gYear, and gYearMonth are used to record particular recurring periods of time. There is potential confusion when validation is turned on, because the recurring periods of time that are used in these schema data types are stored by ESQL as specific points in time.

For example, when the 24th of the month, which is a gDay (a monthly day) type, is written to the logical tree, the missing month and year subfields are supplied from the epoch (January 1970) to provide the specific date 1970-01-24. If you code ESQL to manipulate this date, for example by adding an interval of 10 days, and then generate an output message that is validated, an exception is raised. This is because the result of the calculation is 1970-02-03 which is invalid because the month subfield of the date no longer matches the epoch date.

Use within an MRM domain: In MRM it is possible to define an element that has the logical type of dateTime.

When a dateTime element is parsed, a field is created in the message tree that has the ESQL datatype of CURRENT_TIME or CURRENT_TIMESTAMP. However, the CURRENT_TIME and CURRENT_TIMESTAMP data types do not have the functionality to store timezone information, and the MRM does not adjust the time according to the input timezone and the timezone of the broker.

Although the CURRENT_TIME and CURRENT_TIMESTAMP data types cannot store timezone information, the MRM stores this information as part of the underlying field. This means that if the field is copied between message trees, the timezone information is copied with it, allowing this information to be preserved on output.

Note that the information is preserved only if the field is copied to a field of the same name.

However, if any new field is derived from the original field, the new field does not have the timezone information. This means that if such a field is cast as a character, the new field assumes the timezone of the broker, but its value is not adjusted for any difference between the input timezone and the timezone of the broker.

For example, an input dateTime element containing 2009-02-20T06:08:07-08:00 could be copied from the input message tree to the output message tree and appear in an output message in exactly the same format. However, if the element is cast as character, using format IU, by a broker running GMT the result would be 2009-02-20T06:08:07.000Z.

ESQL NULL data type

All ESQL data types (except REFERENCE) support the concept of the null value. A value of null means that the value is unknown, undefined, or uninitialized. Null values can arise when you refer to message fields that do not exist, access database columns for which no data has been supplied, or use the keyword NULL, which supplies a null literal value.

Null is a distinct state and is not the same as any other value. In particular, for integers it is not the same thing as the value 0 and for character variables it is not the same thing as a string of zero characters. The rules of ESQL arithmetic take null values into account, and you are typically unaware of their existence. Generally, but not always, these rules mean that, if any operand is null, the result is null.

If an expression returns a null value its data type is not, in general, known. All null values, whatever their origin, are therefore treated equally.

This can be regarded as their belonging to the data type NULL, which is a data type that can have just one value, null.

An expression always returns NULL if any of its elements are NULL.

Testing for null values

To test whether a field contains a null value, use the IS operator described in [Operator=](#).

The effect of setting a field to NULL

Take care when assigning a null value to a field. For example, the following command *deletes* the Name field:

```
SET OutputRoot.XMLNS.Msg.Data.Name = NULL; -- this deletes the field
```

The correct way to assign a null value to a field is as follows:

```
SET OutputRoot.XMLNS.Msg.Data.Name VALUE = NULL;  
-- this assigns a NULL value to a field without deleting it
```

ESQL numeric data types

ESQL supports several data types that handle numeric values, including DECIMAL, FLOAT, and INTEGER.

The following data types are collectively known as **numeric** data types:

- “ESQL DECIMAL data type” on page 167
- “ESQL FLOAT data type” on page 168
- “ESQL INTEGER data type” on page 168

Notes:

1. INTEGER and DECIMAL types are represented exactly inside the broker; FLOAT types are inherently subject to rounding error without warning. Do not use FLOAT if you need absolute accuracy, for example, to represent money.
2. Various casts are possible between different numeric types. These can result in loss of precision, if exact types are cast into FLOAT.

For information about numeric functions see “ESQL numeric functions” on page 288.

ESQL DECIMAL data type

The **DECIMAL** data type holds an exact representation of a decimal number. Decimals have precision, scale, and rounding. Precision is the total number of digits of a number:

- The minimum precision is 1
- The maximum precision is 34

Scale is the number of digits to the right of the decimal point:

- The minimum scale (-exponent) is -999,999,999
- The maximum scale (-exponent) is +999,999,999

You cannot define precision and scale when declaring a **DECIMAL**, because they are assigned automatically. It is only possible to specify precision and scale when casting to a **DECIMAL**.

Scale, precision, and rounding:

The following scale, precision, and rounding rules apply:

- Unless rounding is required to keep within the maximum precision, the scale of the result of an addition or subtraction is the greater of the scales of the two operands.
- Unless rounding is required to keep within the maximum precision, the scale of the result of a multiplication is the sum of the scales of the two operands.
- The precision of the result of a division is the smaller of the number of digits needed to represent the result exactly and the maximum precision.
- All addition, subtraction, multiplication, and division calculations round the least significant digits, as necessary, to stay within the maximum precision
- All automatic rounding is *banker's* or *half even symmetric* rounding. The rules of this are:
 - When the first dropped digit is 4 or less, the first retained digit is unchanged
 - When the first dropped digit is 6 or more, the first retained digit is incremented
 - When the first dropped digit is 5, the first retained digit is incremented if it is odd, and unchanged if it is even. Therefore, both 1.5 and 2.5 round to 2 while 3.5 and 4.5 both round to 4
 - Negative numbers are rounded according to the same rule

Decimal literals:

Decimal literals that consist of an unquoted string of digits only, that is, that contain neither a decimal point nor an exponent (for example 12345) are of type **INTEGER** if they are small enough to be represented as integers. Otherwise they are of type **DECIMAL**.

Decimal literals that consist of an unquoted string of digits, optionally a decimal point, and an exponent (for example 123e1), are of type **FLOAT** if they are small enough to be represented as floats. Otherwise they are of type **DECIMAL**.

Decimal literals that consist of the keyword DECIMAL and a quoted string of digits, with or without a decimal point and with or without an exponent, are of type DECIMAL, for example, DECIMAL '42', DECIMAL '1.2346789e+203'.

The strings in this type of literal can also have the values:

- 'NAN', not a number
- 'INF', 'INFINITY'
- '+INF', '+INFINITY'
- '-INF', '-INFINITY'
- 'MAX'
- 'MIN'

(in any mixture of case) to denote the corresponding values.

Note, if you do not specify sufficient precision digits, that INF is returned, as shown in the following example:

```
SET VAL [equals char] CAST('123456' AS DECIMAL(3,0))
```

ESQL FLOAT data type

The FLOAT data type holds a 64-bit, base 2, fraction and exponent approximation to a real number. This gives a range of values between +-1.7E-308 and +- 1.7E+308.

Float literals consist of an unquoted string of digits and either a decimal point (for example 123.4) or an exponent (for example 123e4) or both (for example 123.4e5) . They are of type FLOAT if they are small enough to be represented as floats. Otherwise they are of type DECIMAL

Rounding:

When you CAST a FLOAT to an INTEGER, either implicitly or explicitly, the FLOAT is truncated; that is, the numbers after the decimal point are removed and no rounding occurs.

When you CAST a FLOAT to a DECIMAL or CHARACTER, either implicitly or explicitly, the FLOAT can be rounded to a maximum precision of 15 digits.

ESQL INTEGER data type

The INTEGER data type holds an integer number in 64-bit two's complement form. This gives a range of values between -9223372036854775808 and +9223372036854775807.

Integer literals consist of an unquoted string of digits only; that is, they contain neither a decimal point nor an exponent; for example, 12345. They are of type INTEGER if they are small enough to be represented as integers. Otherwise they are of type DECIMAL.

In addition to this format, you can write integer literals in hexadecimal notation; for example, 0x1234abcd. You can write the hexadecimal letters A to F, and the "x" after the initial zero, in uppercase or lowercase. If you use hexadecimal format, the number must be small enough to fit into an integer. (That is, it cannot be a decimal.)

ESQL REFERENCE data type

The REFERENCE data type holds the location of a field in a message. It cannot hold the location of a constant, a database table, a database column, or another reference.

For compatibility with earlier versions, reference variables can also point at scalar variables

A reference literal is an hierarchic path name, consisting of a list of path elements separated by periods. The first element in the list is known as the correlation name, and identifies a reference, row, or scalar variable. Any subsequent elements apply to references to message trees only, and identify field types, names, and indexes within the message tree relative to the field pointed to by the correlation name.

For example:

```
InputRoot.MQMD.Priority
```

is a field reference literal that refers to the Priority field contained within an MQMD structure within an input message.

ESQL ROW data type

The ROW data type holds a *tree structure*. A row in a database is a particular type of tree structure, but the ROW data type is not restricted to holding data from database rows.

In a database, a row is a fixed, ordered, set of scalar values. A *scalar* is defined as a single entity value or a string.

A database table is an unordered set of rows and therefore represents a two dimensional "array" of scalar values, in which one dimension is fixed and the other is variable. In ESQL, a row is an open-ended, ordered, set of named values in which each value can be scalar, or another row. That is, a row is an open-ended tree structure with no restrictions on dimensions or regularity. Consider the following diagram:

```
Root
  Row
    PartNumber = 1
    Description = 'Chocolate bar'
    Price      = 0.30
  Row
    PartNumber = 2
    Description = 'Biscuit'
    Price      = 0.35
  Row
    PartNumber = 3
    Description = 'Fruit'
    Price      = 0.42
```

In the example, *Root* contains three elements all named "Row". Each of these elements in turn contains three elements with different names and values. This diagram equally describes an instance of an ESQL row data type (that is, a tree structure) or the contents of a database table.

ROW and LIST

The ROW data type is a normal data type. You can use the DECLARE statement to create ROW variables in the same way as you create INTEGER or CHARACTER variables. There is also a more general concept of a ROW data type. In the previous example, *Root* is the root element of a ROW variable. Each of the elements called “Row”, while not the root element of ROW variables, is a root element of substructures. Many ESQL operations (and particularly the SELECT function) work with the general concept of ROW and operate equally on whole trees or parts of trees.

There is also a general concept of a LIST data type. The set of elements called “Row” can be regarded as a list. Some ESQL operations (particularly SELECT) work with the general concept of list.

InputRoot, *OutputRoot*, and so on, are examples of ROW variables that are automatically declared and present in the data structure, ready for use.

ESQL string data types

ESQL supports several data types that handle string values, including BIT, BLOB, and CHARACTER.

The following data types are collectively known as **string** data types:

- “ESQL BIT data type”
- “ESQL BLOB data type”
- “ESQL CHARACTER data type” on page 171

For information about string functions, see “ESQL string manipulation functions” on page 303.

ESQL BIT data type

The BIT data type holds a variable length string of binary digits. It is commonly used to represent arbitrary binary data that does not contain an exact number of bytes. A bit string literal consists of the letter B, followed by a string of binary digits enclosed in single quotation marks, as in the following example:

```
B'0100101001'
```

Any number of digits, which must be either 0 or 1, can be specified. The initial B can be specified in uppercase or lowercase.

ESQL BLOB data type

The BLOB data type holds a variable length string of 8-bit bytes. It is commonly used to represent arbitrary binary data. A BLOB literal consists of the letter X, followed by a string of hexadecimal digits enclosed in single quotation marks, as in the following example:

```
X'0123456789ABCDEF'
```

There must be an even number of digits in the string, because two digits are required to define each byte. Each digit can be one of the hexadecimal digits 0-9 and A-F. Both the initial X and the hexadecimal letters can be specified in uppercase or lowercase.

ESQL CHARACTER data type

The character data type holds a variable length string of Unicode characters. A character string literal consists of any number of characters in single quotation marks. If you want to include a single quotation mark within a character string literal, use another single quotation mark as an escape character.

For example, the assignment SET X='he''was'' puts the value he'was' into X.

ESQL-to-Java data-type mapping table

Table summarizing the mappings from ESQL to Java.

The following table summarizes the mappings from ESQL to Java.

Notes:

- Only the Java scalar wrappers are passed to Java.
- The ESQL scalar types are mapped to Java data types as object wrappers, or object wrapper arrays, depending upon the direction of the procedure parameter. Each wrapper array contains exactly one element.
- Scalar object wrappers are used to allow NULL values to be passed to and from Java methods.

ESQL data types ¹	Java IN data types	Java INOUT and OUT data types
INTEGER, INT	java.lang.Long	java.lang.Long []
FLOAT	java.lang.Double	java.lang.Double[]
DECIMAL	java.math.BigDecimal	java.math.BigDecimal[]
CHARACTER, CHAR	java.lang.String	java.lang.String[]
BLOB	byte[]	byte[][]
BIT	java.util.BitSet	java.util.BitSet[]
DATE	com.ibm.broker.plugin.MbDate	com.ibm.broker.plugin.MbDate[]
TIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
GMTTIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
TIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
GMTTIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
INTERVAL	Not supported	Not supported
BOOLEAN	java.lang.Boolean	java.lang.Boolean[]
REFERENCE (to a message tree) ^{3 4} _{5 6}	com.ibm.broker.plugin.MbElement	com.ibm.broker.plugin.MbElement[] (Supported for INOUT. Not supported for OUT)
ROW	Not supported	Not supported
LIST	Not supported	Not supported

1. Variables that are declared to be CONSTANT (or references to variables that are declared to be CONSTANT) are not allowed to have the direction INOUT or OUT.
2. The time zone set in the Java variable is not important; you obtain the required time zone in the output ESQL.
3. The reference parameter cannot be NULL when passed into a Java method.
4. The reference cannot have the direction OUT when passed into a Java method.

5. If an *MbElement* is passed back from Java to ESQL as an INOUT parameter, it must point to a location in the same message tree as that pointed to by the *MbElement* that was passed into the called Java method.
For example, if an ESQL reference to `OutputRoot.XML.Test` is passed into a Java method as an INOUT *MbElement*, but a different *MbElement* is passed back to ESQL when the call returns, the different element must also point to somewhere in the `OutputRoot` tree.
6. An *MbElement* cannot be returned from a Java method with the RETURNS clause, because no ESQL routine can return a reference. However, an *MbElement* can be returned as an INOUT direction parameter, subject to the conditions described in point 5.

A REFERENCE to a scalar variable can be used in the CALL of a Java method, provided that the data type of the variable to which the reference refers matches the corresponding data type in the Java program signature.

ESQL-to-XPath mapping table

The following table summarizes the mappings from ESQL to XPath.

ESQL	XPath 1.0	XPath 1.0 usage notes
BOOLEAN data types True False Unknown	True() False() No equivalent	Equivalent to Boolean "1" or "True" Equivalent to Boolean "0" or "False"
Date Time data types	No equivalent	
NULL data type	No equivalent	
Numeric data types DECIMAL FLOAT INTEGER	12678967.543233 with or without quotation marks 1.7976931348623158 with or without quotation marks 9223372036854775807 with or without quotation marks	Cannot express an exponent or a leading plus sign. Cannot express an exponent or a leading plus sign. Cannot express an exponent or a leading plus sign.
REFERENCE data type	FilterExpression '/' RelativeLocationPath	For example, <code>\$InputRoot/MQMD/ Priority</code>
String data types BIT BLOB CHARACTER	No equivalent No equivalent Literal	For example 'a "b"' or "a 'b'"
NAME	\$NAME	Can assign such a variable any valid value, of type Boolean, number, or string.
Simple comparison operators > < >= '=' <>	> < >= '=' !=	
Complex comparison operators	No equivalent	

Logical operators AND OR NOT	and or not (operand)	The not function returns true if its argument is false, and false otherwise.
Numeric operators Unary - + - * /	- Unary expression + - * div	Multiplication operator
String operator	No equivalent	
Date time functions	No equivalent	
Numeric functions FLOOR CEIL and CEILING ROUND	floor (number) ceiling (number) No equivalent	
String manipulation functions SUBSTRING TRANSLATE	substring(string, number, number) translate(string, string, string)	

XPath property editors

The XPath files are supplied in three property editors located in the `com.ibm.etools.mft.ibmnodes` plugin. The property editors are:

Read only

Located in
`com.ibm.etools.mft.ibmnodes.editors.xpath.XPathReadOnlyPropertyEditor`

Read write

Located in
`com.ibm.etools.mft.ibmnodes.editors.xpath.XPathReadWritePropertyEditor`

Expression

Located in
`com.ibm.etools.mft.ibmnodes.editors.xpath.XPathPropertyEditor`

For information on adding a property editor to your workspace, see [Adding a property editor or compiler](#).

ESQL variables

ESQL variables can be described as *external variables*, *normal variables*, or *shared variables*; their use is defined in the DECLARE statement.

Types of variable

External

External variables (defined with the EXTERNAL keyword) are also known as *user-defined properties* (see “User-defined properties in ESQL” on page 6). They exist for the entire lifetime of a message flow and are visible to all messages that pass through the flow. You can define *external variables* only at the module and schema level. You can modify their initial values (optionally set by the DECLARE statement) by using the Message Flow editor, or at deployment time, by using the Broker Archive editor. You can

query and set the values of user-defined properties at run time by using the Configuration Manager Proxy (CMP). For more information, see [Setting user-defined properties at run time in a CMP application](#).

Normal

Normal variables have a lifetime of just one message passing through a node. They are visible to that message only. To define a *normal variable*, omit both the EXTERNAL and SHARED keywords.

Shared

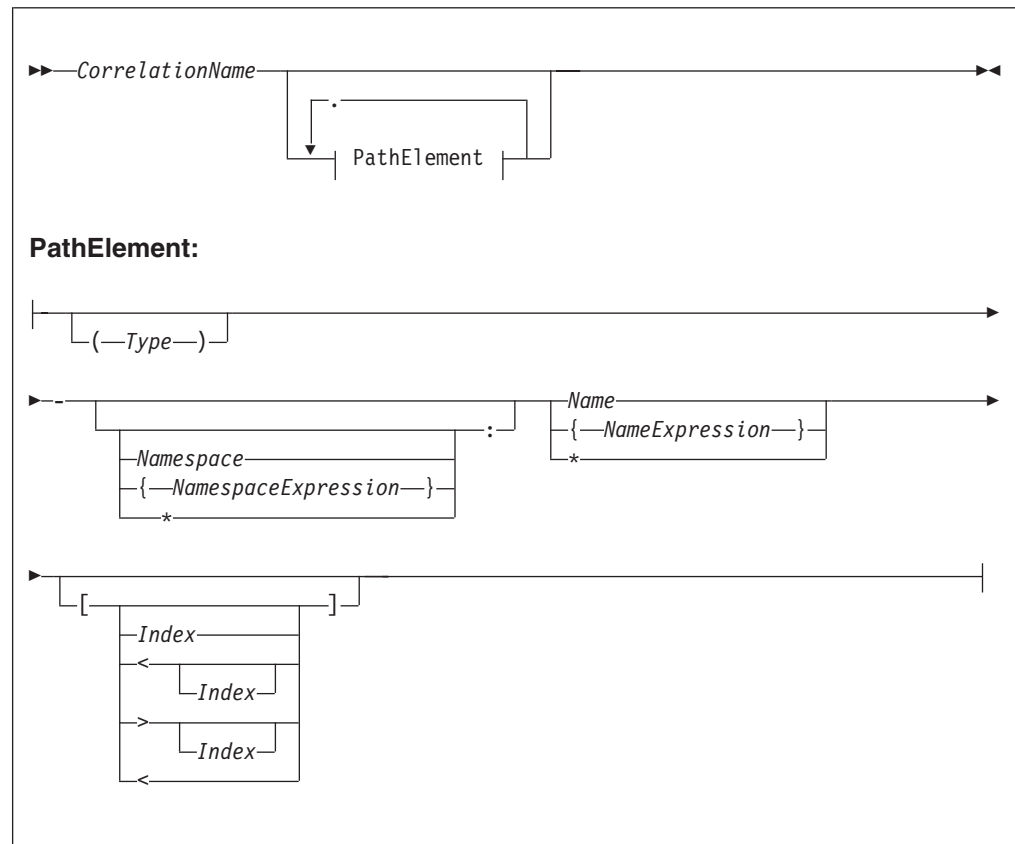
Shared variables (defined with the SHARED keyword) can be used to implement an in-memory cache in the message flow; see [Optimizing message flow response times](#). *Shared variables* have a long lifetime and are visible to multiple messages that pass through the flow; see [“Long-lived variables”](#) on page 7. They exist for the lifetime of the execution group process, the lifetime of the flow or node, or the lifetime of the node's SQL that declares the variable (whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker startup.

See also the ATOMIC option of the [“BEGIN ... END statement”](#) on page 191. The BEGIN ATOMIC construct is useful when a number of changes have to be made to a shared variable and when it is important to prevent other instances seeing the intermediate states of the data.

ESQL field reference overview

You can use ESQL field references to form paths to message body elements.

The full syntax for field references is as shown below:



A field reference consists of a correlation name, followed by zero or more Path Elements separated by periods (.). The correlation name identifies a well-known starting point and must be the name of a constant, a declared variable (scalar, row or reference), or one of the predefined start points; for example, InputRoot. The path Fields define a path from the start point to the desired field.

See:

- “Namespace” on page 176 for the meaning of the different combinations of namespace and name
- “Target field references” on page 180 for the meaning of the different combinations of field references
- “Index” on page 177 for the meaning of the different combinations of index clauses
- “Type” on page 178 for the meaning of the different combinations of types

For example:

InputRoot.XMLNS.Data.Invoice

starts the broker at the location InputRoot (that is, the root of the input message to a Compute node) and then performs a sequence of navigations. First, it navigates from root to the first child field called XMLNS, then to the first child field of the XMLNS field called Data. Finally, the broker navigates to the first child field of the Data field called Invoice. Whenever this field reference occurs in an ESQ program, the invoice field is accessed.

This form of field reference is simple, convenient, and is the most commonly used. However, it does have two limitations:

- Because the names used must be valid ESQL identifiers, you can use only names that conform to the rules of ESQL. That is, the names can contain only alphanumeric characters including underscore, the first character cannot be numeric, and names must be at least one character long. You can avoid these limitations by enclosing names not conforming to these rules in double quotation marks. For example:

```
InputRoot.XMLNS."Customer Data".Invoice
```

If you need to refer to fields that contain quotation marks, use two pairs of quotation marks around the reference. For example:

```
Body.Message."""hello"""
```

Some identifiers are reserved as keywords but, with the exception of the correlation name, you can use them in field references without the use of double quotation marks

- Because the names of the fields appear in the ESQL program, they must be known when the program is written. This limitation can be avoided by using the alternative syntax that uses braces ({ ... }). This syntax allows you to use any expression that returns a non-null value of type character.

For example:

```
InputRoot.XMLNS."Customer Data".{'Customer-' ||  
CurrentCustomer}.Invoice
```

in which the invoices are contained in a folder with a name is formed by concatenating the character literal Customer- with the value in CurrentCustomer (which in this example must be a declared variable of type character).

You can use the asterisk (*) wildcard character in a path element to match any name. You can also use "*" to specify a partial name. For example, Prefix* matches any name that begins with "Prefix".

Note that enclosing anything in double quotation marks in ESQL makes it an identifier; enclosing anything in single quotation marks makes it a character literal. You must enclose all character strings in single quotation marks.

Namespace

Field names can belong to namespaces. Field references provide support for namespaces as follows:

- Each field of each field reference that contains a name clause can also contain a namespace clause defining the namespace to which the specified name belongs.
- Each namespace name can be defined by either a simple identifier or by an expression (enclosed in curly braces). If an identifier is the name of a declared namespace constant, the value of the constant is used. If an expression is used, it must return a non-null value of type character.
- A namespace clause of * explicitly states that namespace information is to be ignored when locating Fields in a tree.
- A namespace clause consisting of only : explicitly targets the notarget namespace. The clause has no identifier, expression or wildcard (*).

For example:


```

DECLARE sp1 NAMESPACE 'http://www.ibm.com/spacel';

/* Namespace declaration to associate prefix 'spacel' with the namespace */

SET OutputRoot.XMLNS.TestCase.(XML.NamespaceDecl)xmlns:spacel = 'http://www.ibm.com/spacel';
SET OutputRoot.XMLNS.TestCase.{sp1}:data1 = 'Hello!';

```

generates:

```

<TestCase xmlns:spacel="http://www.ibm.com/spacel">
<spacel:data1>Hello!</spacel:data1>
</TestCase>

```

Index

Each field of a field reference can contain an index clause. This clause is denoted by brackets ([...]) and accepts any expression that returns a non-null value of type integer. This clause identifies which of several fields with the same name is to be selected. Fields are numbered from the first, starting at one. If this clause is not present, it is assumed that the first field is required. Thus, the two examples below have exactly the same meaning:

```

InputRoot.XMLNS.Data[1].Invoice
InputRoot.XMLNS.Data.Invoice[1]

```

This construct is most commonly used with an index variable, so that a loop steps through all such fields in sequence. For example:

```

WHILE count < 32 DO
    SET TOTAL = TOTAL + InputRoot.XMLNS.Data.Invoice[count].Amount;
    SET COUNT = COUNT + 1
END WHILE;

```

Use this kind of construct with care, because it implies that the broker must count the fields from the beginning each time round the loop. If the repeat count is large, performance will be poor. In such cases, a better alternative is to use a field reference variable.

Index expressions can optionally be preceded by a less-than sign (<), indicating that the required field is to be indexed from the last field, not the first. In this case, the index 1 refers to the last field and the index 2 refers to the penultimate field. For completeness, you can use a greater-than sign to indicate counting from the first field. The example below shows ESQL code that handles indexes where there are four fields called Invoice.

```

InputRoot.XMLNS.Data.Invoice      -- Selects the first
InputRoot.XMLNS.Data.Invoice[1]   -- Selects the first
InputRoot.XMLNS.Data.Invoice[>]   -- Selects the first
InputRoot.XMLNS.Data.Invoice[>1]  -- Selects the first
InputRoot.XMLNS.Data.Invoice[>2]  -- Selects the second
InputRoot.XMLNS.Data.Invoice[<]   -- Selects the fourth
InputRoot.XMLNS.Data.Invoice[<1]  -- Selects the fourth
InputRoot.XMLNS.Data.Invoice[<2]  -- Selects the third
InputRoot.XMLNS.Data.Invoice[<3]  -- Selects the second

```

An index clause can also consist of an empty pair of brackets ([]). This selects all fields with matching names. Use this construct with functions and statements that expect lists (for example, the SELECT, CARDINALITY, SINGULAR, and EXISTS functions, or the SET statement) .

Type

Each field of a field reference can contain a type clause. These are denoted by parentheses (()), and accept any expression that returns a non-null value of type integer. The presence of a type expression restricts the fields that are selected to those of the matching type. This construct is most commonly used with generic XML, where there are many field types and it is possible for one XML field to contain both attributes and further XML Fields with the same name.

For example:

```
<Item Value = '1234'>
  <Value>5678</Value>
</Item>
```

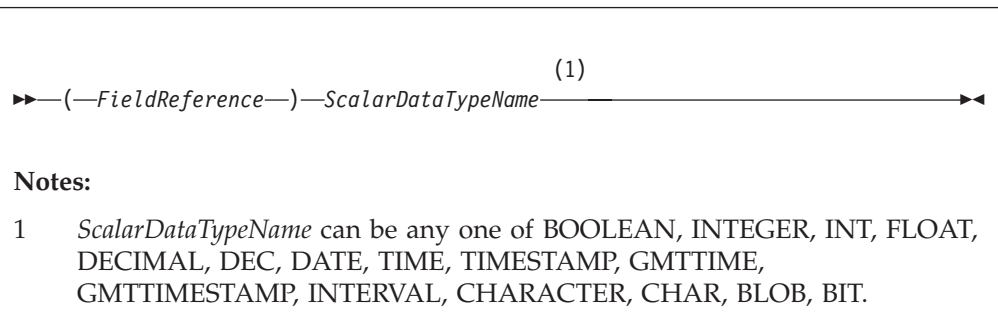
Here, the XML field Item has two child Fields, both called “Value”. The child Fields can be distinguished by using type clauses:

Item.(<Domain>.Attribute)Value to select the attribute, and

Item.(XML.Element)Value to select the field, where <Domain> is one of XML, XMLNS, or XMLNSC, as determined by the message domain of the source.

Type constraints

A type constraint checks the data type returned by a field reference.



Typically, a type constraint causes the scalar value of the reference to be extracted (in a similar way to the FIELDVALUE function) and an exception to be thrown if the reference is not of the correct type. By definition, an exception will be thrown for all nonexistent fields, because these evaluate to NULL. This provides a convenient and fast way of causing exceptions if essential fields are missing from messages.

However, when type constraints occur in expressions that are candidates for being passed to a database (for example, they are in a WHERE clause), the information is used to determine whether the expression can be given to the database. This can be important if a WHERE clause contains a CAST operating on a database table column. In the absence of a type constraint, such expressions cannot be given to the database because the broker cannot tell whether the database is capable of performing the required conversion. Note, however, that you should always exercise caution when using casts operating on column values, because some databases have exceedingly limited data conversion capabilities.

Field references summary

`*`, `*[..]`, `(..)*`, `(..)*[..]`

None of these forms specifies a name or namespace. The target field can have any name, in any namespace or in no namespace. It is located solely by its type, its index, or its type and index, as appropriate.

`Name`, `Name[..]`, `(..)Name`, `(..)Name[..]`

All these forms specify a name but no namespace. The target field is located by namespace and name, and also by type and index where appropriate.

The namespace is taken to be the only namespace in the namespace path containing this name. The only namespace that can be in the path is the notarget namespace.

These forms all existed before namespaces were introduced. Although their behavior has changed in that they now compare both name and namespace, existing transforms should see no change in their behavior because all existing transforms create their Fields in the notarget namespace.

`:*`, `:*[..]`, `(..):*`, `(..):*[..]`

All these forms specify the notarget namespace but no name. The target field is located by its namespace and also by type and index where appropriate.

`:Name`, `:Name[..]`, `(..):Name`, `(..):Name[..]`

All these forms specify a name and the notarget namespace. The target field is located by namespace and name and also by type and index where appropriate.

`*:*`, `*:*[..]`, `(..)*:*`, `(..)*:*[..]`

None of these forms specifies a name or a namespace. Note that `"*:*"` is equivalent to `"**"`, and matches no namespace as well as any namespace. The target field can have any name, in any namespace or in no namespace. It is located solely by its type, its index, or its type and index, as appropriate.

`*:Name`, `*:Name[..]`, `(..)*:Name`, `(..)*:Name[..]`

All these forms specify a name but no namespace. The target field is located by name and also by type and index where appropriate.

`Namespace:*`, `Namespace:*[..]`, `(..)Namespace:*`, `(..)Namespace:*[..]`

All these forms specify a namespace but no name. The target field is located by namespace and also by type and index where appropriate.

`Namespace:Name`, `Namespace:Name[..]`, `(..)Namespace:Name`, `(..)Namespace:Name[..]`

All these forms specify a namespace and name. The target field is located by namespace and name and also by type and index where appropriate.

In all the preceding cases a name, or namespace, provided by an expression contained in braces (`{}`) is equivalent to a name provided as an identifier.

By definition, the name of the notarget namespace is the empty string. The empty string can be selected by expressions which evaluate to the empty string, the empty identifier `""`, or by reference to a namespace constant defined as the empty string.

Target field references

The use of field references usually implies searching for an existing field. However, if the required field does not exist, as is usually the case for field references that are the targets of SET statements and those in the AS clauses of SELECT functions, it is created.

In these situations, there are a variety of circumstances in which the broker cannot tell what the required name or namespace is, and in these situations the following general principles apply :

- If the name clause is absent or does not specify a name, and the namespace clause is absent or does not specify or imply a namespace (that is, there is no name or namespace available), one of the following conditions applies:
 - If the assignment algorithm does **not** copy the name from some existing field, the new field has both its name and namespace set to the empty string and its name flag is **not** set automatically.
In the absence of a type specification, the field's type is not *Name* or *NameValue*, which effectively indicates that the new field is nameless.
 - Otherwise, if the assignment algorithm copies the name from some existing field, the new field has both its name and namespace copied from the existing field and its *Name* flag is set automatically
- If the name clause is present and specifies a name, but the namespace clause is absent or does not specify or imply a namespace (that is, a name is available but a namespace is not), the new field has its:
 - *Name* set to the given value
 - *Namespace* set to the empty string
 - *Name* flag set automatically
- If the name clause is absent or does not specify a name, but the namespace clause is present and specifies or implies a namespace (that is, a namespace is available but a name is not), the new field has its:
 - *Namespace* set to the given value
 - *Name* set to the empty string
 - *Name* flag set automatically
- If the name clause is present and specifies a name, and the namespace clause is present and specifies or implies a namespace, the new field has its:
 - *Name* set to the given value
 - *Namespace* set to the given value
 - *Name* flag set automatically

There are also cases where the broker creates Fields in addition to those referenced by field references:

- Tree copy: new Fields are created by an algorithm that uses a source tree as a template. If the algorithm copies the name of a source field to a new field, its namespace is copied as well.
- Anonymous select expressions: SELECT clauses are not obliged to have AS clauses; those that do not have them, set the names of the newly created Fields to default values (see “SELECT function” on page 345).

These defaults can be derived from field names, column names or can simply be manufactured sequence names. If the name is an field name, this is effectively a tree copy, and the namespace name is copied as above.

Otherwise, the namespace of the newly-created field is derived by searching the path, that is, the name is be treated as the *NameId* syntax of a field reference.

The effect of setting a field to NULL

Take care when assigning a null value to a field. For example, the following command *deletes* the Name field:

```
SET OutputRoot.XMLNS.Msg.Data.Name = NULL; -- this deletes the field
```

The correct way to assign a null value to a field is as follows:

```
SET OutputRoot.XMLNS.Msg.Data.Name VALUE = NULL;  
-- this assigns a NULL value to a field without deleting it
```

Note: to users on compatibility with earlier versions

For compatibility with earlier versions, the LAST keyword is still supported, but its use is deprecated. LAST cannot be used as part of an index expression: [LAST] is valid, and is equivalent to [<], but [LAST3] is not valid.

The LAST keyword has been replaced by the following arrow syntax, which allows both a direction of search and index to be specified:

```
Field [ > ]                -- The first field, equivalent to [ 1 ]  
Field [ > ( a + b ) * 2 ]  
Field [ < ]                -- The last field, equivalent to [ LAST ]  
Field [ < 1 ]              -- The last field, equivalent to [ LAST ]  
Field [ < 2 ]              -- The last but one field  
Field [ < ( a + b ) / 3 ]
```

ESQL operators

This section provides reference information for the following groups of operators, and for the rules for precedence:

- Simple comparison operators
- Complex comparison operators
- Logical operators
- Numeric operators
- String operator
- Rules for operator precedence

ESQL simple comparison operators

This topic describes ESQL's simple comparison operators. For information about ESQL's complex comparison operators, see "ESQL complex comparison operators" on page 182.

ESQL provides a full set of comparison operators (predicates). Each compares two scalar values and returns a Boolean. If either operand is null the result is null. Otherwise the result is true if the condition is satisfied and false if it is not.

Comparison operators can be applied to all scalar data types. However, if the two operands are of different types, special rules apply. These are described in "Implicit casts" on page 365.

Some comparison operators also support the comparison of rows and lists. These are noted below.

Operator >

The first operand is greater than the second.

Operator <

The first operand is less than the second.

Operator >=

The first operand is greater than or equal to the second.

Operator <=

The first operand is less than or equal to the second.

Operator =

The first operand is equal to that of the second.

This operator can also compare rows and lists. See “ROW and LIST comparisons” on page 354 for a description of list and row comparison.

Operator <>

The first operand is not equal to the second.

This operator can also compare rows and lists. See “ROW and LIST comparisons” on page 354 for a description of list and row comparison.

The meanings of “equal”, “less”, and “greater” in this context are as follows:

- For the numeric types (INTEGER, FLOAT, DECIMAL) the numeric values are compared. Thus 4.2 is greater than 2.4 and -2.4 is greater than -4.2.
- For the date/time types (DATE, TIME, TIMESTAMP, GMTTIME, GMTTIMESTAMP but not INTERVAL) a later point in time is regarded as being greater than an earlier point in time. Thus the date 2004-03-31 is greater than the date 1947-10-24.
- For the INTERVAL type, a larger interval of time is regarded as being greater than a smaller interval of time.

For the string types (CHARACTER, BLOB, BIT) the comparison is lexicographic. Starting from the left, the individual elements (each character, byte or bit) are compared. If no difference is found, the strings are equal. If a difference is found, the values are greater if the first different element in the first operand is greater than the corresponding element in the second and less if they are less. In the special case where two strings are of unequal length but equal as far as they go, the longer string is regarded as being greater than the shorter. Thus:

'ABD' is greater than 'ABC'
'ABC' is greater than 'AB'

Trailing blanks are regarded as insignificant in character comparisons. Thus if you want to ensure that two strings are truly equal you need to compare both the strings themselves and their lengths. For example:

'ABC ' is equal to 'ABC'

Note that comparing strings with a length of one is equivalent to comparing individual characters, bytes, or bits. Because ESQL has no single character, byte, or bit data types, it is standard practice to use strings of length one to compare single characters, bytes, or bits.

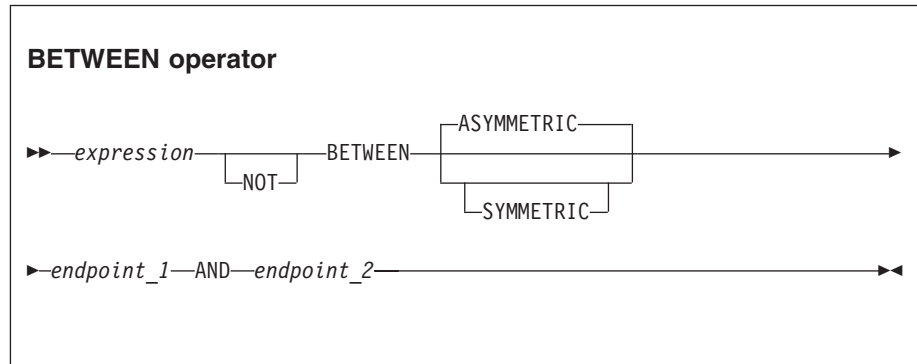
ESQL complex comparison operators

ESQL supports several operators for complex comparison tasks.

If you want to use ESQL to perform a simple comparison, see “ESQL simple comparison operators” on page 181.

Operator BETWEEN

Use the operator BETWEEN to test whether a value lies between two boundary values.



This operator exists in two forms, SYMMETRIC and ASYMMETRIC (which is the default if neither is specified). The SYMMETRIC form is equivalent to:

(source >= boundary1 AND source <= boundary2) OR
 (source >= boundary2 AND source <= boundary1)

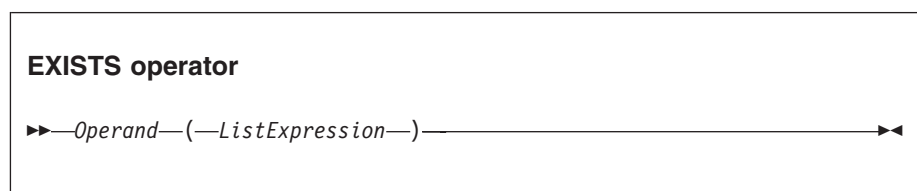
The ASYMMETRIC form is equivalent to:

source >= boundary1 AND source <= boundary2

The ASYMMETRIC form is simpler but returns only the result that you expect when the first boundary value has a smaller value than the second boundary. It is useful only when the boundary condition expressions are literals.

If the operands are of different types, special rules apply. These rules are described in “Implicit casts” on page 365.

Operator EXISTS



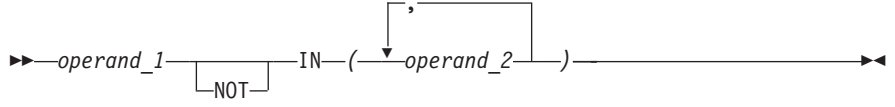
The operator EXISTS returns a Boolean value that indicates whether a SELECT function returned one or more values (TRUE) or none (FALSE).

EXISTS(SELECT * FROM something WHERE predicate)

Operator IN

Use the operator IN to test whether a value is equal to one of a list of values.

IN operator

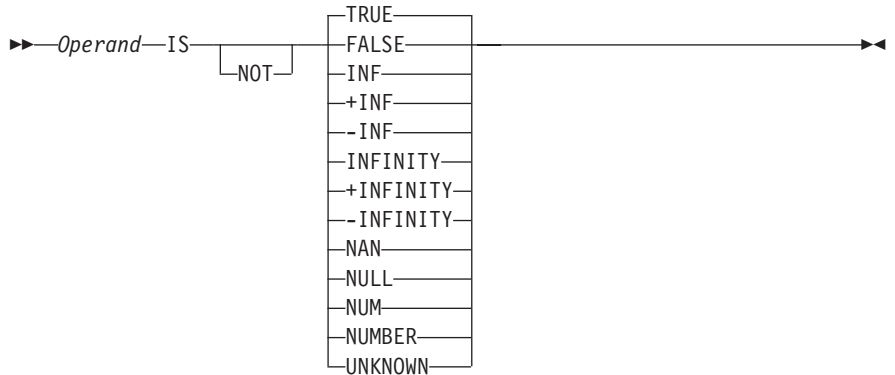


The result is TRUE if the left operand is not NULL and is equal to one of the right operands. The result is FALSE if the left operand is not NULL, and is not equal to one or more of the right operands, none of which have NULL values. Otherwise the result is UNKNOWN. If the operands are of different types, special rules apply. These rules are described in “Implicit casts” on page 365.

Operator IS

Use the operator IS to test whether an expression has returned a special value.

IS operator



The primary purpose of the operator IS is to test whether a value is NULL. You cannot use the comparison operator (=) to test for a NULL value, because the result of comparing any value with NULL is NULL.

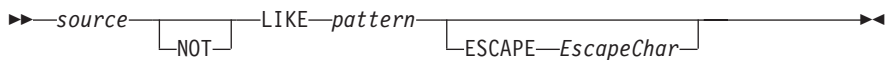
You can also use the IS operand to test for the Boolean values TRUE and FALSE, and the testing of decimal values for special values. These values are denoted by INF, +INF, -INF, NAN (not a number), and NUM (a valid number) in upper, lower, or mixed case. The alternative forms +INFINITY, -INFINITY, and NUMBER are also accepted.

If applied to non-numeric types, the result is FALSE.

Operator LIKE

Use the operator LIKE to search for strings that match a certain pattern.

LIKE operator



The result is TRUE if none of the operands is NULL and the *source* operand matches the pattern operand. The result is FALSE if none of the operands is NULL and the *source* operand does not match the pattern operand. Otherwise the result is UNKNOWN.

The pattern is specified by a string in which the percent (%) and underscore (_) characters have a special meaning:

- The underscore character _ matches a single character.
For example, the following finds matches for IBM and for IGI, but not for International Business Machines or IBM Corp:
`Body.Trade.Company LIKE 'I_'`
- The percent character % matches a string of zero or more characters.
For example, the following phrase finds matches for IBM, IGI, International Business Machines, and IBM Corp:
`Body.Trade.Company LIKE 'I%'`

To use the percent and underscore characters within the expressions that are to be matched, precede the characters with an ESCAPE character, which defaults to the backslash (\) character.

For example, the following predicate finds a match for IBM_Corp.
`Body.Trade.Company LIKE 'IBM_Corp'`

You can specify a different escape character by using the ESCAPE clause. For example, you could also specify the previous example in this way:
`Body.Trade.Company LIKE 'IBM$_Corp' ESCAPE '$'`

Operator SINGULAR

SINGULAR operator



The operator SINGULAR returns a Boolean value of TRUE if the list has exactly one element, otherwise it returns FALSE.

ESQL logical operators

ESQL provides the following logical operators:

Operator AND

The result is the logical AND of the two operands. Both operands must be Boolean values.

Operator OR

The result is the logical OR of the two operands. Both operands must be Boolean values.

Operator NOT

The result is the logical NOT of the operand, which must be a Boolean value.

NULL and UNKNOWN values are treated as special values by these operators. The principles are:

- NULL and UNKNOWN are treated the same.
- If an operand is NULL the result is NULL unless the operation result is already dictated by the other parameter.

The result of AND and OR operations is defined by the following table.

Value of P	Value of Q	Result of P AND Q	Result of P OR Q
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	UNKNOWN	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE
UNKNOWN	FALSE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

The result of NOT operations is defined by the following table.

Operand	Result of NOT
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

ESQL numeric operators

ESQL provides the following numeric operators:

Unary Operator -

The result is the negation of the operand (that is, it has the same magnitude as the operand but the opposite sign). You can negate numeric values (INTEGER, DECIMAL and FLOAT) and intervals (INTERVAL).

Operator +

The result is the sum of the two operands. You can add two numeric values, two intervals, and an interval to a datetime value (DATE, TIME, TIMESTAMP, GMTTIME, and GMTTIMESTAMP).

Operator -

The result is the difference between the two operands. It is possible to:

- Subtract one numeric value from another.
- Subtract one date-time from another. The result is an interval.
- Subtract one interval from another. The result is an interval.
- Subtract an interval from a datetime value. The result is a date-time.

When subtracting one date-time from another, you must indicate the type of interval required. You do this by using a qualifier consisting of parentheses enclosing the expression, followed by an interval qualifier. For example:

```
SET OutputRoot.XMLNS.Data.Age =  
    (DATE '2005-03-31' - DATE '1947-10-24') YEAR TO MONTH;
```

Operator *

The result is the product of the two operands. You can multiply numeric values and multiply an interval by a numeric value.

Operator /

The result is the dividend of the two operands. You can divide numeric values and divide an interval by a numeric value.

Operator ||

The result is the concatenation of the two operands. You can concatenate string values (CHARACTER, BIT, and BLOB).

In all cases, if either operand is NULL, the result is NULL. If the operands are of different types, special rules apply. These are described in “Implicit casts” on page 365.

For examples of how you can use these operators to manipulate datetime values, see “Using numeric operators with datetime values” on page 40.

ESQL string operator

A single string operator, concatenation, is supported in ESQL.

Operator ||

The result is the concatenation of the two operands. You can concatenate string values (CHARACTER, BIT, and BLOB).

If either operand is NULL, the result is NULL.

Rules for ESQL operator precedence

When an expression involves more than one operator, the order in which the expression is evaluated might affect the result. Consider the following example:

```
SET a = b + c * d;
```

Under ESQL's precedence rules, *c* is multiplied by *d* and the result is added to *b*. This rule states that multiplication takes precedence over addition, so reordering the expression as follows:

```
SET a = c * d + b;
```

makes no difference. ESQL's precedence rules are set out below but it is generally considered good practice to use parentheses to make the meaning clear. The order of precedence is:

1. Parentheses
2. Unary operators including unary - and NOT
3. Multiplication and division
4. Concatenation
5. Addition and subtraction

Operations at the same level are evaluated from left to right.

ESQL statements

You can use ESQL statements to manipulate message trees, update databases, or interact with nodes.

The following table summarizes the ESQL statements and what they do.

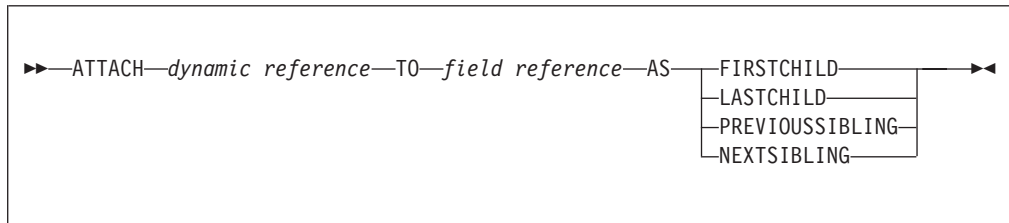
Statement type	Description
Basic statements:	
“BEGIN ... END statement” on page 191	Gives the statements defined within the BEGIN and END keywords the status of a single statement.
“CALL statement” on page 196	Invokes a user-written routine that has been defined using a CREATE FUNCTION or CREATE PROCEDURE statement.
“CASE statement” on page 199	Uses rules defined in WHEN clauses to select a block of statements to execute.
“CREATE FUNCTION statement” on page 209	Like CREATE PROCEDURE, CREATE FUNCTION defines a user-written routine. (The few differences between CREATE FUNCTION and CREATE ROUTINE are described in the reference material.)
“CREATE MODULE statement” on page 218	Creates a module (a named container associated with a node).
“CREATE PROCEDURE statement” on page 220	Like CREATE FUNCTION, CREATE PROCEDURE defines a user-written routine. (The few differences between CREATE FUNCTION and CREATE ROUTINE are described in the reference material.)
“DECLARE statement” on page 235	Declares one or more variables that can be used to store temporary values.
“IF statement” on page 248	Processes a set of statements based on the result of evaluating condition expressions.
“ITERATE statement” on page 253	Abandons processing the current iteration of the containing WHILE, REPEAT, LOOP, or BEGIN statement, and might start the next iteration.
“LEAVE statement” on page 254	Abandons processing the current iteration of the containing WHILE, REPEAT, LOOP or BEGIN statement, and stops looping.
“LOOP statement” on page 256	Processes a sequence of statements repeatedly and unconditionally.
“REPEAT statement” on page 264	Processes a sequence of statements, then evaluates a condition expression. If the expression evaluates to TRUE, executes the statements again.
“RETURN statement” on page 266	Stops processing the current function or procedure and passes control back to the caller.
“SET statement” on page 268	Evaluates a source expression, and assigns the result to the target entity.

Statement type	Description
“THROW statement” on page 270	Generates a user exception.
“WHILE statement” on page 275	Evaluates a condition expression, and if it is TRUE executes a sequence of statements.
Message tree manipulation statements:	
“ATTACH statement”	Attaches a portion of a message tree into a new position in the message hierarchy.
“CREATE statement” on page 201	Creates a new message field.
“DELETE statement” on page 245	Detaches and destroys a portion of a message tree, allowing its memory to be reused.
“DETACH statement” on page 246	Detaches a portion of a message tree without deleting it.
“FOR statement” on page 248	Iterates through a list (for example, a message array).
“MOVE statement” on page 257	Changes the field pointed to by a target reference variable.
Database update statements:	
“DELETE FROM statement” on page 243	Deletes rows from a table in an external database based on a search condition.
“INSERT statement” on page 249	Adds a new row to an external database.
“PASSTHRU statement” on page 259	Takes a character value and passes it as an SQL statement to an external database.
“UPDATE statement” on page 272	Updates the values of specified rows and columns in a table in an external database.
Node interaction statements:	
“PROPAGATE statement” on page 261	Propagates a message to the downstream nodes within the message flow.
Other statements:	
“BROKER SCHEMA statement” on page 193	This statement is optional and is used in an ESQL file to explicitly identify the schema that contains the file.
“DECLARE HANDLER statement” on page 240	Declares an error handler.
“EVAL statement” on page 246	Takes a character value, interprets it as an SQL statement, and executes it.
“LOG statement” on page 255	Writes a record to the event or user trace log.
“RESIGNAL statement” on page 265	Re-throws the current exception (if any). This is used by an error handler, when it cannot handle an exception, to give an error handler in higher scope the opportunity of handling the exception.

ATTACH statement

The ATTACH statement attaches a portion of a message tree into a new position in the message hierarchy.

Syntax



The following example illustrates how to use the ATTACH statement, together with the DETACH statement described in “DETACH statement” on page 246, to modify a message structure. The dynamic reference supplied to the DETACH statement must point to a modifiable message tree such as Environment, LocalEnvironment, OutputRoot, OutputExceptionList, or InputLocalEnvironment.

There are some limitations on the use of ATTACH. In general, elements detached from the output trees of a Compute node are not attached to the environment or to input trees.

For example, if you take the following message:

```
<Data>  
  <Order>  
    <Item>cheese  
      <Type>stilton</Type>  
    </Item>  
    <Item>bread</Item>  
  </Order>  
  <Order>  
    <Item>garlic</Item>  
    <Item>wine</Item>  
  </Order>  
</Data>
```

the following ESQL statements:

```
SET OutputRoot = InputRoot;  
DECLARE ref1 REFERENCE TO OutputRoot.XMLNSC.Data.Order[1].Item[1];  
DETACH ref1;  
ATTACH ref1 TO OutputRoot.XMLNSC.Data.Order[2] AS LASTCHILD;
```

result in the following new message structure:

```
<Data>  
  <Order>  
    <Item>bread</Item>  
  </Order>  
  <Order>  
    <Item>garlic</Item>  
    <Item>wine</Item>  
    <Item>cheese  
      <Type>stilton</Type>  
    </Item>  
  </Order>  
</Data>
```

For information about dynamic references see “Creating dynamic field references” on page 36.

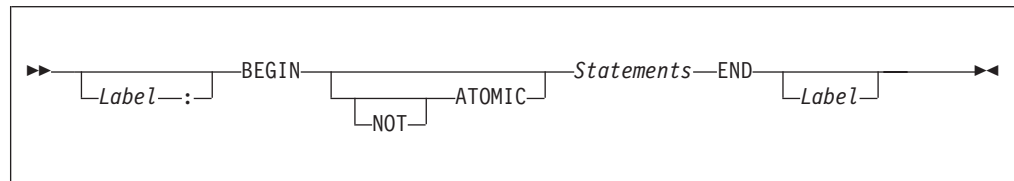
BEGIN ... END statement

The BEGIN ... END statement gives the statements defined within the BEGIN and END keywords the status of a single statement.

This allows the contained statements to:

- Be the body of a function or a procedure
- Have their exceptions handled by a handler
- Have their execution discontinued by a LEAVE statement

Syntax



The second *Label* can be present only if the first *Label* is present. If both labels are present, they must be identical. Two or more labeled statements at the same level can have the same label, but this partly negates the advantage of the second label. The advantage is that the labels unambiguously and accurately match each END with its BEGIN. However, a labeled statement nested within *Statements* cannot have the same label, because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

Scope of variables

A new local variable scope is opened immediately after the opening BEGIN and, therefore, any variables declared within this statement go out of scope when the terminating END is reached. If a local variable has the same name as an existing variable, any references to that name that occur after the declaration access the local variable. For example:

```
DECLARE Variable1 CHAR 'Existing variable';

-- A reference to Variable1 here returns 'Existing variable'

BEGIN
  -- A reference to Variable1 here returns 'Existing variable'

  DECLARE Variable1 CHAR 'Local variable'; -- Perfectly legal even though
the name is the same

  -- A reference to Variable1 here returns 'Local variable'
END;
```

ATOMIC

If ATOMIC is specified, only one instance of a message flow (that is, one thread) is allowed to execute the statements of a specific BEGIN ATOMIC... END statement (identified by its schema and label), at any one time. If no label is present, the behavior is as if a zero length label had been specified.

The BEGIN ATOMIC construct is useful when a number of changes need to be made to a shared variable and it is important to prevent other instances seeing the intermediate states of the data. Consider the following code example:

```
CREATE PROCEDURE WriteSharedVariable1(IN NewValue CHARACTER)
SharedVariableMutex1 : BEGIN ATOMIC
  -- Set new value into shared variable
END;

CREATE FUNCTION ReadSharedVariable1() RETURNS CHARACTER
SharedVariableMutex1 : BEGIN ATOMIC
  DECLARE Value CHARACTER;
  -- Get value from shared variable
  RETURN Value;
END;
```

The last example assumes that the procedure WriteSharedVariable1 and the function ReadSharedVariable1 are in the same schema and are used by nodes within the same flow. However, it does not matter whether or not the procedure and function are contained within modules, or whether they are used within the same or different nodes. The broker ensures that, at any particular time, only one thread is executing any of the statements within the atomic sections. This ensures that, for example, two simultaneous writes or a simultaneous read and write are executed serially. Note that:

- The serialization is limited to the flow. Two flows which use BEGIN ATOMIC... END statements with the same schema and label can be executed simultaneously. In this respect, multiple instances within a flow and multiple copies of a flow are not equivalent.
- The serialization is limited by the schema and label. Atomic BEGIN ... END statements specified in different schemas or with different labels do not interact with each other.

Note: You can look at this in a different way, if you prefer. For each combination of message flow, schema, and label, the broker has a mutex that prevents simultaneous access to the statements associated with that mutex.

Do not nest BEGIN ATOMIC... END statements, either directly or indirectly, because this could lead to “deadly embraces”. For this reason, do not use a PROPAGATE statement from within an atomic block.

It is not necessary to use the BEGIN ATOMIC construct in flows that will never be deployed with more than one instance (but it might be unwise to leave this to chance). It is also unnecessary to use the BEGIN ATOMIC construct on reads and writes to shared variables. The broker always safely writes a new value to, and safely reads the latest value from, a shared variable. ATOMIC is only required when the application is sensitive to seeing intermediate results.

Consider the following example:

```
DECLARE LastOrderDate SHARED DATE;
...
SET LastOrderDate = CURRENT_DATE;
...
SET OutputRoot.XMLNSC.Data.Orders.Order[1].Date = LastOrderDate;
```

Here we assume that one thread is periodically updating LastOrderDate and another is periodically reading it. There is no need to use ATOMIC, because the second SET statement always reads a valid value. If the updating and reading

occur very closely in time, whether the old or new value is read is indeterminate, but it is always one or the other. The result will never be garbage.

But now consider the following example:

```
DECLARE Count SHARED INT;  
...  
SET Count = Count + 1;
```

Here we assume that several threads are periodically executing the SET statement. In this case you do need to use ATOMIC, because two threads might read Count in almost the same instant, and get the same value. Both threads perform the addition and both store the same value back. The end result is thus N+1 and not N+2.

The broker does not automatically provide higher-level locking than this (for example, locking covering the whole SET statement), because such locking is liable to cause “deadly embraces”.

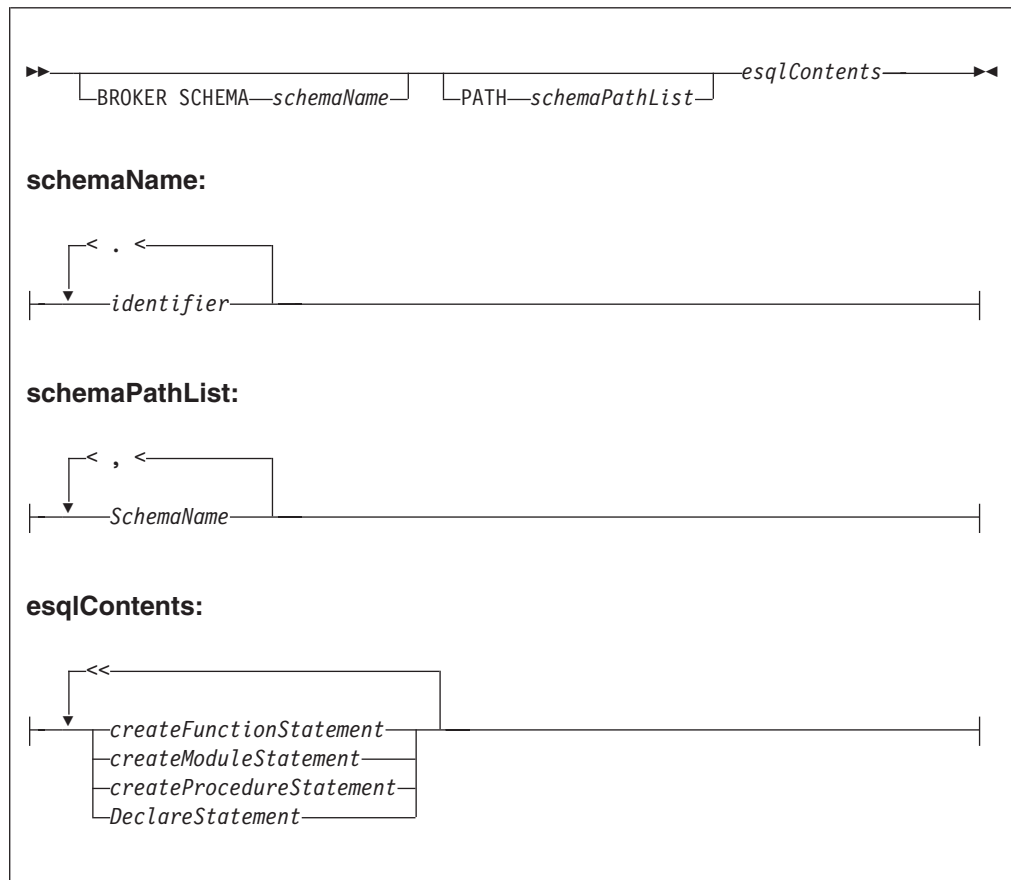
Hint

You can consider the BEGIN ... END statement to be a looping construct, which always loops just once. The effect of an ITERATE or LEAVE statement nested within a BEGIN ... END statement is then as you would expect: control is transferred to the statement following the END. Using ITERATE or LEAVE within a BEGIN ... END statement is useful in cases where there is a long series of computations that needs to be abandoned, either because a definite result has been achieved or because an error has occurred.

BROKER SCHEMA statement

The BROKER SCHEMA statement is optional; use it in an ESQL file to explicitly identify the schema that contains the file.

Syntax



An ESQL schema is a named container for functions, procedures, modules, and variables. ESQL schema is similar to the namespace concept of C++ and XML, and to the package concept of Java.

In the absence of a BROKER SCHEMA statement, all functions, procedures, modules, and constants belong to the default schema. The default schema is similar to the default namespace in C++, the no-target namespace in XML Schema, and the default package in Java.

The concept of BROKER SCHEMA in ESQL language is equivalent to *package* in Java. The BROKER SCHEMA is not in the broker ESQL.

PATH clause

The PATH clause specifies a list of additional schemas to be searched when matching function and procedure calls to their implementations. The schema in which the call lies is implicitly included in the PATH clause.

The PATH clause is used to resolve unqualified function and procedure names in the tools according to the following algorithm.

A single function or procedure must match the unqualified name, or the tools report an error. You can correct the error by qualifying the function or procedure name with a *schemaId*:

1. The current module (if any) is searched for a matching function or procedure. Module-scope functions or procedures are visible only within their containing

module. If functions or procedures with the same name are found in the current module and schema, module-scope functions or procedures take precedence over schema scoped functions or procedures.

2. The <node schema> (but none of its contained modules) and the <SQL-broker schema> or schemas identified by the PATH clause are searched for a matching function or procedure.

Note: The *schemaId* must be a fully qualified schema name.

When you start a function or procedure, the name that you use must be qualified by the schema name. The behavior depends on the circumstances:

For a module routine:

- If the schema is specified, the named schema routine is started. The scalar built-in functions, excluding CAST, EXTRACT, and the special registers, are considered to be defined within an implicitly declared schema called SQL.
- If the schema is not specified, and the calling statement is in a module routine, and a routine of the given name exists in the local module, then that local routine is started.
- If the schema is not specified, and the calling statement is in a module routine, and a routine of the given name does not exist in the local module, then all of the schemas in the schema path are searched for a routine of the same name.

If a matching function exists in one schema, it is used. A compile-time error occurs if a matching function exists in more than one schema. If there is no matching function, then the schema SQL is searched.

This rule and the preceding rule imply that a local module routine takes priority over a built-in routine of the same name.

For a schema routine:

- If the schema is specified, the named schema routine is started. The scalar built-in functions, excluding CAST, EXTRACT, and the special registers, are considered to be defined within an implicitly declared schema called SQL.
- If the schema is not specified, and the caller is a schema routine, and a routine of the given name exists in the local schema, that local routine is started.
- If the schema is not specified, and the calling statement is in a schema routine, and a routine of the given name does not exist in the local schema, then all of the schemas in the schema path are searched for a routine of the same name.

If a matching function exists in one schema, it is used. A compile-time error occurs if a matching function exists in more than one schema. If there is no matching function, the schema SQL is searched.

This rule and the preceding rule imply that a local schema routine takes priority over a built-in routine of the same name.

The <node schema> is defined as the schema containing the node's message flow.

The <node schema> is specified in this manner to provide compatibility with earlier versions of WebSphere Message Broker.

When the <node schema> is the only schema referenced, the broker XML message does not include the extra features contained in WebSphere Message Broker V6.1.

Brokers in previous versions of WebSphere Message Broker do not support multiple schemas, for example, subroutine libraries for reuse. To deploy to a broker

in a previous version of the product, put all of the ESQL subroutines into the same schema as the message flow and node that start the ESQL subroutines.

Eclipse tooling uses WebSphere Message Broker V6.1 ESQL syntax in content assist and source code validation.

The broker schema of the message flow must contain, at the schema level, any of the following in its ESQL files:

- A schema level function
- A schema level procedure
- A schema level constant
- A module level constant
- A module level variable

Without the presence of any of the preceding items, the Eclipse tooling generates broker ESQL without module and function Main wrappers.

Function and procedure names must be unique within their schema or module.

Examples

The following example adds a path to a schema called CommonUtils:

```
BROKER SCHEMA CommonUtils  
PATH SpecialUtils;  
  
MODULE ....
```

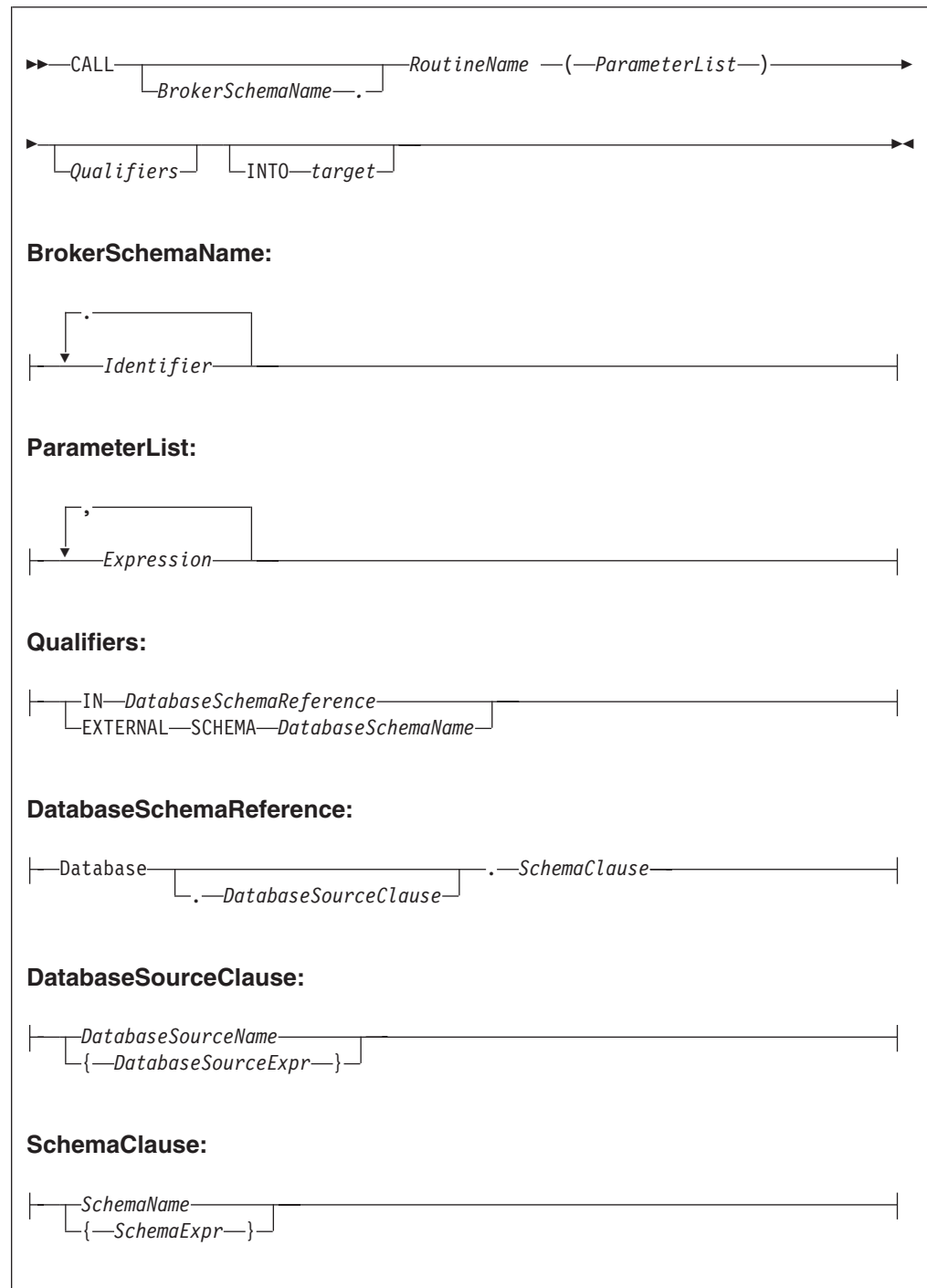
The next example adds a path to the default schema:

```
PATH CommonUtils, SpecialUtils;  
  
MODULE ....
```

CALL statement

The CALL statement calls (invokes) a routine.

Syntax



Using the CALL statement

The CALL statement invokes a routine. A routine is a user-defined function or procedure that has been defined by one of the following:

- A CREATE FUNCTION statement
- A CREATE PROCEDURE statement

As well as standard user-defined functions and procedures, you can also use CALL to invoke built-in (broker-provided) functions and user-defined SQL functions. However, the usual way of invoking these types of function is simply to include their names in expressions.

The called routine must be invoked in a way that matches its definition. For example, if you have defined a routine with three parameters, the first two of type integer and the third of type character, the CALL statement must pass three variables to the routine, each of a data type that matches the definition. This technique is called *exact signature matching*, which means that the signature provided by the CALL statement must match the signature provided by the definition of the routine.

Exact signature matching also applies to the value returned to a routine. If the RETURNS clause is specified on the CREATE FUNCTION statement, or the routine is a built-in function, the INTO clause must be specified on the CALL statement. A return value from a routine cannot be ignored. Conversely, if the RETURNS clause is not specified on the CREATE FUNCTION statement, the INTO clause must not be specified, because there is no return value from the routine.

You can use the CALL statement to invoke a routine that has been implemented in all the following ways:

- ESQL.
- Java.
- As a stored procedure in a database.
- As a built-in (broker-provided) function (although as stated earlier, names are typically included in expressions).

This variety of implementation means that some of the clauses in the CALL syntax diagram are not applicable (or allowed) for all types of routine. It also allows the CALL statement to invoke any type of routine, irrespective of how the routine has been defined.

When the optional *BrokerSchemaName* parameter is not specified, the broker SQL parser searches for the named procedure using the algorithm described in the PATH statement (see the “PATH clause” on page 194 of the BROKER SCHEMA statement).

When the *BrokerSchemaName* parameter is specified, the broker SQL parser invokes the named procedure in the specified schema without first searching the path. However, if a procedure reference is ambiguous (that is, there are two procedures with the same name in different broker schemas) and the reference is not qualified by the optional *BrokerSchemaName*, the Eclipse toolset generates an error in the Problems view that you must correct to deploy the ambiguous code.

The broker-provided built-in functions are automatically placed in a predefined broker schema called SQL. The SQL schema is always searched last for a routine that has not been matched to a user-defined routine. Therefore, a user-defined module takes precedence over a built-in routine of the same name.

Each broker schema provides a unique symbol or namespace for a routine, so a routine name is unique when it is qualified by the name of the schema to which it belongs.

The INTO clause is used to store the return value from a routine that has been defined with a RETURNS clause, or from a built-in function. The *target* can be an

ESQL variable of a data type that matches the data type on the RETURNS clause, or a dot-separated message reference. For example, both of the following ESQL statements are valid:

```
CALL myProc1() INTO cursor;  
CALL myProc1() INTO OutputRoot.XMLNS.TestValue1;
```

The CALL statement passes the parameters into the procedure in the order given to it. Parameters that have been defined as IN or INOUT on the routine's definition are evaluated before the CALL is made, but parameters defined as OUT are always passed in as NULL parameters of the correct type. When the procedure has completed, any parameters declared as OUT or INOUT are updated to reflect any changes made to them during the procedure's execution. Parameters defined as IN are never changed during the course of a procedure's execution.

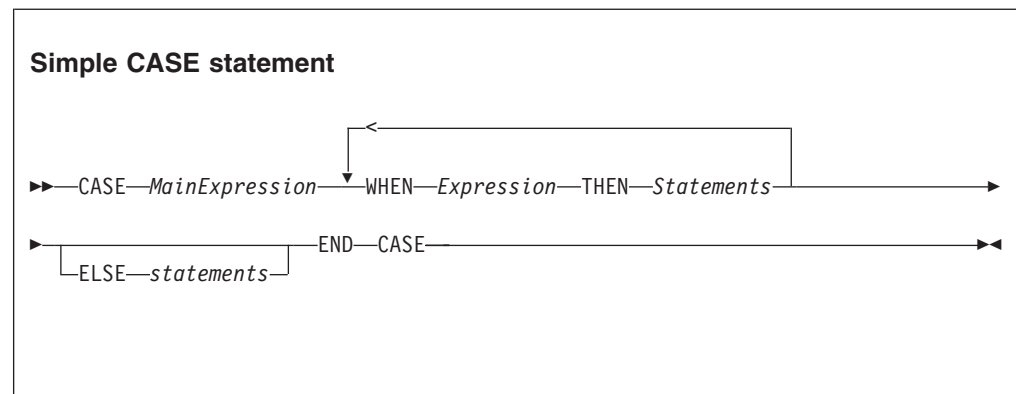
Routine overloading is not supported, which means that you cannot create two routines of the same name in the same broker schema. If the broker detects that a routine has been overloaded, it raises an exception. Similarly, you cannot invoke a database stored procedure that has been overloaded. A database stored procedure is overloaded if another procedure of the same name exists in the same database schema. However, you can invoke an overloaded Java method, as long as you create a separate ESQL definition for each overloaded method you want to call, and give each ESQL definition a unique routine name.

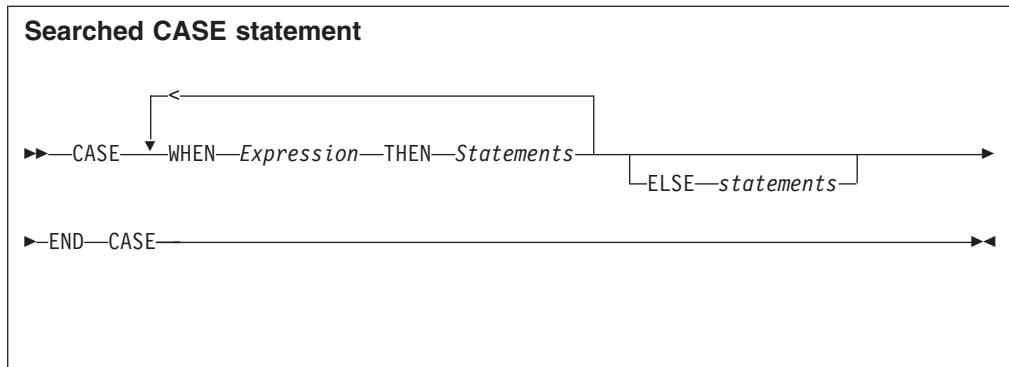
CASE statement

The CASE statement uses rules defined in WHEN clauses to select a block of statements to process.

There are two forms of the CASE statement: the simple form and the searched form.

Syntax





In the simple form, the main expression is evaluated first. Each WHEN clause expression is evaluated in turn until the result is equal to the main expression's result. That WHEN clause's statements are then processed. If no match is found and the optional ELSE clause is present, the ELSE clause's statements are executed instead. The test values do not have to be literals. The only requirement is that the main expression and the WHEN clause expressions evaluate to types that can be compared.

In the searched form, each WHEN clause expression is evaluated in turn until one evaluates to TRUE. That WHEN clause's statements are then executed. If none of the expressions evaluates to TRUE and the optional ELSE clause is present, the ELSE clause's statements are executed. There does not have to be any similarity between the expressions in each CASE clause. The only requirement is that they all evaluate to a Boolean value.

The ESQL language has both a CASE statement and a CASE function (see "CASE function" on page 329 for details of the CASE function). The CASE statement chooses one of a set of statements to execute. The CASE function chooses one of a set of expressions to evaluate and returns as its value the return value of the chosen expression.

Examples

Simple CASE statement:

```

CASE size
  WHEN minimum + 0 THEN
    SET description = 'small';
  WHEN minimum + 1 THEN
    SET description = 'medium';
  WHEN minimum + 2 THEN
    SET description = 'large';
    CALL handleLargeObject();
  ELSE
    SET description = 'unknown';
    CALL handleError();
END CASE;

```

Searched CASE statement:

```

CASE
  WHEN i <> 0 THEN
    CALL handleI(i);
  WHEN j > 1 THEN

```

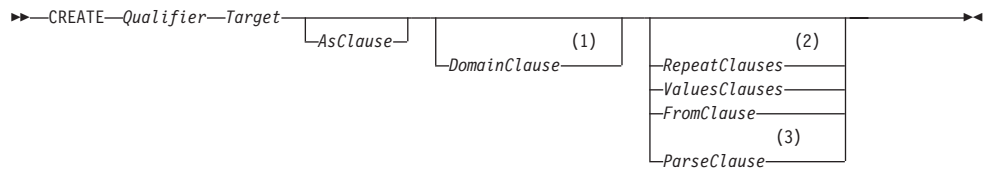


```
        CALL handleIZeroAndPositiveJ(j);
    ELSE
        CALL handleAllOtherCases(j);
END CASE;
```

CREATE statement

The CREATE statement creates a new message field.

Syntax



Qualifier:



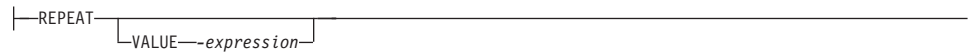
AsClause:



DomainClause:



RepeatClauses:



ValuesClauses:



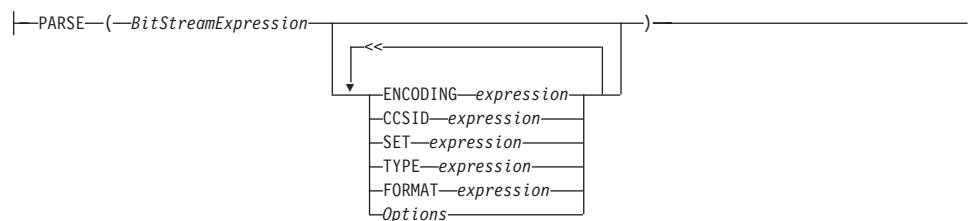
NamesClauses:



FromClause:



ParseClause:



field that is specified by Target. When creating PREVIOUSIBLING or NEXTSIBLING, you can use the REPEAT keyword to copy the type and name of the new field from the current field.

AS clause:

If present, the AS clause moves the named reference variable to point at the newly-created field. Use this clause if you want to involve the new field in some further processing.

DOMAIN clause:

If present, the DOMAIN clause associates the new field with a new parser of the specified type. This clause expects a root field name (for example, XMLNS or MQRFH2). If the DOMAIN clause is present, but the value supplied is a zero-length character string, a new parser of the same type as the parser that owns the field specified by target is created. An exception is thrown if the supplied domain name is not CHARACTER data type or its value is NULL. Do not specify the DOMAIN clause with the FIELD clause; it is not certain that a new field is created.

REPEAT clause:

Use the REPEAT clause to copy the new field's type and name from the target field. Alternatively, the new field's type, name, and value can be:

- Copied from any existing field (using the FROM clause)
- Specified explicitly (using the VALUES clause)
- Defined by parsing a bit stream (using the PARSE clause)

In the case of the FROM and PARSE clauses, you can also create children of the new field.

VALUES clause:

For the VALUES clause, the type, name, and value (or any subset of these) can be specified by any expression that returns a suitable data type (INTEGER for type, CHARACTER for name, and any scalar type for value). An exception is thrown if the value supplied for a type or name is NULL.

NAMES clause:

The NAMES clause takes any expression that returns a non-null value of type character. The meaning depends on the presence of NAME and NAMESPACE clauses as follows:

NAMESPACE	NAME	Element named as follows
No	No	The element is nameless (the name flag is not automatically set).
No	Yes	The element is given the name in the default namespace.
Yes	No	The element is given the empty name in the given namespace.
Yes	Yes	The element is given the given name in the given namespace.

The IDENTITY operand takes a single path element in place of the TYPE and NAME clauses, where a path element contains (at most) a type, a namespace, a name, and an index. These elements specify the type, namespace, name, and index of the element to be created and follow all the rules described in the topic for field references (see “ESQL field reference overview” on page 174). For example:

```
IDENTITY (XMLNS.attribute)Space1:Name1[42]
```

See the following Examples section for information about how to use the IDENTITY operand.

FROM clause:

For the FROM clause, the new field's type, name, and value are taken from the field pointed to by *SourceFieldReference*. Any existing child fields of the target are detached (the field might already exist in the case of a FIELD clause), and the new field is given copies of the source field's children.

PARSE clause:

If a PARSE clause is present, a subtree is built under the newly-created field from the supplied bit stream. The algorithm for building the subtree varies from parser to parser and according to the options specified. All parsers support the mode RootBitStream, in which the tree creation algorithm is the same as that used by an input node.

Some parsers also support a second mode, FolderBitStream, which generates a subtree from a bit stream created by the ASBITSTREAM function (see “ASBITSTREAM function” on page 316) that is using that mode.

When you use the PARSE clause, specify a scalar value containing the bit stream that is to be parsed for *BitStreamExpression*. If you use a message tree field reference you must ensure it contains a scalar value that contains the bit stream. An existing message body folder such as InputRoot.XMLNSC does not contain a bit stream and therefore cannot be used this to serialize the XMLNSC folder. If you pass a value other than a scalar containing the bit stream to the PARSE clause for *BitStreamExpression*, the message flow produces a BIP2906 error message. Instead, you must first call the ASBITSTREAM function to serialize the existing message tree folder. The result of the ASBITSTREAM function can then be passed as the *BitStreamExpression* to the PARSE clause.

The following example shows how to serialize the XMLNSC folder, then use the result of the ASBITSTREAM in the PARSE clause.

```
DECLARE inCCSID INT InputProperties.CodedCharSetId;
DECLARE inEncoding INT InputProperties.Encoding;
DECLARE inBitStream BLOB ASBITSTREAM(InputRoot.XMLNSC, inEncoding, inCCSID);
CREATE LASTCHILD OF OutputRoot DOMAIN('MRM')
    PARSE(inBitStream, inEncoding, inCCSID, 'DP3UK14002001',
        'TestCase', 'XML1', options);
```

When the PARSE statement is processed, any PARSE clause expressions are evaluated. An exception is thrown if any of the following expressions do not result in a non-null value of the appropriate type:

Clause	Type	Default value
OPTIONS	Integer	RootBitStream & ValidateNone
ENCODING	Integer	0

Clause	Type	Default value
CCSID	Integer	0
SET	Character	Zero length string
TYPE	Character	Zero length string
FORMAT	Character	Zero length string

For details of the syntax of the TYPE clause, see Specifying namespaces in the Message Type property.

Although the OPTIONS clause accepts any expression that returns a value of type integer, it is only meaningful to generate option values from the list of supplied constants, using the BITOR function if more than one option is required.

Once generated, the value becomes an integer and you can save it in a variable or pass it as a parameter to a function, as well as using it directly with a CREATE statement. The list of globally defined constants is:

```

Validate master options...
ValidateContentAndValue
ValidateValue -- Can be used with ValidateContent
ValidateContent -- Can be used with ValidateValue
ValidateNone

Validate failure action options...
ValidateException
ValidateExceptionList
ValidateLocalError
ValidateUserTrace

Validate timing options...
ValidateComplete
ValidateImmediate
ValidateDeferred

```

Notes:

1. For full details of the validation options, refer to Validation properties.
2. The *Validate timing options* correspond to *Parse Timing options* and, in particular, *ValidateDeferred* corresponds to *Parse Timing On Demand*.

C and Java equivalent APIs

Note that equivalent options are not available on:

- The Java plugin node API MBElement methods createElementAsLastChildFromBitstream() and toBitstream()
- The C plugin node API methods cniCreateElementAsLastChildFromBitstream() and cniElementAsBitstream.

You can specify only one option from each group, with the exception of ValidateValue and ValidateContent, which you can use together to obtain the content and value validation. If you do not specify an option within a group, the option in bold is used.

The ENCODING clause accepts any expression that returns a value of type integer. However, it is only meaningful to generate option values from the list of supplied constants:

```
MQENC_INTEGER_NORMAL
MQENC_INTEGER_REVERSED
MQENC_DECIMAL_NORMAL
MQENC_DECIMAL_REVERSED
MQENC_FLOAT_IEEE_NORMAL
MQENC_FLOAT_IEEE_REVERSED
MQENC_FLOAT_S390
```

The values used for the CCSID clause follow the normal numbering system. For example, 1200 = UCS-2, 1208 = UTF-8.

For absent clauses, the given default values are used. Use the CCSID and encoding default values because these take their values from the queue manager's encoding and CCSID settings.

Similarly, using the default values for each of the message set, type, and format options is useful, because many parsers do not require message set, type, or format information, and so any valid value is sufficient.

When any expressions have been evaluated, a bit stream is parsed using the results of the expressions.

Note: Because this function has a large number of clauses, an alternative syntax is supported, in which the parameters are supplied as a comma-separated list rather than by named clauses. In this case the expressions must be in the order:

```
ENCODING -> CCSID -> SET -> TYPE -> FORMAT -> OPTIONS
```

The list can be truncated at any point and an entirely empty expression can be used in any clauses where you do not supply a value.

Examples of how to use the CREATE statement

1. The following example creates the specified field:

```
CREATE FIELD OutputRoot.XMLNS.Data;
```

2. The following example creates a field with no name, type, or value as the first child of ref1:

```
CREATE FIRSTCHILD OF ref1;
```

3. The following example creates a field using the specified type, name, and value:

```
CREATE NEXTSIBLING OF ref1 TYPE NameValue NAME 'Price' VALUE 92.3;
```

4. The following example creates a field with a type and name, but no value; the field is added before the sibling indicated by the dynamic reference (ref1):

```
CREATE PREVIOUSIBLING OF ref1 TYPE Name NAME 'Quantity';
```

5. The following example creates a field named Component, and moves the reference variable targetCursor to point at it:

```
CREATE FIRSTCHILD OF targetCursor AS targetCursor NAME 'Component';
```

6. The following example creates a new field as the right sibling of the field pointed to by the reference variable targetCursor having the same type and name as that field. The statement then moves targetCursor to point at the new field:

```
CREATE NEXTSIBLING OF targetCursor AS targetCursor REPEAT;
```

7. The following example shows how to use the PARSE clause:

```

DECLARE bodyBlob BLOB ASBITSTREAM(InputRoot.XMLNS, InputProperties.Encoding,
    InputProperties.CodedCharSetId);
DECLARE creationPtr REFERENCE TO OutputRoot;
CREATE LASTCHILD OF creationPtr DOMAIN('XMLNS') PARSE(bodyBlob,
    InputProperties.Encoding,
    InputProperties.CodedCharSetId);

```

This example can be extended to show the serializing and parsing of a field or folder:

```

DECLARE bodyBlob BLOB ASBITSTREAM(InputRoot.XMLNS.TestCase.myFolder,
    InputProperties.Encoding,
    InputProperties.CodedCharSetId, ",", " ", FolderBitStream);
DECLARE creationPtr REFERENCE TO OutputRoot;
CREATE LASTCHILD OF creationPtr DOMAIN('XMLNS') PARSE(bodyBlob,
    InputProperties.Encoding,
    InputProperties.CodedCharSetId, ",", " ", FolderBitStream);

```

8. The following example shows how to use the IDENTITY operand:

```

CREATE FIELD OutputRoot.XMLNS.TestCase.Root IDENTITY (XML.ParserRoot)Root;
CREATE FIELD OutputRoot.XMLNS.TestCase.Root.Attribute
    IDENTITY (XML.Attribute)NSpace1:Attribute VALUE 'Attrib Value';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Root
    IDENTITY (XML.Element)NSpace1:Element1[1] VALUE 'Element 1 Value';
CREATE LASTCHILD OF OutputRoot.XMLNS.TestCase.Root
    IDENTITY (XML.Element)NSpace1:Element1[2] VALUE 'Element 2 Value';

```

This produces the following output message:

```

<TestCase>
  <Root xmlns:NS1="NSpace1" NS1:Attribute="Attrib Value">
    <NS1:Element1>Element 1 Value</NS1:Element1>
    <NS1:Element1>Element 2 Value</NS1:Element1>
  </Root>
</TestCase>

```

9. The following example shows how you can use the DOMAIN clause to avoid losing information unique to the XMLNS parser when an unlike parser copy occurs:

```

DECLARE bodyBlob BLOB ASBITSTREAM(InputRoot.XMLNS, InputProperties.Encoding,
    InputProperties.CodedCharSetId);
CREATE FIELD Environment.Variables.myXMLTree;
DECLARE creationPtr REFERENCE TO Environment.Variables.myXMLTree;
CREATE FIRSTCHILD OF creationPtr DOMAIN('XMLNS') PARSE(bodyBlob,
    InputProperties.Encoding,
    InputProperties.CodedCharSetId);

```

An example of a CREATE statement

This example provides sample ESQL and an input message, which together produce the output message at the end of the example.

```

CREATE COMPUTE MODULE CreateStatement_Compute
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    CALL CopyMessageHeaders();

    CREATE FIELD OutputRoot.XMLNS.TestCase.description TYPE NameValue VALUE 'This is my TestCase' ;
    DECLARE cursor REFERENCE TO OutputRoot.XMLNS.TestCase;
    CREATE FIRSTCHILD OF cursor Domain('XMLNS')
        NAME 'Identifier' VALUE InputRoot.XMLNS.TestCase.Identifier;
    CREATE LASTCHILD OF cursor Domain('XMLNS') NAME 'Sport' VALUE InputRoot.XMLNS.TestCase.Sport;
    CREATE LASTCHILD OF cursor Domain('XMLNS') NAME 'Date' VALUE InputRoot.XMLNS.TestCase.Date;
    CREATE LASTCHILD OF cursor Domain('XMLNS') NAME 'Type' VALUE InputRoot.XMLNS.TestCase.Type;
    CREATE FIELD
        cursor.Division[1].Number TYPE NameValue VALUE 'Premiership';
    CREATE FIELD
        cursor.Division[1].Result[1].Number TYPE NameValue VALUE '1' ;
    CREATE FIELD
        cursor.Division[1].Result[1].Home TYPE Name;
    CREATE LASTCHILD OF
        cursor.Division[1].Result[1].Home NAME 'Team' VALUE 'Liverpool' ;
    CREATE LASTCHILD OF
        cursor.Division[1].Result[1].Home NAME 'Score' VALUE '4';

```



```

CREATE FIELD          cursor.Division[1].Result[1].Away TYPE Name;
CREATE LASTCHILD OF  cursor.Division[1].Result[1].Away NAME 'Team' VALUE 'Everton';
CREATE LASTCHILD OF  cursor.Division[1].Result[1].Away NAME 'Score' VALUE '0';
CREATE FIELD          cursor.Division[1].Result[2].Number TYPE NameValue VALUE '2';
CREATE FIELD          cursor.Division[1].Result[2].Home TYPE Name;
CREATE LASTCHILD OF  cursor.Division[1].Result[2].Home NAME 'Team' VALUE 'Manchester United';
CREATE LASTCHILD OF  cursor.Division[1].Result[2].Home NAME 'Score' VALUE '2';
CREATE FIELD          cursor.Division[1].Result[2].Away TYPE Name;
CREATE LASTCHILD OF  cursor.Division[1].Result[2].Away NAME 'Team' VALUE 'Arsenal';
CREATE LASTCHILD OF  cursor.Division[1].Result[2].Away NAME 'Score' VALUE '3';
CREATE FIELD          cursor.Division[2].Number TYPE NameValue VALUE '2';
CREATE FIELD          cursor.Division[2].Result[1].Number TYPE NameValue VALUE '1';
CREATE FIELD          cursor.Division[2].Result[1].Home TYPE Name;
CREATE LASTCHILD OF  cursor.Division[2].Result[1].Home NAME 'Team' VALUE 'Port Vale';
CREATE LASTCHILD OF  cursor.Division[2].Result[1].Home NAME 'Score' VALUE '9' ;
CREATE FIELD          cursor.Division[2].Result[1].Away TYPE Name;
CREATE LASTCHILD OF  cursor.Division[2].Result[1].Away NAME 'Team' VALUE 'Brentford';
CREATE LASTCHILD OF  cursor.Division[2].Result[1].Away NAME 'Score' VALUE '5';

END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
  DECLARE I INTEGER 1;
  DECLARE J INTEGER CARDINALITY(InputRoot.*[]);
  WHILE I < J DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I = I + 1;
  END WHILE;
END;

END MODULE;

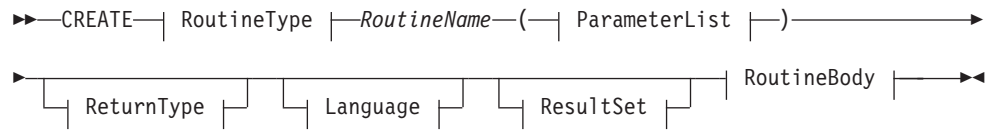
```

CREATE FUNCTION statement

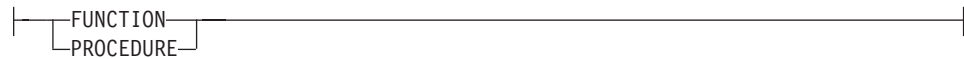
The CREATE FUNCTION statement defines a callable function or procedure.

You can also use the CREATE PROCEDURE statement to define a callable function or procedure, also known as a routine.

Syntax



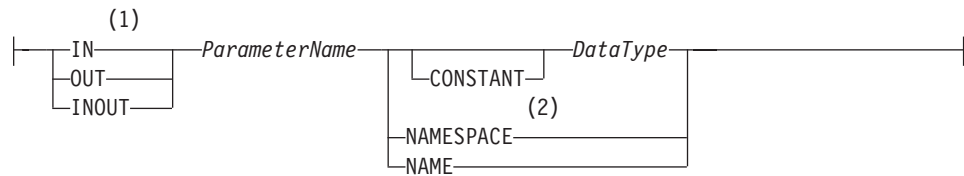
RoutineType:



ParameterList:



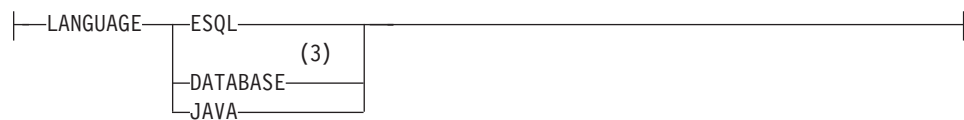
Parameter:



ReturnType:



Language:



ResultSet:



RoutineBody:



Notes:

- 1 If the routine type is FUNCTION, the direction indicator (IN, OUT, or

INOUT) is optional for each parameter. However, for documentation purposes, it is good programming practice to specify a direction indicator for all new routines; if you do not specify the direction, a default value of IN is used.

- 2 When the NAMESPACE or NAME clause is used, its value is implicitly CONSTANT and of type CHARACTER. For information about the use of CONSTANT variables, see the “DECLARE statement” on page 235.
- 3 If the routine type is FUNCTION, you cannot specify a LANGUAGE of DATABASE.

Overview

The CREATE FUNCTION and CREATE PROCEDURE statements define a callable function or procedure, also known as a routine.

In previous versions of this product, CREATE FUNCTION and CREATE PROCEDURE had different uses and different capabilities. Subsequent enhancements have resulted in the differences listed previously in notes 1 and 3.

Routines are useful for creating reusable blocks of code that can be run independently many times. You can implement them as a series of ESQL statements, a Java method, or a database stored procedure. This flexibility means that some of the clauses in the syntax diagram are not applicable (or allowed) for all types of routine.

Each routine has a name, which must be unique within the schema to which it belongs. Routine names therefore cannot be overloaded; if the broker detects that a routine name has been overloaded, it raises an exception.

The LANGUAGE clause specifies the language in which the routine's body is written. The options are:

DATABASE

The procedure is called as a database stored procedure.

ESQL

The procedure is called as an ESQL routine.

JAVA

The procedure is called as a static method in a Java class.

Unspecified

If you do not specify the LANGUAGE clause, the default language is ESQL unless you specify the EXTERNAL NAME clause (in which case, the default language is DATABASE).

Restrictions on the use of the LANGUAGE clause exist. You cannot use:

- The ESQL option with an EXTERNAL NAME clause
- The DATABASE or JAVA options without an EXTERNAL NAME clause
- The DATABASE option with a routine type of FUNCTION

Specify the name of the routine using the *RoutineName* clause, and its parameters by using the *ParameterList* clause. If the LANGUAGE clause specifies ESQL, implement the routine using a single ESQL statement. This statement is most

useful if it is a compound statement (BEGIN ... END), because it can then contain as many ESQL statements as necessary to fulfil its function.

Alternatively, instead of providing an ESQL body for the routine, you can specify a LANGUAGE clause other than ESQL. You can then use the EXTERNAL NAME clause to provide a reference to the actual body of the routine, wherever it is located externally to the broker. For more information about using the EXTERNAL NAME clause, see “Invoking stored procedures” on page 77 and Calling a Java routine.

Routines of any LANGUAGE type can have IN, OUT, and INOUT parameters. The caller can pass several values into the routine, and receive back several updated values. These returned parameters are in addition to any RETURNS clause that you have defined for the routine. The RETURNS clause defines the value that the routine returns to the caller.

Routines that are implemented in different languages have their own restrictions on which data types can be passed in or returned; these restrictions are documented later in this section. The data type of the returned value must match the data type of the value that is defined to be returned from the routine. Also, if a routine is defined to have a return value, the caller of the routine cannot ignore it. For more information see “CALL statement” on page 196.

Routines can be defined in either a module or a schema. Routines that are defined in a module are local in scope to the current node, which means that only code belonging to that same module (or node) can invoke them. Routines that are defined in a schema, however, can be invoked by using either of the following options:

- Code in the same schema
- Code in any other schema, if either of the following conditions applies:
 - The other schema's PATH clause contains the path to the called routine
 - The called routine is invoked using its fully qualified name (which is its name, prefixed by its schema name, separated by a period)

Thus, if you need to invoke the same routine in more than one node, define it in a schema.

For any language or routine type, the method of invocation of the routine must match the manner of declaration of the routine. If the routine has a RETURNS clause, use either the FUNCTION invocation syntax or a CALL statement with an INTO clause. Conversely, if a routine has no RETURNS clause, you must use a CALL statement without an INTO clause.

Parameter directions

Parameters that are passed to routines always have a direction associated with them, which is one of the following types:

IN The value of the parameter cannot be changed by the routine. A NULL value for the parameter is allowed, and can be passed to the routine.

OUT

When it is received by the called routine, the parameter that is passed into the routine always has a NULL value of the correct data type. This value is set irrespective of its value before the routine is called. The routine is allowed to change the value of the parameter.

INOUT

INOUT is both an IN and an OUT parameter. It passes a value into the routine, and the value that is passed in can be changed by the routine. A NULL value for the parameter is allowed, and can be passed both into and out of the routine.

If the routine type is FUNCTION, the direction indicator (IN, OUT, INOUT) is optional for each parameter. However, it is good programming practice to specify a direction indicator for all new routines of any type for documentation purposes.

ESQL variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction OUT or INOUT.

ESQL routines

ESQL routines are written in ESQL, and have a LANGUAGE clause of ESQL. The body of an ESQL routine is typically a compound statement of the form BEGIN ... END, that contains multiple statements for processing the parameters that are passed to the routine.

ESQL example 1

The following example shows the same procedure as in “Database routine example 1” on page 231, but is implemented as an ESQL routine and not as a stored procedure. The CALL syntax and results of this routine are the same as those in “Restrictions on Java routines” on page 217.

```
CREATE PROCEDURE swapParms (  
  IN parm1 CHARACTER,  
  OUT parm2 CHARACTER,  
  INOUT parm3 CHARACTER )  
BEGIN  
  SET parm2 = parm3;  
  SET parm3 = parm1;  
END;
```

ESQL example 2

This example procedure shows the recursive use of an ESQL routine. It parses a tree, visiting all places at and below the specified starting point, and reports what it has found:

```
SET OutputRoot.MQMD = InputRoot.MQMD;  
  
DECLARE answer CHARACTER;  
SET answer = '';  
  
CALL navigate(InputRoot.XMLNS, answer);  
SET OutputRoot.XMLNS.Data.FieldNames = answer;  
  
CREATE PROCEDURE navigate (IN root REFERENCE, INOUT answer CHARACTER)  
BEGIN  
  SET answer = answer || 'Reached Field... Type:'  
  || CAST(FIELDTYPE(root) AS CHAR)||  
  ': Name:' || FIELDNAME(root) || ': Value : ' || root || ': '  
  
  DECLARE cursor REFERENCE TO root;  
  MOVE cursor FIRSTCHILD;  
  IF LASTMOVE(cursor) THEN  
    SET answer = answer || 'Field has children... drilling down '  
  ELSE
```

```

    SET answer = answer || 'Listing siblings... ';
END IF;

WHILE LASTMOVE(cursor) DO
    CALL navigate(cursor, answer);
    MOVE cursor NEXTSIBLING;
END WHILE;

SET answer = answer || 'Finished siblings... Popping up ';
END;

```

When given the following input message:

```

<Person>
  <Name>John Smith</Name>
  <Salary period='monthly' taxable='yes'>-1200</Salary>
</Person>

```

the procedure produces the following output, which has been manually formatted:

```

Reached Field... Type:16777232: Name:XML: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Person: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Name:
Value :John Smith: Field has children... drilling down
Reached Field... Type:33554432: Name::
Value :John Smith: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Reached Field... Type:16777216: Name:Salary:
Value :-1200: Field has children... drilling down
Reached Field... Type:50331648: Name:period:
Value :monthly: Listing siblings... Finished siblings... Popping up
Reached Field... Type:50331648: Name:taxable:
Value :yes: Listing siblings... Finished siblings... Popping up
Reached Field... Type:33554432: Name::
Value :-1200: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up

```

Java routines

A Java routine is implemented as a Java method, and has a LANGUAGE clause of JAVA. For Java routines, the *ExternalRoutineName* must contain the class name and method name of the Java method to be called. Specify the *ExternalRoutineName* like this:

```
>>--"-- className---.---methodName--"-----<<
```

where *className* identifies the class that contains the method and *methodName* identifies the method to invoke. If the class is part of a package, the class identifier part must include the complete package prefix; for example, "com.ibm.broker.test.MyClass.myMethod".

To find the Java class, the broker uses the search method that is described in "Deploying Java classes" on page 217.

Any Java method that you want to invoke must have the following basic signature:
public static <return-type> <method-name> (< 0 - N parameters>)

where <return-type> must be in the list of Java IN data types in the table in "ESQL to Java data type mapping" on page 216 (excluding the REFERENCE type, which is not permitted as a return value), or the Java void data type. The

parameter data types must also be in the “ESQL to Java data type mapping” on page 216 table. In addition, the Java method is not allowed to have an exception throws clause in its signature.

The Java method's signature must match the ESQL routine's declaration of the method. You must also observe the following rules:

- Ensure that the Java method name, including the class name and any package qualifiers, matches the procedure's EXTERNAL NAME.
- If the Java return type is void, do not put a RETURNS clause on the ESQL routine's definition. Conversely, if the Java return type is not void, you must put a RETURNS clause on the ESQL routine's definition.
- Ensure that every parameter's type and direction matches the ESQL declaration, according to the rules listed in the table in “ESQL to Java data type mapping” on page 216.
- Ensure that the method's return type matches the data type of the RETURNS clause.
- Enclose EXTERNAL NAME in quotation marks because it must contain at least "class.method".
- If you want to invoke an overloaded Java method, you must create a separate ESQL definition for each overloaded method and give each ESQL definition a unique routine name.

You can use the Java user-defined node API in your Java method, provided that you observe the restrictions documented in “Restrictions on Java routines” on page 217. For more information about using the Java API, see Compiling a Java user-defined node.

Java routine example 1

This routine contains three parameters of varying directions, and returns an integer, which maps to a Java return type of `java.lang.Long`.

```
CREATE FUNCTION myProc1( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  RETURNS INTEGER
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod1";
```

You can use the following ESQL to invoke `myProc1`:

```
CALL myProc1( intVar1, intVar2, intVar3) INTO intReturnVar3;
-- or
SET intReturnVar3 = myProc1( intVar1, intVar2, intVar3);
```

Java routine example 2

This routine contains three parameters of varying directions and has a Java return type of `void`.

```
CREATE PROCEDURE myProc2( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod2";
```

You must use the following ESQL to invoke `myProc2`:

```
CALL myProc2(intVar1, intVar2, intVar3);
```

The following Java class provides a method for each of the preceding Java examples:

```

package com.ibm.broker.test;

class MyClass {
public static Long myMethod1( Long P1, Long[] P2 Long[] P3) { ... }
public static void myMethod2( Long P2, Long[] P2 Long[] P3) { ... }

/* When either of these methods is called:
P1 might or might not be NULL (depending on the value of intVar1).
P2[0] is always NULL (whatever the value of intVar2).
P3[0] might or might not be NULL (depending on the value of intVar3).
This is the same as with LANGUAGE ESQL routines.
When these methods return:
    intVar1 is unchanged
    intVar2 might still be NULL or might have been changed
    intVar3 might contain the same value or might have been changed.
This is the same as with LANGUAGE ESQL routines.

When myMethod1 returns: intReturnVar3 is either NULL (if the
method returns NULL) or it contains the value returned by the
method.
*/
}

```

ESQL to Java data type mapping

The following table summarizes the mappings from ESQL to Java.

Notes:

- Only the Java scalar wrappers are passed to Java.
- The ESQL scalar types are mapped to Java data types as object wrappers, or object wrapper arrays, depending upon the direction of the procedure parameter. Each wrapper array contains exactly one element.
- Scalar object wrappers are used to allow NULL values to be passed to and from Java methods.

ESQL data types ¹	Java IN data types	Java INOUT and OUT data types
INTEGER, INT	java.lang.Long	java.lang.Long []
FLOAT	java.lang.Double	java.lang.Double[]
DECIMAL	java.math.BigDecimal	java.math.BigDecimal[]
CHARACTER, CHAR	java.lang.String	java.lang.String[]
BLOB	byte[]	byte[][]
BIT	java.util.BitSet	java.util.BitSet[]
DATE	com.ibm.broker.plugin.MbDate	com.ibm.broker.plugin.MbDate[]
TIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
GMTTIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
TIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
GMTTIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
INTERVAL	Not supported	Not supported
BOOLEAN	java.lang.Boolean	java.lang.Boolean[]
REFERENCE (to a message tree) ^{3 4} _{5 6}	com.ibm.broker.plugin.MbElement	com.ibm.broker.plugin.MbElement[] (Supported for INOUT. Not supported for OUT)
ROW	Not supported	Not supported

LIST	Not supported	Not supported
------	---------------	---------------

1. Variables that are declared to be CONSTANT (or references to variables that are declared to be CONSTANT) are not allowed to have the direction INOUT or OUT.
2. The time zone set in the Java variable is not important; you obtain the required time zone in the output ESQL.
3. The reference parameter cannot be NULL when passed into a Java method.
4. The reference cannot have the direction OUT when passed into a Java method.
5. If an *MbElement* is passed back from Java to ESQL as an INOUT parameter, it must point to a location in the same message tree as that pointed to by the *MbElement* that was passed into the called Java method.
For example, if an ESQL reference to `OutputRoot.XML.Test` is passed into a Java method as an INOUT *MbElement*, but a different *MbElement* is passed back to ESQL when the call returns, the different element must also point to somewhere in the `OutputRoot` tree.
6. An *MbElement* cannot be returned from a Java method with the RETURNS clause, because no ESQL routine can return a reference. However, an *MbElement* can be returned as an INOUT direction parameter, subject to the conditions described in point 5.

A REFERENCE to a scalar variable can be used in the CALL of a Java method, provided that the data type of the variable to which the reference refers matches the corresponding data type in the Java program signature.

Restrictions on Java routines

The following restrictions apply to Java routines that are called from ESQL:

- The Java method must be threadsafe (reentrant).
- Database connections must be JDBC type 2 or type 4. Furthermore, database operations are not part of a broker transaction and therefore cannot be controlled by an external resource coordinator (as is the case in an XA environment).
- The Java user-defined node API must be used only by the same thread that invoked the Java method.

You can create threads inside your method. However, created threads must not use the Java APIs, and you must return control back to the broker.

All restrictions that apply to the usage of the Java API also apply to Java methods that are called from ESQL.

- Java methods that are called from ESQL must not use the `MbNode` class. Therefore, they cannot create objects of type `MbNode`, or call any of the methods on an existing `MbNode` object.
- WebSphere MQ or JMS work done inside a Java method that is called from ESQL must be done in accordance with the guidelines for performing WebSphere MQ and JMS work in a user-defined node. See *Planning user-defined input nodes*.

Deploying Java classes

You can deploy your Java classes to a broker within a Java Archive (JAR) file, using one of the following two methods:

1. **Add the JAR file to the broker archive (BAR) file**

The most efficient and flexible method of deploying to the broker is to add your JAR file to the BAR file. You can do this manually or automatically using the workbench.

If the workbench finds the correct Java class inside a referenced Java project open in the workspace, it automatically compiles the Java class into a JAR file and adds it to the BAR file. This procedure is the same procedure that you follow to deploy a JavaCompute node inside a JAR, as described in *User-defined node classloading*.

When you deploy a JAR file from the workbench, the flow that has been redeployed reloads the JAR file contained in the BAR file.

The files are also reloaded if the message flow that references a Java class is stopped and restarted. Ensure that you stop and restart (or redeploy) all flows that reference the JAR file that you want to update. This action avoids the problem of some flows running with the old version of the JAR file and other flows running with the new version.

The workbench deploys only JAR files; it does not deploy stand-alone Java class files.

2. Store the JAR file in either of the following locations:

- a. The *workpath/shared-classes/* folder on the machine running the broker
- b. The CLASSPATH environment variable on the computer running the broker

You must complete this action manually; you cannot use the workbench.

In this method, redeploying the message flow does not reload the referenced Java classes; neither does stopping and restarting the message flow. The only way to reload the classes in this case is to stop and restart the broker itself.

To enable the broker to find a Java class, ensure that it is in one of the preceding locations. If the broker cannot find the specified class, it generates an exception.

Although you have the choices shown previously when you deploy the JAR file, using the workbench to deploy the BAR file provides the greatest flexibility when redeploying the JAR file.

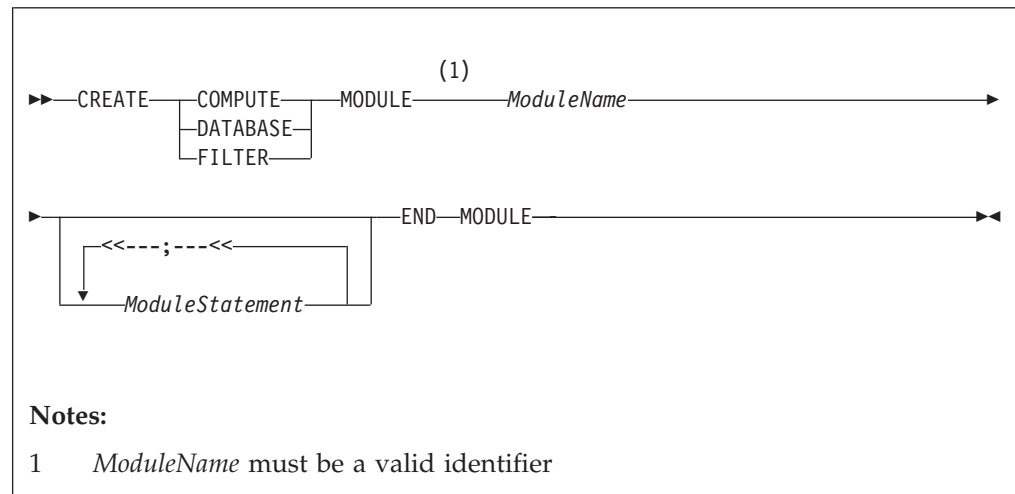
Database routines

CREATE FUNCTION does not support database routines. Use CREATE PROCEDURE to define a database routine.

CREATE MODULE statement

The CREATE MODULE statement creates a module, which is a named container associated with a node.

Syntax



A module in the Eclipse tools is referred to from a message processing node by name. The module must be in the <node schema>.

Module names occupy the same symbol space as functions and procedures defined in the schema. That is, modules, functions, and procedures contained by a schema must all have unique names.

Note: You are warned if there is no module associated with an ESQL node. You cannot deploy a flow containing a node in which a module is missing.

The modules for the Compute node, Database node, and Filter node must all contain exactly one function called Main. This function should return a Boolean. It is the entry point used by a message flow node when processing a message.

Correlation name	Compute module	Filter module	Database module
Database	×	×	×
Environment	×	×	×
Root		×	×
Body		×	×
Properties		×	×
ExceptionList		×	×
LocalEnvironment		×	×
InputRoot	×		
InputBody	×		
InputProperties	×		
InputExceptionList	×		
InputLocalEnvironment	×		
OutputRoot	×		
OutputExceptionList	×		
OutputLocalEnvironment	×		
DestinationList	Deprecated synonym for LocalEnvironment		

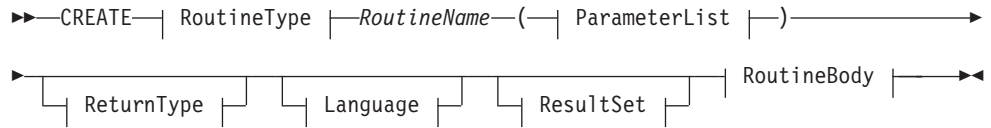
Correlation name	Compute module	Filter module	Database module
InputDestinationList	Deprecated synonym for InputLocalEnvironment		
OutputDestinationList	Deprecated synonym for OutputLocalEnvironment		

CREATE PROCEDURE statement

The CREATE PROCEDURE statement defines a callable function or procedure.

You can also use the CREATE FUNCTION statement to define a callable function or procedure, also known as a routine.

Syntax



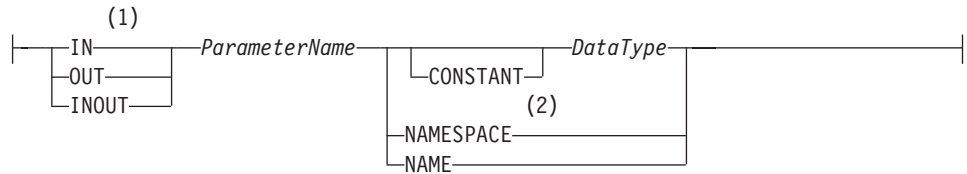
RoutineType:



ParameterList:



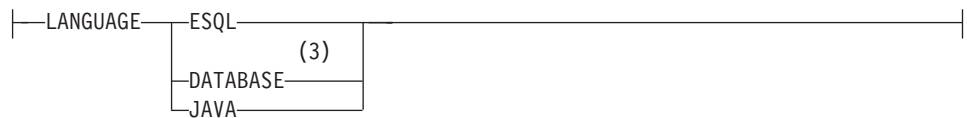
Parameter:



ReturnType:



Language:



ResultSet:



RoutineBody:



Notes:

- 1 If the routine type is FUNCTION, the direction indicator (IN, OUT, or

INOUT) is optional for each parameter. However, for documentation purposes, it is good programming practice to specify a direction indicator for all new routines; if you do not specify the direction, a default value of IN is used.

- 2 When the NAMESPACE or NAME clause is used, its value is implicitly CONSTANT and of type CHARACTER. For information about the use of CONSTANT variables, see the “DECLARE statement” on page 235.
- 3 If the routine type is FUNCTION, you cannot specify a LANGUAGE of DATABASE.

Overview

The CREATE FUNCTION and CREATE PROCEDURE statements define a callable function or procedure, also known as a routine.

In previous versions of this product, CREATE FUNCTION and CREATE PROCEDURE had different uses and different capabilities. Subsequent enhancements have resulted in the differences listed previously in notes 1 and 3.

Routines are useful for creating reusable blocks of code that can be run independently many times. You can implement them as a series of ESQL statements, a Java method, or a database stored procedure. This flexibility means that some of the clauses in the syntax diagram are not applicable (or allowed) for all types of routine.

Each routine has a name, which must be unique within the schema to which it belongs. Routine names therefore cannot be overloaded; if the broker detects that a routine name has been overloaded, it raises an exception.

The LANGUAGE clause specifies the language in which the routine's body is written. The options are:

DATABASE

The procedure is called as a database stored procedure.

ESQL

The procedure is called as an ESQL routine.

JAVA

The procedure is called as a static method in a Java class.

Unspecified

If you do not specify the LANGUAGE clause, the default language is ESQL unless you specify the EXTERNAL NAME clause (in which case, the default language is DATABASE).

Restrictions on the use of the LANGUAGE clause exist. You cannot use:

- The ESQL option with an EXTERNAL NAME clause
- The DATABASE or JAVA options without an EXTERNAL NAME clause
- The DATABASE option with a routine type of FUNCTION

Specify the name of the routine using the *RoutineName* clause, and its parameters by using the *ParameterList* clause. If the LANGUAGE clause specifies ESQL, implement the routine using a single ESQL statement. This statement is most

useful if it is a compound statement (BEGIN ... END), because it can then contain as many ESQL statements as necessary to fulfil its function.

Alternatively, instead of providing an ESQL body for the routine, you can specify a LANGUAGE clause other than ESQL. You can then use the EXTERNAL NAME clause to provide a reference to the actual body of the routine, wherever it is located externally to the broker. For more information about using the EXTERNAL NAME clause, see “Invoking stored procedures” on page 77 and Calling a Java routine.

Routines of any LANGUAGE type can have IN, OUT, and INOUT parameters. The caller can pass several values into the routine, and receive back several updated values. These returned parameters are in addition to any RETURNS clause that you have defined for the routine. The RETURNS clause defines the value that the routine returns to the caller.

Routines that are implemented in different languages have their own restrictions on which data types can be passed in or returned; these restrictions are documented later in this section. The data type of the returned value must match the data type of the value that is defined to be returned from the routine. Also, if a routine is defined to have a return value, the caller of the routine cannot ignore it. For more information see “CALL statement” on page 196.

Routines can be defined in either a module or a schema. Routines that are defined in a module are local in scope to the current node, which means that only code belonging to that same module (or node) can invoke them. Routines that are defined in a schema, however, can be invoked by using either of the following options:

- Code in the same schema
- Code in any other schema, if either of the following conditions applies:
 - The other schema's PATH clause contains the path to the called routine
 - The called routine is invoked using its fully qualified name (which is its name, prefixed by its schema name, separated by a period)

Thus, if you need to invoke the same routine in more than one node, define it in a schema.

For any language or routine type, the method of invocation of the routine must match the manner of declaration of the routine. If the routine has a RETURNS clause, use either the FUNCTION invocation syntax or a CALL statement with an INTO clause. Conversely, if a routine has no RETURNS clause, you must use a CALL statement without an INTO clause.

Parameter directions

Parameters that are passed to routines always have a direction associated with them, which is one of the following types:

IN The value of the parameter cannot be changed by the routine. A NULL value for the parameter is allowed, and can be passed to the routine.

OUT

When it is received by the called routine, the parameter that is passed into the routine always has a NULL value of the correct data type. This value is set irrespective of its value before the routine is called. The routine is allowed to change the value of the parameter.

INOUT

INOUT is both an IN and an OUT parameter. It passes a value into the routine, and the value that is passed in can be changed by the routine. A NULL value for the parameter is allowed, and can be passed both into and out of the routine.

If the routine type is FUNCTION, the direction indicator (IN, OUT, INOUT) is optional for each parameter. However, it is good programming practice to specify a direction indicator for all new routines of any type for documentation purposes.

ESQL variables that are declared to be CONSTANT (or references to variables declared to be CONSTANT) are not allowed to have the direction OUT or INOUT.

ESQL routines

ESQL routines are written in ESQL, and have a LANGUAGE clause of ESQL. The body of an ESQL routine is typically a compound statement of the form BEGIN ... END, that contains multiple statements for processing the parameters that are passed to the routine.

ESQL example 1

The following example shows the same procedure as in “Database routine example 1” on page 231, but is implemented as an ESQL routine and not as a stored procedure. The CALL syntax and results of this routine are the same as those in “Restrictions on Java routines” on page 217.

```
CREATE PROCEDURE swapParms (  
    IN parm1 CHARACTER,  
    OUT parm2 CHARACTER,  
    INOUT parm3 CHARACTER )  
BEGIN  
    SET parm2 = parm3;  
    SET parm3 = parm1;  
END;
```

ESQL example 2

This example procedure shows the recursive use of an ESQL routine. It parses a tree, visiting all places at and below the specified starting point, and reports what it has found:

```
SET OutputRoot.MQMD = InputRoot.MQMD;  
  
DECLARE answer CHARACTER;  
SET    answer = '';  
  
CALL navigate(InputRoot.XMLNS, answer);  
SET OutputRoot.XMLNS.Data.FieldNames = answer;  
  
CREATE PROCEDURE navigate (IN root REFERENCE, INOUT answer CHARACTER)  
BEGIN  
    SET answer = answer || 'Reached Field... Type:'  
    || CAST(FIELDTYPE(root) AS CHAR)||  
    ': Name:' || FIELDNAME(root) || ': Value : ' || root || ': '  
  
    DECLARE cursor REFERENCE TO root;  
    MOVE cursor FIRSTCHILD;  
    IF LASTMOVE(cursor) THEN  
        SET answer = answer || 'Field has children... drilling down '  
    ELSE
```



```

    SET answer = answer || 'Listing siblings... ';
END IF;

WHILE LASTMOVE(cursor) DO
    CALL navigate(cursor, answer);
    MOVE cursor NEXTSIBLING;
END WHILE;

SET answer = answer || 'Finished siblings... Popping up ';
END;

```

When given the following input message:

```

<Person>
  <Name>John Smith</Name>
  <Salary period='monthly' taxable='yes'>-1200</Salary>
</Person>

```

the procedure produces the following output, which has been manually formatted:

```

Reached Field... Type:16777232: Name:XML: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Person: Value :: Field has children...
drilling down
Reached Field... Type:16777216: Name:Name:
Value :John Smith: Field has children... drilling down
Reached Field... Type:33554432: Name::
Value :John Smith: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Reached Field... Type:16777216: Name:Salary:
Value :-1200: Field has children... drilling down
Reached Field... Type:50331648: Name:period:
Value :monthly: Listing siblings... Finished siblings... Popping up
Reached Field... Type:50331648: Name:taxable:
Value :yes: Listing siblings... Finished siblings... Popping up
Reached Field... Type:33554432: Name::
Value :-1200: Listing siblings... Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up
Finished siblings... Popping up

```

Java routines

A Java routine is implemented as a Java method, and has a LANGUAGE clause of JAVA. For Java routines, the *ExternalRoutineName* must contain the class name and method name of the Java method to be called. Specify the *ExternalRoutineName* like this:

```
>>--"-- className---.--methodName--"-----<<
```

where *className* identifies the class that contains the method and *methodName* identifies the method to invoke. If the class is part of a package, the class identifier part must include the complete package prefix; for example, "com.ibm.broker.test.MyClass.myMethod".

To find the Java class, the broker uses the search method that is described in "Deploying Java classes" on page 217.

Any Java method that you want to invoke must have the following basic signature:
public static <return-type> <method-name> (< 0 - N parameters>)

where <return-type> must be in the list of Java IN data types in the table in "ESQL to Java data type mapping" on page 216 (excluding the REFERENCE type, which is not permitted as a return value), or the Java void data type. The

parameter data types must also be in the “ESQL to Java data type mapping” on page 216 table. In addition, the Java method is not allowed to have an exception throws clause in its signature.

The Java method's signature must match the ESQL routine's declaration of the method. You must also observe the following rules:

- Ensure that the Java method name, including the class name and any package qualifiers, matches the procedure's EXTERNAL NAME.
- If the Java return type is void, do not put a RETURNS clause on the ESQL routine's definition. Conversely, if the Java return type is not void, you must put a RETURNS clause on the ESQL routine's definition.
- Ensure that every parameter's type and direction matches the ESQL declaration, according to the rules listed in the table in “ESQL to Java data type mapping” on page 216.
- Ensure that the method's return type matches the data type of the RETURNS clause.
- Enclose EXTERNAL NAME in quotation marks because it must contain at least "class.method".
- If you want to invoke an overloaded Java method, you must create a separate ESQL definition for each overloaded method and give each ESQL definition a unique routine name.

You can use the Java user-defined node API in your Java method, provided that you observe the restrictions documented in “Restrictions on Java routines” on page 217. For more information about using the Java API, see Compiling a Java user-defined node.

Java routine example 1

This routine contains three parameters of varying directions, and returns an integer, which maps to a Java return type of `java.lang.Long`.

```
CREATE FUNCTION myProc1( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  RETURNS INTEGER
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod1";
```

You can use the following ESQL to invoke `myProc1`:

```
CALL myProc1( intVar1, intVar2, intVar3) INTO intReturnVar3;
-- or
SET intReturnVar3 = myProc1( intVar1, intVar2, intVar3);
```

Java routine example 2

This routine contains three parameters of varying directions and has a Java return type of `void`.

```
CREATE PROCEDURE myProc2( IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER )
  LANGUAGE JAVA
  EXTERNAL NAME "com.ibm.broker.test.MyClass.myMethod2";
```

You must use the following ESQL to invoke `myProc2`:

```
CALL myProc2(intVar1, intVar2, intVar3);
```

The following Java class provides a method for each of the preceding Java examples:

```

package com.ibm.broker.test;

class MyClass {
public static Long myMethod1( Long P1, Long[] P2 Long[] P3) { ... }
public static void myMethod2( Long P2, Long[] P2 Long[] P3) { ... }

/* When either of these methods is called:
   P1 might or might not be NULL (depending on the value of intVar1).
   P2[0] is always NULL (whatever the value of intVar2).
   P3[0] might or might not be NULL (depending on the value of intVar3).
   This is the same as with LANGUAGE ESQL routines.
   When these methods return:
       intVar1 is unchanged
       intVar2 might still be NULL or might have been changed
       intVar3 might contain the same value or might have been changed.
   This is the same as with LANGUAGE ESQL routines.

   When myMethod1 returns: intReturnVar3 is either NULL (if the
   method returns NULL) or it contains the value returned by the
   method.
*/
}

```

ESQL to Java data type mapping

The following table summarizes the mappings from ESQL to Java.

Notes:

- Only the Java scalar wrappers are passed to Java.
- The ESQL scalar types are mapped to Java data types as object wrappers, or object wrapper arrays, depending upon the direction of the procedure parameter. Each wrapper array contains exactly one element.
- Scalar object wrappers are used to allow NULL values to be passed to and from Java methods.

ESQL data types ¹	Java IN data types	Java INOUT and OUT data types
INTEGER, INT	java.lang.Long	java.lang.Long []
FLOAT	java.lang.Double	java.lang.Double[]
DECIMAL	java.math.BigDecimal	java.math.BigDecimal[]
CHARACTER, CHAR	java.lang.String	java.lang.String[]
BLOB	byte[]	byte[][]
BIT	java.util.BitSet	java.util.BitSet[]
DATE	com.ibm.broker.plugin.MbDate	com.ibm.broker.plugin.MbDate[]
TIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
GMTTIME ²	com.ibm.broker.plugin.MbTime	com.ibm.broker.plugin.MbTime[]
TIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
GMTTIMESTAMP ²	com.ibm.broker.plugin.MbTimestamp	com.ibm.broker.plugin.MbTimestamp[]
INTERVAL	Not supported	Not supported
BOOLEAN	java.lang.Boolean	java.lang.Boolean[]
REFERENCE (to a message tree) ^{3 4} _{5 6}	com.ibm.broker.plugin.MbElement	com.ibm.broker.plugin.MbElement[] (Supported for INOUT. Not supported for OUT)
ROW	Not supported	Not supported

LIST	Not supported	Not supported
------	---------------	---------------

- Variables that are declared to be CONSTANT (or references to variables that are declared to be CONSTANT) are not allowed to have the direction INOUT or OUT.
- The time zone set in the Java variable is not important; you obtain the required time zone in the output ESQL.
- The reference parameter cannot be NULL when passed into a Java method.
- The reference cannot have the direction OUT when passed into a Java method.
- If an *MbElement* is passed back from Java to ESQL as an INOUT parameter, it must point to a location in the same message tree as that pointed to by the *MbElement* that was passed into the called Java method.
For example, if an ESQL reference to `OutputRoot.XML.Test` is passed into a Java method as an INOUT *MbElement*, but a different *MbElement* is passed back to ESQL when the call returns, the different element must also point to somewhere in the `OutputRoot` tree.
- An *MbElement* cannot be returned from a Java method with the RETURNS clause, because no ESQL routine can return a reference. However, an *MbElement* can be returned as an INOUT direction parameter, subject to the conditions described in point 5.

A REFERENCE to a scalar variable can be used in the CALL of a Java method, provided that the data type of the variable to which the reference refers matches the corresponding data type in the Java program signature.

Restrictions on Java routines

The following restrictions apply to Java routines that are called from ESQL:

- The Java method must be threadsafe (reentrant).
- Database connections must be JDBC type 2 or type 4. Furthermore, database operations are not part of a broker transaction and therefore cannot be controlled by an external resource coordinator (as is the case in an XA environment).
- The Java user-defined node API must be used only by the same thread that invoked the Java method.

You can create threads inside your method. However, created threads must not use the Java APIs, and you must return control back to the broker.

All restrictions that apply to the usage of the Java API also apply to Java methods that are called from ESQL.

- Java methods that are called from ESQL must not use the `MbNode` class. Therefore, they cannot create objects of type `MbNode`, or call any of the methods on an existing `MbNode` object.
- WebSphere MQ or JMS work done inside a Java method that is called from ESQL must be done in accordance with the guidelines for performing WebSphere MQ and JMS work in a user-defined node. See Planning user-defined input nodes.

Deploying Java classes

You can deploy your Java classes to a broker within a Java Archive (JAR) file, using one of the following two methods:

- Add the JAR file to the broker archive (BAR) file**

The most efficient and flexible method of deploying to the broker is to add your JAR file to the BAR file. You can do this manually or automatically using the workbench.

If the workbench finds the correct Java class inside a referenced Java project open in the workspace, it automatically compiles the Java class into a JAR file and adds it to the BAR file. This procedure is the same procedure that you follow to deploy a JavaCompute node inside a JAR, as described in *User-defined node classloading*.

When you deploy a JAR file from the workbench, the flow that has been redeployed reloads the JAR file contained in the BAR file.

The files are also reloaded if the message flow that references a Java class is stopped and restarted. Ensure that you stop and restart (or redeploy) all flows that reference the JAR file that you want to update. This action avoids the problem of some flows running with the old version of the JAR file and other flows running with the new version.

The workbench deploys only JAR files; it does not deploy stand-alone Java class files.

2. Store the JAR file in either of the following locations:

- a. The *workpath/shared-classes/* folder on the machine running the broker
- b. The CLASSPATH environment variable on the computer running the broker

You must complete this action manually; you cannot use the workbench.

In this method, redeploying the message flow does not reload the referenced Java classes; neither does stopping and restarting the message flow. The only way to reload the classes in this case is to stop and restart the broker itself.

To enable the broker to find a Java class, ensure that it is in one of the preceding locations. If the broker cannot find the specified class, it generates an exception.

Although you have the choices shown previously when you deploy the JAR file, using the workbench to deploy the BAR file provides the greatest flexibility when redeploying the JAR file.

Database routines

Database routines are implemented as database stored procedures. Database routines have a LANGUAGE clause of DATABASE, and must have a routine type of PROCEDURE.

When writing stored procedures in languages like C, you must use NULL indicators to ensure that your procedure can process the data correctly.

Although the database definitions of a stored procedure vary between the databases, the ESQL used to invoke them does not. The names given to parameters in the ESQL do not have to match the names they are given on the database side. However, the external name of the routine, including any package or container specifications, must match its defined name in the database.

The DYNAMIC RESULT SETS clause is allowed only for database routines. It is required only if a stored procedure returns one or more result sets. The integer parameter to this clause must be 0 (zero) or more, and specifies the number of result sets to be returned.

The optional RETURNS clause is required if a stored procedure returns a single scalar value.

The EXTERNAL NAME clause specifies the name by which the database knows the routine. This can be either a qualified or an unqualified name, where the qualifier is the name of the database schema in which the procedure is defined. If you do not provide a schema name, the database connection user name is used as the schema in which to locate the procedure. If the required procedure does not exist in this schema, you must provide an explicit schema name, either on the routine definition or on the CALL to the routine at run time. For more information about dynamically choosing the schema that contains the routine, see the “CALL statement” on page 196. When a qualified name is used, the name must be in quotation marks.

A fully qualified routine typically takes the form:

```
EXTERNAL NAME "mySchema.myProc";
```

However, if the procedure belongs to an Oracle package, the package is treated as part of the procedure's name. Therefore you must provide a schema name and the package name, in the form:

```
EXTERNAL NAME "mySchema.myPackage.myProc";
```

This form allows the schema, but not the package name, to be chosen dynamically in the CALL statement.

If the name of the procedure contains SQL wildcards (which are the percent (%) character and the underscore (_) character), the procedure name is modified by the broker to include the database escape character immediately before each wildcard character. This technique ensures that the database receives the wildcards as literal characters. For example, assuming that the database escape character is a backslash, the following clause is modified by the broker so that “mySchema.Proc_” is passed to the database.

```
EXTERNAL NAME "mySchema.Proc\_";
```

All external procedures have the following restrictions:

- A stored procedure cannot be overloaded on the database side. A stored procedure is considered overloaded if there is more than one procedure of the same name in the same database schema. If the broker detects that a procedure has been overloaded, it raises an exception.
- Parameters cannot be of the ESQL REFERENCE, ROW, LIST, or INTERVAL data types.
- User-defined types cannot be used as parameters or as return values.

For LANGUAGE DATABASE routines, the *ExternalRoutineName* is not optional and contains the schema name, package name and procedure name of the routine to be called. Specify the *ExternalRoutineName* like this:

```
>>--"schemaName---.---packageName---.---procedureName--"-----><
```

where:

- *schemaName* is optional.
- *packageName* is optional and applies only to Oracle data sources. If you supply a *packageName* you must supply a *schemaName*.
- *procedureName* is not optional.

Database routine example 1

The following example shows an ESQL definition of a stored procedure that returns a single scalar value and an OUT parameter:

```
CREATE PROCEDURE myProc1(IN P1 INT, OUT P2 INT)
  RETURNS INTEGER
  LANGUAGE DATABASE
  EXTERNAL NAME "myschema.myproc";
```

Use this ESQL to invoke the myProc1 routine:

```
/*using CALL statement invocation syntax*/
CALL myProc1(intVar1, intVar2) INTO intReturnVar3;

/*or using function invocation syntax*/
SET intReturnVar3 = myProc1(intVar1, intVar2);
```

Database routine example 2

The following ESQL code demonstrates how to define and call DB2 stored procedures:

```
ESQL Definition:
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
  IN parm1 CHARACTER,
  OUT parm2 CHARACTER,
  INOUT parm3 CHARACTER
)
  LANGUAGE DATABASE
  EXTERNAL NAME dbSwapParms;
```

To register this stored procedure with DB2, copy the following script to a file (for example, test1.sql)

```
-- DB2 Example Stored Procedure
DROP PROCEDURE dbSwapParms @
CREATE PROCEDURE dbSwapParms
( IN in_param CHAR(32),
  OUT out_param CHAR(32),
  INOUT inout_param CHAR(32))
  LANGUAGE SQL
  BEGIN
  SET out_param = inout_param;
  SET inout_param = in_param;
  END @
```

Now run the file from the DB2 command prompt:

```
db2 -td@ -vf test1.sql
```

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes "World".
- The value of the INOUT parameter changes to "Hello".

Database routine example 3

The following ESQL code demonstrates how to define and call Oracle stored procedures:

```
ESQL Definition:
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
    IN parm1 CHARACTER,
    OUT parm2 CHARACTER,
    INOUT parm3 CHARACTER
)
LANGUAGE DATABASE
EXTERNAL NAME dbSwapParms;
```

To register this stored procedure with Oracle, copy the following script to a file (for example, test1.sql)

```
CREATE OR REPLACE PROCEDURE dbSwapParms
( in_param IN VARCHAR2,
  out_param OUT VARCHAR2,
  inout_param IN OUT VARCHAR2 )
AS
BEGIN
    out_param := inout_param;
    inout_param := in_param;
END;
/
```

Now run the file:

```
sqlplus userID/password @test1.sql
```

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes “World”.
- The value of the INOUT parameter changes to “Hello”.

Database routine example 4

The following ESQL code demonstrates how to define and call SQL Server stored procedures:

```
ESQL Definition:
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
    IN parm1 CHARACTER,
    INOUT parm2 CHARACTER,
    INOUT parm3 CHARACTER
)
LANGUAGE DATABASE
EXTERNAL NAME dbSwapParms;
```


To register this stored procedure with SQL Server, copy the following script to a file (for example, test1.sql)

```
-- SQLServer Example Stored Procedure
DROP PROCEDURE dbSwapParms
go
CREATE PROCEDURE dbSwapParms
  @in_param    CHAR(32),
  @out_param   CHAR(32) OUT,
  @inout_param CHAR(32) OUT
AS
  SET NOCOUNT ON
  SET @out_param = @inout_param
  SET @inout_param = @in_param
go
```

Now run file:

```
isql -UserID -Ppassword -Sserver -ddatasource -itest1.sql
```

SQL Server considers OUTPUT parameters from stored procedures as INPUT/OUTPUT parameters. If you declare these as OUT parameters in your ESQL you encounter a type mismatch error at run time. To avoid that mismatch you must declare SQL Server OUTPUT parameters as INOUT in your ESQL.

Use the SET NOCOUNT ON option, as shown in the preceding example, with SQL stored procedures for the following reasons:

1. To limit the amount of data returned from SQL Server to the broker.
2. To allow result sets to be returned correctly.

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes "World".
- The value of the INOUT parameter changes to "Hello".

Database routine example 5

The following ESQL code demonstrates how to define and call Sybase stored procedures:

```
ESQL Definition:
DECLARE inputParm CHARACTER;
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER;

SET inputParm = 'Hello';
SET inputOutputParm = 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
  IN parm1 CHARACTER,
  INOUT parm2 CHARACTER,
  INOUT parm3 CHARACTER
)
LANGUAGE DATABASE
EXTERNAL NAME dbSwapParms;
```

To register this stored procedure with Sybase, copy the following script to a file (for example, test1.sql)

```
-- SYBASE Example Stored Procedure
DROP PROCEDURE dbSwapParms
go
CREATE PROCEDURE dbSwapParms
```

```

@in_param    CHAR(32),
@out_param   CHAR(32) OUT,
@inout_param CHAR(32) OUT
AS
  SET @out_param = @inout_param
  SET @inout_param = @in_param
go

```

Now run file:

```
isql -U<userID> -P<password> -S<server> -D<datasource> -itest1.sql
```

Sybase considers OUTPUT parameters from stored procedures as INPUT/OUTPUT parameters. If you declare these as OUT parameters in your ESQL, you encounter a type mismatch error at run time. To avoid that mismatch, declare Sybase OUTPUT parameters as INOUT in your ESQL.

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.
- The value of the OUT parameter becomes "World".
- The value of the INOUT parameter changes to "Hello".

Database routine example 6

The following ESQL code demonstrates how to define and call Informix stored procedures:

```

ESQL Definition:
DECLARE inputParm CHARACTER 'Hello';
DECLARE outputParm CHARACTER;
DECLARE inputOutputParm CHARACTER 'World';
CALL swapParms( inputParm, outputParm, inputOutputParm );

CREATE PROCEDURE swapParms (
  IN parm1 CHARACTER,
  INOUT parm2 CHARACTER,
  INOUT parm3 CHARACTER
)
LANGUAGE DATABASE
EXTERNAL NAME dbSwapParms;

```

To register this stored procedure with Informix, copy the following script to a file (for example, test1.sql)

```

DROP SPECIFIC PROCEDURE dbSwapParms;
CREATE PROCEDURE dbSwapParms
(
  inParm    CHAR(20),
  OUT      outParm    CHAR(20),
  INOUT    inoutParm  CHAR(20))

SPECIFIC dbSwapParms

  LET outParm = inoutParm;
  LET inoutParm = inParm;
END PROCEDURE;

```

Now run file:

From the Informix server shell environment enter:

```
dbaccess <dataBaseName> <fully qualified path/test1.sql>
```

Expect the following results from running this code:

- The value of the IN parameter does not (and cannot, by definition) change.

- The value of the OUT parameter becomes "World".
- The value of the INOUT parameter changes to "Hello".

The following restrictions apply to Informix stored procedures:

- Procedures that use the Informix INTERVAL datatype cannot be invoked from the broker.
- Procedures can return only one result set.
- Procedures that return a result set must only have IN parameters.
- Procedures cannot return CLOBs or BLOBs in result sets or as scalar return values.
- Procedures can return either a result set or a scalar value, but not both.

Database routine example 7

This example shows how to call a stored procedure that returns two result sets, in addition to an out parameter:

```
CREATE PROCEDURE myProc1 (IN P1 INT, OUT P2 INT)
LANGUAGE DATABASE
DYNAMIC RESULT SETS 2
EXTERNAL NAME "myschema.myproc";
```

Use the following ESQL to invoke myProc1:

```
/* using a field reference */
CALL myProc1(intVar1, intVar2, Environment.RetVal [], OutputRoot.XMLNS.A [])
/* using a reference variable*/
CALL myProc1(intVar1, intVar2, myReferenceVariable.RetVal [], myRef2.B [])
```

DECLARE statement

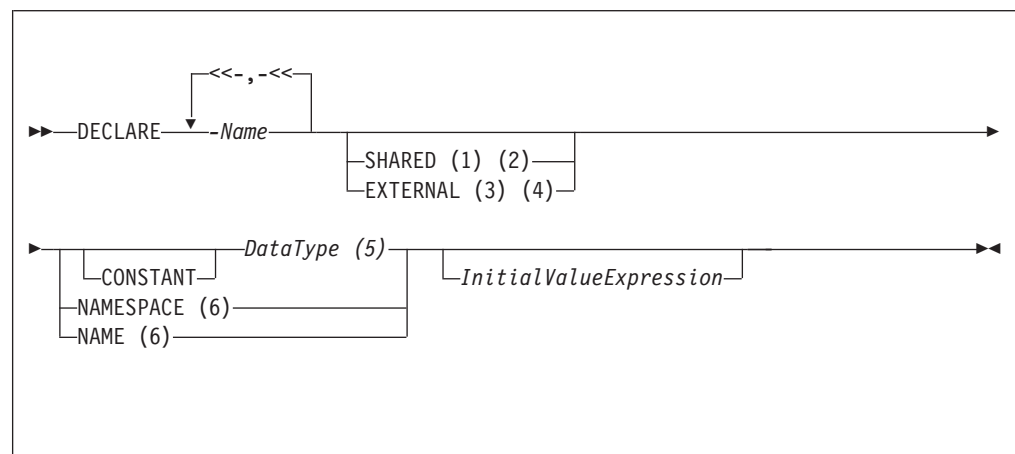
Use the DECLARE statement to define a variable, the data type of the variable and, optionally, its initial value.

You can define three types of variable with the DECLARE statement:

- External
- Normal
- Shared

For further information, see "ESQL variables" on page 173.

Syntax



Notes:

1. The SHARED keyword is not valid within a function or procedure.
2. You cannot specify SHARED with a *DataType* of REFERENCE TO. To store a message tree in a shared variable, use the ROW data type.
3. EXTERNAL variables are implicitly constant.
4. It is good programming practice to give an EXTERNAL variable an initial value.
5. If you specify a *DataType* of REFERENCE TO, you must specify an initial value (of either a variable or a tree) in *InitialValueExpression*.
6. When you use the NAMESPACE and NAME clauses, their values are implicitly constant and of type CHARACTER.

Follow the links to see more information about all these parameters:

- “CONSTANT”
- “DataType”
- “EXTERNAL” on page 237
- “NAME” on page 238
- “NAMESPACE” on page 238
- “SHARED” on page 239

CONSTANT

Use CONSTANT to define a constant. You can declare constants within schemas, modules, routines, or compound statements (both implicit and explicit). The behavior of these cases is shown in the following list:

- Within a compound statement, constants and variables occupy the same namespace.
- Within expressions, a constant or variable that is declared within a compound statement overlays all constants and variables of the same name that are declared in containing compound statements, modules, and schemas.
- Within field reference namespace fields, a namespace constant that is declared within a compound statement overlays all namespace constants of the same name that are declared in containing compound statements.

A constant or variable that is declared within a routine overlays any parameters of the same name, as well as all constants and variables of the same name that are declared in a containing module or schema.

DataType

The values that you can specify for *DataType* are:

- BOOLEAN
- INT
- INTEGER
- FLOAT
- DECIMAL
- DATE
- TIME
- TIMESTAMP
- GMTTIME
- GMTTIMESTAMP

- INTERVAL. This value does not apply to external variables (EXTERNAL keyword specified).
- CHAR
- CHARACTER
- BLOB
- BIT
- ROW. This value does not apply to external variables (EXTERNAL keyword specified).
- REFERENCE TO. This value does not apply to external or shared variables (EXTERNAL or SHARED keyword specified).

EXTERNAL

Use EXTERNAL to denote a user-defined property (UDP). A UDP is a user-defined constant whose initial value (optionally set by the DECLARE statement) can be modified, at design time, by the Message Flow editor (see Message Flow editor) or overridden, at deployment time, by the Broker Archive editor (see Broker Archive editor). The value of a UDP cannot be modified by ESQL.

When a UDP is given an initial value on the DECLARE statement, this value becomes its default. However, any value that you specify in the Message Flow editor at design time, or in the Broker Archive editor at deployment time (even a zero length string) overrides any initial value that was coded on the DECLARE statement.

For example, if you code:

```
DECLARE deployEnvironment EXTERNAL CHARACTER 'Dev';
```

you have defined a UDP variable of `deployEnvironment` with an initial value `Dev`.

Add the UDP to the message flow by using the UDP tab in the message flow editor. When you add the flow to the BAR file, the UDP is there as an attribute of the flow; you must name the attribute to be the same as the ESQL variable in the DECLARE statement (in this case, `deployEnvironment`) to ensure that the initial value that you set is unchanged.

All the UDPs in a message flow must have a value, given either on the DECLARE statement or by the Message Flow or Broker Archive editor; otherwise a deployment-time error occurs. At run time, after the UDP has been declared, its value can be queried by subsequent ESQL statements.

The advantage of UDPs is that their values can be changed at deployment time. For example, if you use the UDPs to hold configuration data, it means that you can configure a message flow for a particular computer, task, or environment at deployment time, without having to change the code at the node level. UDPs can also be modified at run time by using the CMP API.

You can declare UDPs only in modules or schemas; that is, you can use the DECLARE statement with the EXTERNAL keyword only at the MODULE or SCHEMA level. If you use a DECLARE statement with the EXTERNAL keyword within a PROCEDURE or FUNCTION, a BIP2402E exception occurs when you deploy the message flow.

The following types of broker node can access UDPs:

- Compute node
- Database node

- Filter node
- Nodes that are derived from these node types

Take care when specifying the data type of a UDP, because a CAST is used to change the value to the requested *DataType*.

For an overview of UDPs, see “User-defined properties in ESQL” on page 6.

Example 1:

```
DECLARE mycolor EXTERNAL CHARACTER 'blue';
```

Example 2:

```
DECLARE TODAYSCOLOR EXTERNAL CHARACTER;
SET COLOR = TODAYSCOLOR;
```

where TODAYSCOLOR is a user-defined property that has a TYPE of CHARACTER and a VALUE set by the Message Flow editor.

NAME

Use NAME to define an alias (an alternative name) by which a variable can be known.

Example 1:

```
-- The following statement gives Schema1 an alias of 'Joe'.
DECLARE Schema1 NAME 'Joe';
-- The following statement produces a field called 'Joe'.
SET OutputRoot.XMLNS.Data.Schema1 = 42;

-- The following statement inserts a value into a table called Table1
-- in the schema called 'Joe'.
INSERT INTO Database.Schema1.Table1 (Answer) VALUES 42;
```

Example 2:

```
-- At Module scope define ColourElementName and set it external
-- so that its default value of 'black' can be overridden as a UDP
DECLARE ColourElementName EXTERNAL NAME 'black';

-- Use the ColourElementName in a function
CREATE FIRSTCHILD OF OutputRoot.XMLNSC.TestCase.ColourElementName
    Domain('XMLNSC')
    NAME 'Node1' VALUE '1';
```

If the owning message flow has been configured with a UDP named ColourElementName of type String, which has been given the value red, the following output message is generated:

```
<xml version="1.0"?>
<TestCase>
  <red>
    <Node1>1</Node1>
  </red>
```

NAMESPACE

Use NAMESPACE to define an alias (an alternative name) by which a namespace can be known.

Example:

This example illustrates a namespace declaration, its use as a *SpaceId* in a path, and its use as a character constant in a namespace expression:

```
DECLARE prefixOne NAMESPACE 'http://www.example.com/P01';

-- On the right hand side of the assignment a namespace constant
-- is being used as such while, on the left hand side, one is
-- being used as an ordinary constant (that is, in an expression).

SET OutputRoot.XMLNS.{prefixOne}:{'PurchaseOrder'} =
    InputRoot.XMLNS.prefixOne:PurchaseOrder;
```

SHARED

Use SHARED to define a shared variable. Shared variables are private to the flow (if declared within a schema) or node (if declared within a module), but are shared between instances of the flow (threads). No type of variable is visible beyond the flow level; for example, you cannot share variables across execution groups.

You can use shared variables to implement an in-memory cache in the message flow; see *Optimizing message flow response times*. Shared variables have a long lifetime and are visible to multiple messages passing through a flow; see “Long-lived variables” on page 7.

Shared variables exist for the lifetime of the:

- Execution group process
- Flow or node, or
- Node ESQL code that declares the variable

(whichever is the shortest). They are initialized when the first message passes through the flow or node after each broker startup.

You cannot define a shared variable within a function or procedure.

The advantages of shared variables, relative to databases, are that:

- Write access is much faster.
- Read access to small data structures is faster.
- Access is direct; that is, there is no need to use a special function (SELECT) to get data, or special statements (INSERT, UPDATE, or DELETE) to modify data. You can refer to the data directly in expressions.

The advantages of databases, relative to shared variables, are that:

- The data is persistent.
- The data is changed transactionally.

These read-write variables are ideal for users who are prepared to sacrifice the persistence and transactional advantages of databases in order to obtain better performance, because they have a longer life than only one message and perform better than a database.

Because shared variables can be updated by multiple additional instances, you must ensure that you do not change shared variables that might cause unexpected results, for example, if the variable is being used as a counter.

You can prevent other instances seeing the intermediate stages of the data by using a BEGIN ATOMIC construct; see “BEGIN ... END statement” on page 191.

Your user program can make an efficient read, or write, copy of an input message in the input node by using shared-row variables, which simplifies the technique for handling large messages.

Restriction:

Subtrees cannot be copied directly from one shared row variable to another shared row variable. Subtrees can be copied indirectly by using a non-shared row variable. Scalar values extracted from one shared row variable (by using the FIELDVALUE function) can be copied to another shared row variable.

Sample program

The following sample shows how to use both shared and external variables:

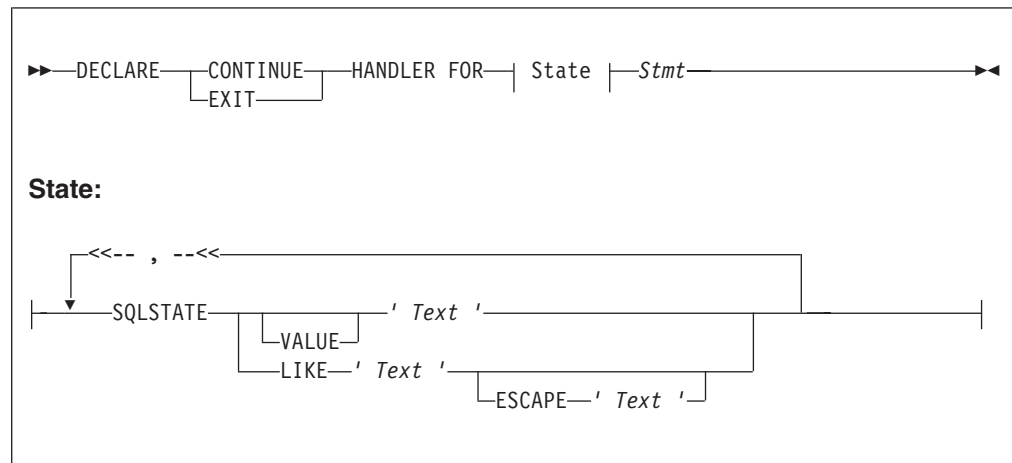
- Message Routing

You can view samples only when you use the information center that is integrated with the Message Broker Toolkit.

DECLARE HANDLER statement

The DECLARE HANDLER statement creates an error handler for handling exceptions.

Syntax



You can declare handlers in both explicitly declared (BEGIN...END) scopes and implicitly declared scopes (for example, the ELSE clause of an IF statement). However, all handler declarations must be together at the top of the scope, before any other statements.

If there are no exceptions, the presence of handlers has no effect on the behavior or performance of an SQL program. If an exception occurs, WebSphere Message Broker compares the SQL state of the exception with the SQL states associated with any relevant handlers, until either the exception leaves the node (just as it would if there were no handlers) or a matching handler is found. Within any one scope,

handlers are searched in the order they are declared; that is, first to last. Scopes are searched from the innermost to outermost.

The SQL state values provided in DECLARE... HANDLER... statements can be compared directly with the SQL state of the exception or can be compared using wildcard characters. To compare the state values directly, specify either VALUE or no condition operator. To make a wildcard comparison, use the underscore and percent characters to represent single and multiple character wildcards, respectively, and specify the LIKE operator. The wildcard method allows all exceptions of a general type to be handled without having to list them exhaustively.

If a matching handler is found, the SQLSTATE and other special registers are updated (according to the rules described below) and the handler's statement is processed.

As the handler's statement must be a single statement, it is typically a compound statement (such as BEGIN...END) that contains multiple other statements. There is no special behavior associated with these inner statements and there are no special restrictions. They can, for example, include RETURN, ITERATE, or LEAVE; these affect their containing routines and looping constructs in the same way as if they were contained in the scope itself.

Handlers can contain handlers for exceptions occurring within the handler itself

If processing of the handler's code completes without throwing further unhandled exceptions, execution of the normal code is resumed as follows:

- For EXIT handlers, the next statement processed is the first statement after the handler's scope.
- For CONTINUE handlers, it is the first directly-contained statement after the one that produced the exception.

Each handler has its own SQLCODE, SQLSTATE, SQLNATIVEERROR, and SQLERRORTXT special registers. These come into scope and their values are set just before the handler's first statement is executed. They remain valid until the handler's last statement has been executed. Because there is no carry over of SQLSTATE values from one handler to another, handlers can be written independently.

Handlers absorb exceptions, preventing their reaching the input node and thus causing the transaction to be committed rather than rolled back. A handler can use a RESIGNAL or THROW statement to prevent this.

See "SQLSTATE function" on page 281 for a list of the valid SQLSTATES that you can use in a HANDLER.

Example 1

The following example demonstrates proper use of a USER EXCEPTION, and passing SQLCODE, SQLSTATE, SQLNATIVEERROR and SQLERRORTXT from the exception to the handler. The example also uses the SQLSTATE to catch the specific exception:

```
DECLARE retryCount INTEGER 0;  
DECLARE afterCount INTEGER 0;
```

```
WHILE retryCount <= 10 DO
```

```

DECLARE EXIT HANDLER FOR SQLSTATE VALUE 'U11222'
BEGIN
  /* This demonstrates how to pass data to the HANDLER in the SQL
     special registers */
  SET OutputRoot.XMLNSC.Top.WHILE.mySQLCODE = SQLCODE;
  SET OutputRoot.XMLNSC.Top.WHILE.mySQLSTATE = SQLSTATE;
  SET OutputRoot.XMLNSC.Top.WHILE.mySQLNATIVEERROR = SQLNATIVEERROR;
  SET OutputRoot.XMLNSC.Top.WHILE.mySQLERRORTTEXT = SQLERRORTTEXT;

  SET retryCount = retryCount + 1;

  /* If we are an EXIT HANDLER, control is now passed to back to the
     WHILE statement */
END;

/* In a real scenario this could be a PROPAGATE statement, and the exception
   could be thrown by a 'downstream' node. In this case the HANDLER would
   normally cope with a wider range of exception, for example, using LIKE '%' */
THROW USER EXCEPTION VALUES( -1, 'U11222', 42, 'error text' );

/* This is the next statement executed if it is a CONTINUE HANDLER */
SET afterCount = afterCount + 1;

END WHILE;

SET OutputRoot.XMLNSC.Top.WHILE.retryCount = retryCount;
SET OutputRoot.XMLNSC.Top.WHILE.afterCount = afterCount;

```

With EXIT (as above) the output is:

```

<Top>
  <WHILE>
    <mySQLCODE>-1</mySQLCODE>
    <mySQLSTATE>U11222</mySQLSTATE>
    <mySQLNATIVEERROR>42</mySQLNATIVEERROR>
    <mySQLERRORTTEXT>error text</mySQLERRORTTEXT>
    <retryCount>11</retryCOUNT>
    <afterCount>0</afterCOUNT>
  </WHILE>
</Top>

```

Changing the HANDLER to be CONTINUE (DECLARE CONTINUE HANDLER FOR SQLSTATE VALUE 'U11222') then the output is:

```

<Top>
  <WHILE>
    <mySQLCODE>-1</mySQLCODE>
    <mySQLSTATE>U11222</mySQLSTATE>
    <mySQLNATIVEERROR>42</mySQLNATIVEERROR>
    <mySQLERRORTTEXT>error text</mySQLERRORTTEXT>
    <retryCount>11</retryCOUNT>
    <afterCount>11</afterCOUNT>
  </WHILE>
</Top>

```

You see the difference in afterCount in the output message.

Example 2

```

-- Drop the tables so that they can be recreated with the latest definition.
-- If the program has never been run before, errors will occur because you
-- can't drop tables that don't exist. We ignore these.
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE LIKE '%' BEGIN END;

  PASSTHRU 'DROP TABLE Shop.Customers' TO Database.DSN1;

```

```

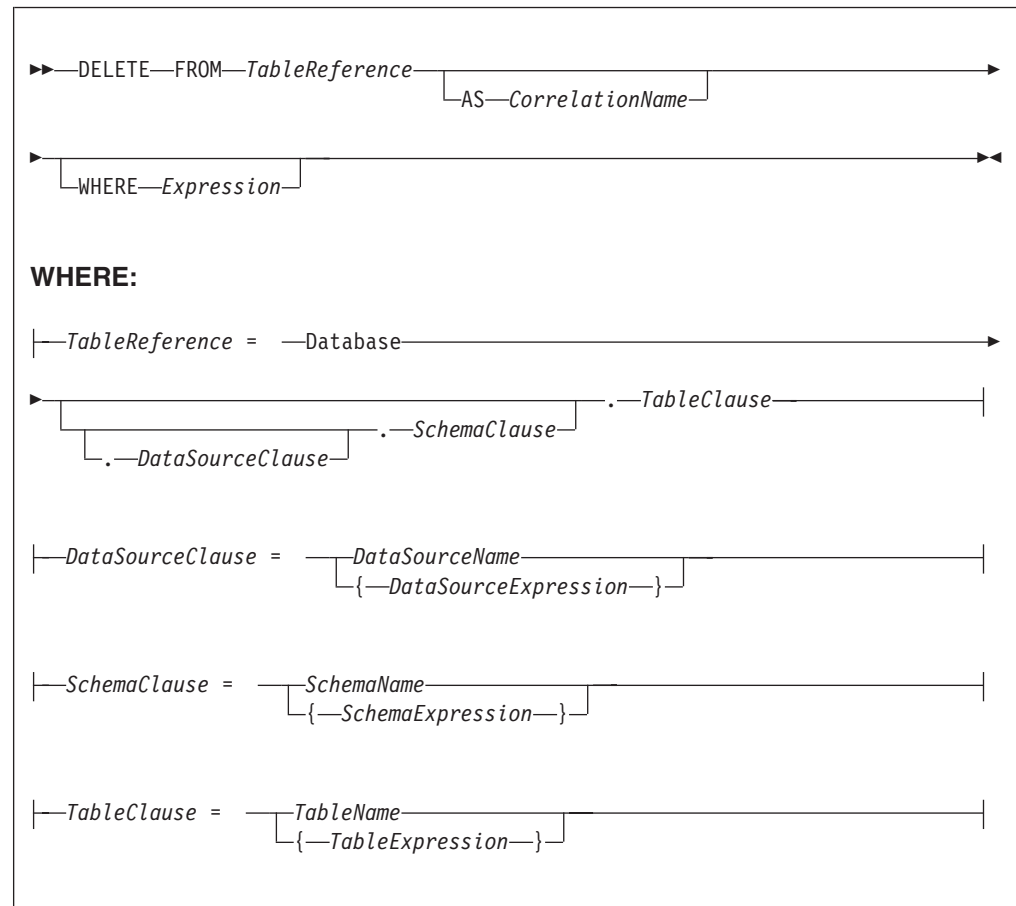
PASSTHRU 'DROP TABLE Shop.Invoices' TO Database.DSN1;
PASSTHRU 'DROP TABLE Shop.Sales' TO Database.DSN1;
PASSTHRU 'DROP TABLE Shop.Parts' TO Database.DSN1;
END;

```

DELETE FROM statement

The DELETE FROM statement deletes rows from a table in an external database, based on a search condition.

Syntax



All rows for which the WHERE clause expression evaluates to TRUE are deleted from the table identified by *TableReference*.

Each row is examined in turn and a variable is set to point to the current row. Typically, the WHERE clause expression uses this variable to access column values and thus cause rows to be retained or deleted according to their contents. The variable is referred to by *CorrelationName* or, in the absence of an AS clause, by *TableName*.

Table reference

A table reference is a special case of the field references that are used to refer to message trees. It always starts with the word "Database" and can contain any of the following elements:

- A table name only

- A schema name and a table name
- A data source name (that is, the name of a database instance), a schema name, and a table name

In each case, the name can be specified directly or by an expression enclosed in braces ({...}). A directly-specified data source, schema, or table name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see “DECLARE statement” on page 235).

If a schema name is not specified, the default schema for the broker's database user is used.

If a data source name is not specified, the database pointed to by the node's data source attribute is used.

The WHERE clause

The WHERE clause expression can use any of the broker's operators and functions in any combination. It can refer to table columns, message fields, and any declared variables or constants.

However, be aware that the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be evaluated by the database. If it can, it is given to the database. In order to be evaluated by the database, it must use only those functions and operators supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECT functions, and to any other declared variables or constants within scope.

If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each sub-expression separately. It then attempts to give the database those sub-expressions that it can evaluate, leaving the broker to evaluate the rest. You need to be aware of this situation for two reasons:

1. Apparently trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.
2. Some databases' functions exhibit subtle differences of behavior from those of the broker.

Handling errors

It is possible for errors to occur during delete operations. For example, the database might not be operational. In these cases, an exception is thrown (unless the node has its throw exception on database error property set to FALSE). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the DECLARE HANDLER statement).

For further information about handling database errors, see “Capturing database state” on page 84.

Examples

The following example assumes that the `dataSource` property has been configured and that the database it identifies has a table called `SHAREHOLDINGS`, with a column called `ACCOUNTNO`.

```
DELETE FROM Database.SHAREHOLDINGS AS S
      WHERE S.ACCOUNTNO = InputBody.AccountNumber;
```

This example removes all the rows from the `SHAREHOLDINGS` table where the value in the `ACCOUNTNO` column (in the table) is equal to that in the `AccountNumber` field in the message. This operation might delete zero, one, or more rows from the table.

The next example shows the use of calculated data source, schema, and table names:

```
-- Declare variables to hold the data source, schema, and table names and
-- set their default values
DECLARE Source CHARACTER 'Production';
DECLARE Schema CHARACTER 'db2admin';
DECLARE Table CHARACTER 'DynamicTable1';

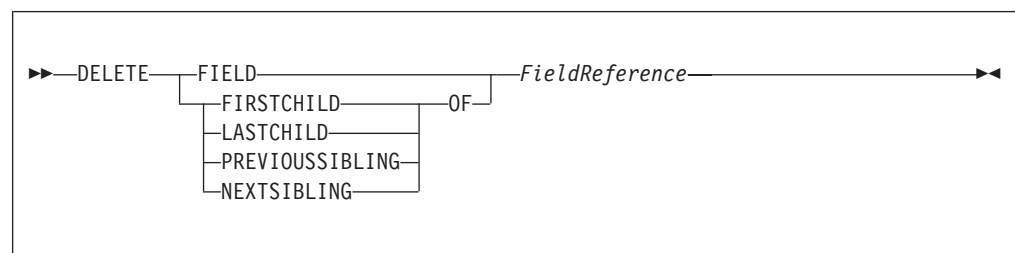
-- Code which calculates their actual values comes here

-- Delete rows from the table
DELETE FROM Database.{Source}.{Schema}.{Table} As R WHERE R.Name = 'Joe';
```

DELETE statement

The `DELETE` statement detaches and destroys a portion of a message tree, allowing its memory to be reused. This statement is particularly useful when handling very large messages.

Syntax



If the target field does not exist, the statement does nothing and normal processing continues. If any reference variables point into the deleted portion, they are disconnected from the tree so that no action involving them has any effect, and the `LASTMOVE` function returns `FALSE`. Disconnected reference variables can be reconnected by using a `MOVE... TO...` statement.

Example

```
DELETE FIELD OutputRoot.XMLNS.Data.Folder1.Folder12;
DELETE LASTCHILD OF Cursor;
```

DETACH statement

The DETACH statement detaches a portion of a message tree without deleting it. This portion can be reattached using the ATTACH statement.

Syntax

```
▶—DETACH—dynamic_reference—▶
```

For information about dynamic references, see “Creating dynamic field references” on page 36.

For an example of DETACH, see the example in “ATTACH statement” on page 189.

EVAL statement

The EVAL statement takes a character value, interprets it as an SQL statement, and processes that statement.

For details of the EVAL function, see “EVAL function” on page 372.

Syntax

```
▶—EVAL—( SQL_character_value )—▶
```

EVAL takes one parameter in the form of an expression, evaluates this expression, and casts the resulting value to a character string if it is not one already. The expression that is passed to EVAL must therefore be able to be represented as a character string.

After this first stage evaluation is complete, the behavior of EVAL depends on whether it is being used as a complete ESQL statement, or in place of an expression that forms part of an ESQL statement:

- If it is a complete ESQL statement, the character string derived from the first stage evaluation is processed as if it were an ESQL statement.
- If it is an expression that forms part of an ESQL statement, the character string is evaluated as if it were an ESQL expression and EVAL returns the result.

User defined procedures cannot be defined within an EVAL statement but EVAL can be used to call a user-defined procedure that is in scope where the EVAL statement is used.

If you use the EVAL statement to call out to a user-defined procedure that is not called from anywhere else in the ESQL for a given node, you need to add the following code to your ESQL, to ensure that the user-defined procedure being called is actually included when the code is compiled:

```
IF (FALSE) THEN CALL procedure(<parameters>); END IF;
```

Note, that in the preceding code, you must replace `procedure()` with the named procedure in question.

In the following examples, A and B are integer scalar variables, and `scalarVar1` and `OperatorAsString` are character string scalar variables.

The following examples are valid uses of EVAL:

- `SET OutputRoot.XMLNS.Data.Result = EVAL(A+B);`

The expression `A+B` is acceptable because, although it returns an integer value, integer values are representable as character strings, and the necessary cast is performed before EVAL continues with its second stage of evaluation.

- `SET OutputRoot.XMLNS.Data.Result = EVAL('A' || operatorAsString || 'B');`

- `EVAL('SET ' || scalarVar1 || ' = 2;');`

The semicolon included at the end of the final string literal is necessary, because if EVAL is being used in place of an ESQL statement, its first stage evaluation must return a string that represents a valid ESQL statement, including the terminating semicolon.

Variables declared in an EVAL statement do not exist outside that EVAL statement.

The real power of EVAL is that it allows you to dynamically construct ESQL statements or expressions. In the second and third examples above, the value of `scalarVar1` or `operatorAsString` can be set according to the value of an incoming message field, or other dynamic value, allowing you to effectively control what ESQL is processed without requiring a potentially lengthy IF-THEN ladder.

However, consider the performance implications in using EVAL. Dynamic construction and processing of statements or expressions is necessarily more time-consuming than simply processing pre-constructed ones. If performance is vital, you might prefer to write more specific, but faster, ESQL.

The following are not valid uses of EVAL:

- `SET EVAL(scalarVar1) = 2;`

In this example, EVAL is being used to replace a field reference, not an expression.

- `SET OutputRoot.XMLNS.Data.Result[] = EVAL((SELECT T.x FROM Database.y AS T));`

In this example, the `(SELECT T.x FROM Database.y)` passed to EVAL returns a list, which is not representable as a character string.

The following example is acceptable because `(SELECT T.x FROM Database.y AS T)` is a character string literal, not an expression in itself, and therefore is representable as a character string.

```
SET OutputRoot.XMLNS.Data.Result[]
  = EVAL('(SELECT T.x FROM Database.y AS T)');
```

Functions that are referenced only in an EVAL statement, and not in the rest of the ESQL module, might not be included in the BAR file. In the following examples, the function `MyFunction` must be referenced somewhere else in the ESQL module, otherwise the BAR file might fail to deploy.

```
EVAL('CALL MyFunction(parm1, parm2);');
```

```

| DECLARE functionName CHARACTER 'Function';
| DECLARE callStmt CHARACTER 'CALL My' || functionName || '(parm1, parm2)';
| EVAL(callStmt);

```

FOR statement

The FOR statement iterates through a list (for example, a message array).

Syntax

```

▶▶—FOR—correlation_name—AS—field_reference—DO—statements—END—FOR—◀◀

```

For each iteration, the FOR statement makes the correlation variable (*correlation_name* in the syntax diagram) equal to the current member of the list (*field_reference*), then executes the block of statements. The advantage of the FOR statement is that it iterates through a list without your having to write any sort of loop construct (and eliminates the possibility of infinite loops).

For example the following ESQL:

```

SET OutputRoot.MQMD=InputRoot.MQMD;

SET Environment.SourceData.Folder[1].Field1 = 'Field1Value';
SET Environment.SourceData.Folder[1].Field2 = 'Field12Value';
SET Environment.SourceData.Folder[2].Field1 = 'Field21Value';
SET Environment.SourceData.Folder[2].Field2 = 'Field22Value';

DECLARE i INTEGER 1;
FOR source AS Environment.SourceData.Folder[] DO
    CREATE LASTCHILD OF OutputRoot.XMLNSC.Data.ResultData.MessageArrayTest.Folder[i]
        NAME 'FieldA' VALUE '\ ' || source.Field1 || '\ ' || CAST(i AS CHAR);

    CREATE LASTCHILD OF OutputRoot.XMLNSC.Data.ResultData.MessageArrayTest.Folder[i]
        NAME 'FieldB' VALUE '\ ' || source.Field2 || '\ ' || CAST(i AS CHAR);
    SET i = i + 1;
END FOR;

```

generates the output message:

```

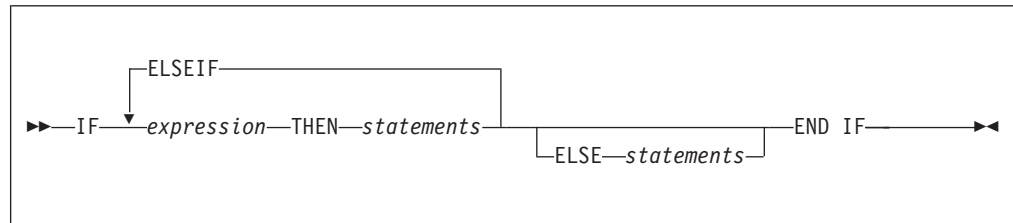
<Data>
  <ResultData>
    <MessageArrayTest>
      <Folder>
        <FieldA>Field1Value\1</FieldA>
        <FieldB>Field12Value\1</FieldB>
      </Folder>
      <Folder>
        <FieldA>Field21Value\2</FieldA>
        <FieldB>Field22Value\2</FieldB>
      </Folder>
    </MessageArrayTest>
  </ResultData>
</Data>

```

IF statement

The IF statement executes one set of statements based on the result of evaluating condition expressions.

Syntax



Each expression is evaluated in turn until one results in TRUE; the corresponding set of statements is then executed. If none of the expressions returns TRUE, and the optional ELSE clause is present, the ELSE clause's statements are executed.

UNKNOWN and FALSE are treated the same: the next condition expression is evaluated. ELSEIF is one word with no space between the ELSE and the IF. However, you can nest an IF statement within an ELSE clause: if you do, you can terminate both statements with END IF.

Example

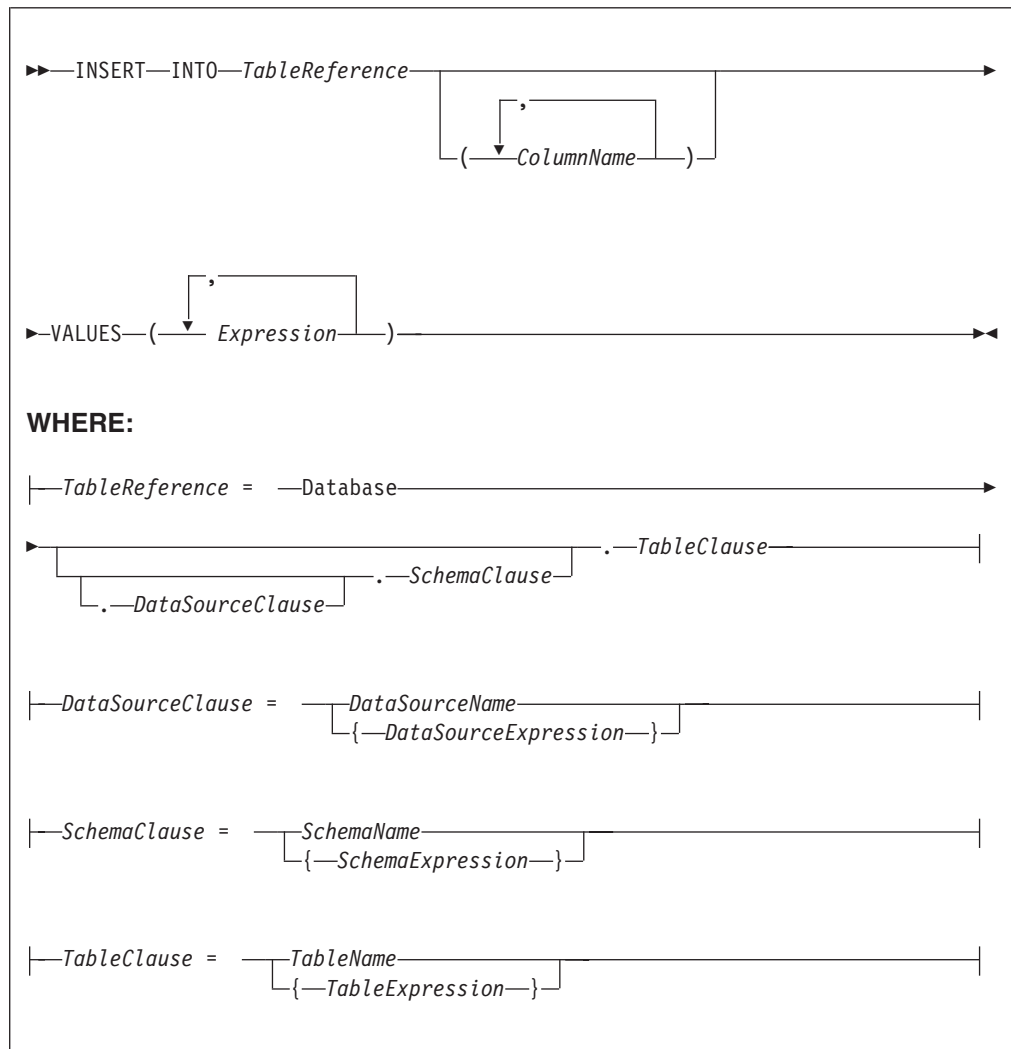
```
IF i = 0 THEN
    SET size = 'small';
ELSEIF i = 1 THEN
    SET size = 'medium';
ELSEIF j = 4 THEN
    SET size = 'large';
ELSE
    SET size = 'unknown';
END IF;
```

```
IF J > MAX THEN
    SET J = MAX;
    SET Limit = TRUE;
END IF;
```

INSERT statement

The INSERT statement inserts a row into a database table.

Syntax



A single row is inserted into the table identified by *TableReference*. The *ColumnName* list identifies those columns in the target table that are to be given specific values. These values are determined by the expressions within the VALUES clause (the first expression gives the value of the first named column, and so on). The number of expressions in the VALUES clause must be the same as the number of named columns. Any columns present in the table but not mentioned in the list are given their default values.

Table reference

A table reference is a special case of the field references that are used to refer to message trees. It always starts with the word "Database" and can contain any of the following elements:

- A table name only
- A schema name and a table name
- A data source name (that is, the name of a database instance), a schema name, and a table name

In each case, the name can be specified directly or by an expression enclosed in braces ({...}). A directly-specified data source, schema, or table name is subject to

name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see “DECLARE statement” on page 235).

If a schema name is not specified, the default schema for the broker's database user is used.

If a data source name is not specified, the database pointed to by the node's data source attribute is used.

Handling errors

Errors can occur during insert operations. For example, the database might not be operational, or the table might have constraints defined that the new row would violate. In these cases, an exception is thrown, unless you have cleared the node property `Throw Exception on Database Error`. These exceptions set appropriate values for the following items, and can be dealt with by error handlers (see the `DECLARE HANDLER` statement):

- SQL code
- State
- Native error
- Error text

For further information about handling database errors, see “Capturing database state” on page 84.

Examples

The following example assumes that the Data Source property of the Database node has been configured, and that the database it identifies has a table called `TABLE1` with columns `A`, `B`, and `C`.

Given a message with the following generic XML body:

```
<A>
  <B>1</B>
  <C>2</C>
  <D>3</D>
</A>
```

The following `INSERT` statement inserts a new row into the table with the values 1, 2, and 3 for the columns `A`, `B`, and `C`:

```
INSERT INTO Database.TABLE1(A, B, C) VALUES (Body.A.B, Body.A.C, Body.A.D);
```

The next example shows the use of calculated data source, schema, and table names:

```
-- Declare variables to hold the data source, schema, and table names
-- and set their default values
DECLARE Source CHARACTER 'Production';
DECLARE Schema CHARACTER 'db2admin';
DECLARE Table CHARACTER 'DynamicTable1';

-- Code which calculates their actual values comes here

-- Insert the data into the table
INSERT INTO Database.{Source}.{Schema}.{Table} (Name, Value) values ('Joe', 12.34);
```

Inserting a bitstream into a database

If the database column into which you want to insert data is set to a binary data type such as BLOB, the input message must be represented in bitstream form. If the input message is in the BLOB domain, use the following ESQL code:

```
DECLARE msgBitStream BLOB InputRoot.BLOB.BLOB;
INSERT INTO Database.TABLE1(MSGDATA) VALUES (msgBitStream);
```

Alternatively, if the input message is in an XML domain such as XMLNS, then the message tree must be serialized before the INSERT statement. To serialize the message tree and insert the contents into the database, use the following ESQL code:

```
DECLARE propRef REFERENCE TO InputRoot.Properties;
DECLARE msgBitStream BLOB ASBITSTREAM(InputRoot.XMLNS, propRef.Encoding, propRef.CodedCharSetId);
INSERT INTO Database.TABLE1(MSGDATA) VALUES (msgBitStream);
```

If the input messages received by your message flow come from different code pages, the CodedCharSetID and Encoding information is lost if you use the previous example. To capture CodedCharSetID and Encoding information, you can extend the table with two numeric columns to store the CodedCharSetID and Encoding data. To extend the table, modify the ESQL from the previous example to insert the CodedCharSetID and Encoding data into separate database columns:

```
DECLARE propRef REFERENCE TO InputRoot.Properties;
DECLARE inCCSID INT propRef.CodedCharSetId;
DECLARE inEncoding INT propRef.Encoding;
DECLARE msgBitStream BLOB ASBITSTREAM(InputRoot.XMLNS, inEncoding, inCCSID);
INSERT INTO Database.TABLE1(MSGDATA, MSGENCODING, MSGCCSID) VALUES
(msgBitStream, inEncoding, inCCSID);
```

As an extension to the previous example, if you require the entire message to be stored along with its MQMD header, and use it later for reconstructing the entire message in another message flow on a different platform using a different code page and encoding, the database table can be extended to hold all the numeric fields of the MQMD header.

For example, a message flow running on AIX[®] inserts a message bitstream into the database table and another message flow running on Windows retrieves it and attempts to reconstruct the message along with the stored MQMD header.

The following set of numeric fields are contained in the MQMD header:

- BackoutCount (MQLONG)
- CodedCharSetId (MQLONG)
- Encoding (MQLONG)
- Expiry (MQLONG)
- Feedback (MQLONG)
- MsgFlags (MQLONG)
- MsgSeqNumber (MQLONG)
- MsgType (MQLONG)
- Offset (MQLONG)
- OriginalLength (MQLONG)
- Persistence (MQLONG)
- Priority (MQLONG)
- PutAppType (MQLONG)
- Report (MQLONG)
- Version (MQLONG)

The following example uses CodedCharSetID, Encoding, Priority, and MsgSeqNumber:

```
DECLARE propRef REFERENCE TO InputRoot.Properties;
DECLARE mqmdRef REFERENCE TO InputRoot.MQMD;
DECLARE inCCSID INT propRef.CodedCharSetId;
```

```

DECLARE inEncoding INT propRef.Encoding;

DECLARE inPriority INT mqmdRef.Priority;
DECLARE inMsgSeqNumber INT mqmdRef.MsgSeqNumber;

DECLARE msgBitStream BLOB ASBITSTREAM(InputRoot, inEncoding, inCCSID);

INSERT INTO Database.TABLE1(MSGDATA, MSGENCODING, MSGCCSID, MSGPRIORITY,MSGSEQNUMBER)
VALUES (msgBitStream, inEncoding, inCCSID, inPriority, inMsgSeqNumber);

```

If you want to insert an XML message into a database column that has a CHAR or VARCHAR data type, the ESQL must be modified to convert the input message to the CHAR data type before the INSERT statement. In the following example, a CAST is used to transform the serialized message to the CHAR data type. The CodedCharSetID and Encoding data are inserted into separate database columns.

```

DECLARE propRef REFERENCE TO InputRoot.Properties;
DECLARE inCCSID INT propRef.CodedCharSetId;
DECLARE inEncoding INT propRef.Encoding;
DECLARE msgBitStream BLOB ASBITSTREAM(InputRoot.XMLNS, inEncoding, inCCSID);
DECLARE msgChar CHAR CAST(msgBitStream AS CHAR CCSID inCCSID);
INSERT INTO Database.TABLE1(MSGDATA, MSGENCODING, MSGCCSID) VALUES (msgChar, inEncoding, inCCSID);

```

For examples of how to extract a message bitstream from a database, based on the two previous examples, see “Selecting bitstream data from a database” on page 72.

ITERATE statement

The ITERATE statement stops the current iteration of the containing WHILE, REPEAT, LOOP, or BEGIN statement identified by Label.

The containing statement evaluates its loop condition (if any), and either starts the next iteration or stops looping, as the condition dictates.

Syntax

```

▶▶—ITERATE—Label—▶▶

```

Example

In the following example, the loop iterates four times; that is the line identified by the comment Some statements 1 is passed through four times. However, the line identified by the comment Some statements 2 is passed through twice only because of the action of the IF and ITERATE statements. The ITERATE statement does **not** bypass testing the loop condition. Take particular care that the action of the ITERATE does not bypass the logic that makes the loop advance and eventually terminate. The loop count is incremented at the start of the loop in this example:

```

DECLARE i INTEGER;
SET i = 0;
X : REPEAT
  SET i = i + 1;

  -- Some statements 1

  IF i IN(2, 3) THEN
    ITERATE X;

```

```

    END IF;

    -- Some statements 2

UNTIL
    i >= 4
END REPEAT X;

```

ITERATE statements do not have to be directly contained by their labelled statement, making ITERATE statements particularly powerful.

LEAVE statement

The LEAVE statement stops the current iteration of the containing WHILE, REPEAT, LOOP, or BEGIN statement identified by Label.

The containing statement's evaluation of its loop condition (if any) is bypassed and looping stops.

Syntax

```

▶▶—LEAVE—Label—◀◀

```

Examples

In the following example, the loop iterates four times:

```

DECLARE i INTEGER;
SET i = 1;
X : REPEAT
    ...
    IF i >= 4 THEN
        LEAVE X;
    END IF;

    SET i = i + 1;
UNTIL
    FALSE
END REPEAT;

```

LEAVE statements do not have to be directly contained by their labelled statement, making LEAVE statements particularly powerful.

```

DECLARE i INTEGER;
SET i = 0;
X : REPEAT                                     -- Outer loop
    ...
    DECLARE j INTEGER;
    SET j = 0;
    REPEAT                                     -- Inner loop
        ...
        IF i >= 2 AND j = 1 THEN
            LEAVE X;                             -- Outer loop left from within inner loop
        END IF;
        ...
        SET j = j + 1;
    UNTIL
        j >= 3
    END REPEAT;

```

```

SET i = i + 1;
UNTIL
  i >= 3
END REPEAT X;

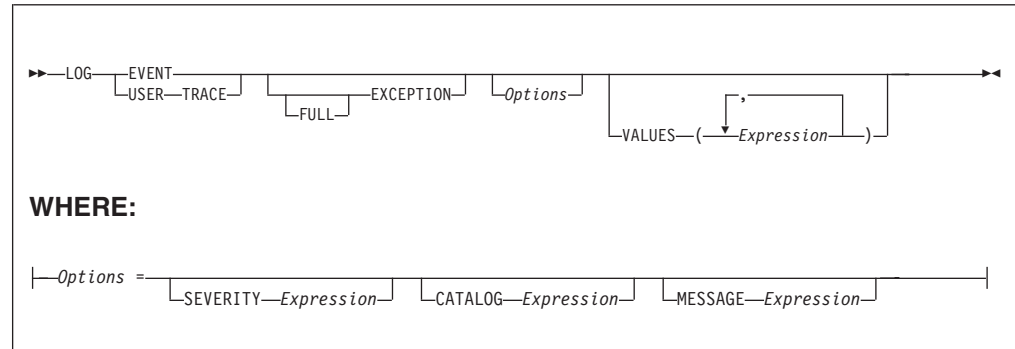
-- Execution resumes here after the leave

```

LOG statement

Use the LOG statement to write a record to the event log or to the user trace.

Syntax



CATALOG

CATALOG is an optional clause; if you omit it, CATALOG defaults to the WebSphere Message Broker current version catalog. To use the current version message catalog explicitly, use BIPv610 on all operating systems.

EVENT

A record is written to the event log, and also to the user trace, if user tracing is enabled.

EXCEPTION

The current exception, if any, is logged.

For more information on exceptions, see Errors and exception handling.

FULL

The complete nested exception report is logged, just as if the exception had reached the input node. If FULL is not specified, any wrapping exceptions are ignored, and only the original exception is logged. Therefore, you can have either a full report or simply the actual error report without the extra information regarding what was going on at the time. A current exception only exists within handler blocks (see Handling errors in message flows).

MESSAGE

The number of the message to be used. If specified, the MESSAGE clause can contain any expression that returns a non-NULL, integer, value.

If you omit MESSAGE, its value defaults to the first message number (2951) in a block of messages that is provided for use by the LOG and THROW statements in the WebSphere Message Broker catalog. If you specify a message number, you can use message numbers 2951 through 2999. Alternatively, you can generate your own catalog.

SEVERITY

The severity associated with the message. If specified, the SEVERITY clause can contain any expression that returns a non-NULL, integer, value. If you omit the clause, its value defaults to 1.

USER TRACE

A record is written to the user trace, whether user trace is enabled or not.

VALUES

Use the optional VALUES clause to provide values for the data inserts in your message. You can insert any number of pieces of information, but the messages supplied (2951 - 2999) cater for a maximum of ten data inserts.

Note the general similarity of the LOG statement to the THROW statement.

```
-- Write a message to the event log specifying the severity, catalog and message
-- number. Four inserts are provided
LOG EVENT SEVERITY 1 CATALOG 'BIPv610' MESSAGE 2951 VALUES(1,2,3,4);

-- Write to the trace log whenever a divide by zero occurs
BEGIN
  DECLARE a INT 42;
  DECLARE b INT 0;
  DECLARE r INT;

  BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE LIKE 'S22012' BEGIN
      LOG USER TRACE EXCEPTION VALUES(SQLSTATE, 'DivideByZero');

      SET r = 0x7FFFFFFFFFFFFFFF;
    END;

    SET r = a / b;
  END;

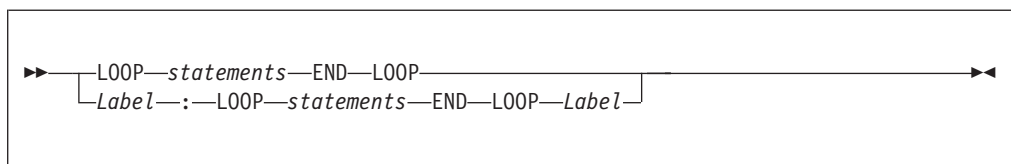
  SET OutputRoot.XMLNS.Data.Result = r;
END;
```

LOOP statement

The LOOP statement executes the sequence of statements repeatedly and unconditionally.

Ensure that the logic of the program provides some means of terminating the loop. You can use either LEAVE or RETURN statements.

Syntax



If present, *Label* gives the statement a name. This has no effect on the behavior of the LOOP statement, but allows *statements* to include ITERATE and LEAVE statements or other labelled statements, which in turn include ITERATE and LEAVE. The second *Label* can be present only if the first *Label* is present and, if it is, the labels must be identical.

Two or more labelled statements at the same level can have the same *Label* but this partly negates the advantage of the second *Label*. The advantage is that it unambiguously and accurately matches each END with its LOOP. However, a labelled statement within *statements* cannot have the same label, because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

The LOOP statement is useful in cases where the required logic dictates that a loop is always exited part way through. This is because, in these cases, the testing of a loop condition that occurs in REPEAT or WHILE statements is both unnecessary and wasteful.

Example

```

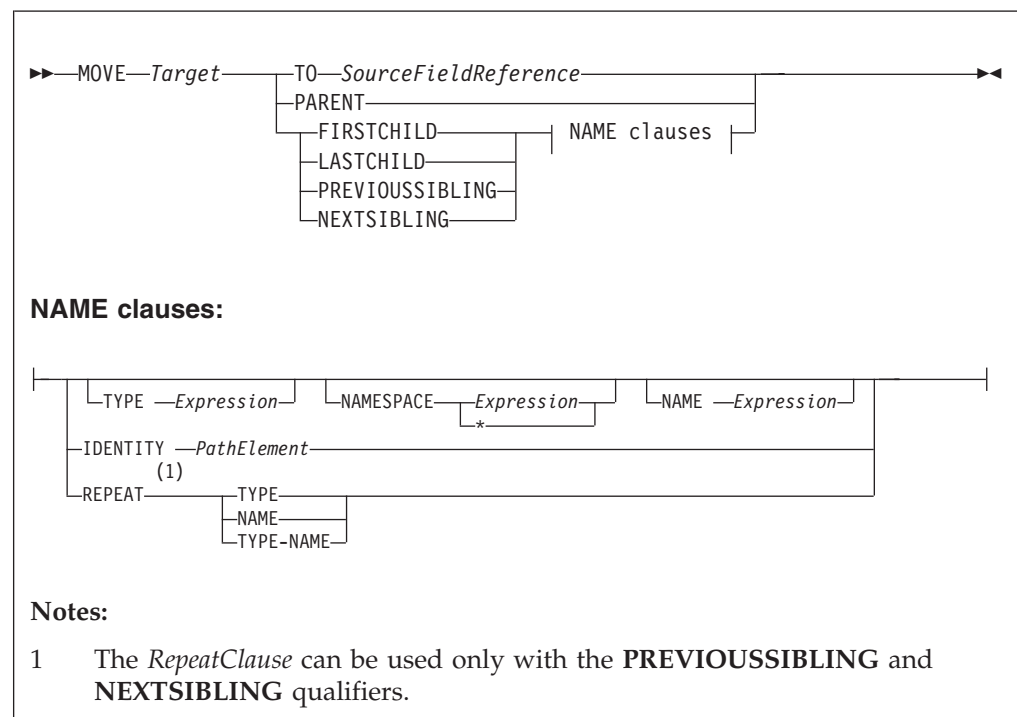
DECLARE i INTEGER;
SET i = 1;
X : LOOP
  ...
  IF i >= 4 THEN
    LEAVE X;
  END IF;
  SET i = i + 1;
END LOOP X;

```

MOVE statement

The MOVE statement changes the field to which the *Target* reference variable points.

Syntax



If you include a TO clause, this clause changes the target reference to point to the same entity as that pointed to by source; this can either be a message field or a declared variable.

If you include a PARENT, PREVIOUSIBLING, NEXTSIBLING, FIRSTCHILD, or LASTCHILD clause, the MOVE statement attempts to move the target reference variable in the direction specified relative to its current position. If any field exists in the given direction, the move succeeds. If there is no such field, the move fails; that is the reference variable continues to point to the same field or variable as

before, and the LASTMOVE function returns false. You can use the LASTMOVE function to determine the success or failure of a move.

If a TYPE clause, NAME clause, or both are present, the target is again moved in the direction specified (PREVIOUSIBLING or NEXTSIBLING, or FIRSTCHILD or LASTCHILD) but to a field with the given type, name, or both. This is particularly useful when the name or type (or both) of the target field is known, because this reduces the number of MOVE statements required to navigate to a field. This is because fields that do not match the criteria are skipped over; this can also include unexpected message tree fields, for example, those representing white space.

If the specified move cannot be made (that is, a field with the given type or name does not exist), the target remains unchanged and the LASTMOVE function returns false. The TYPE clause, NAME clause, or both clauses can contain any expression that returns a value of a suitable data type (INTEGER for type and CHARACTER for name). An exception is thrown if the value supplied is NULL.

Two further clauses, NAMESPACE and IDENTITY enhance the functionality of the NAME clause.

The NAMESPACE clause takes any expression that returns a non-null value of type character. It also takes an * indicating any namespace. Note that this cannot be confused with an expression because * is not a unary operator in ESQL.

The meaning depends on the presence of NAME and NAMESPACE clauses as follows:

NAMESPACE	NAME	Element located by...
No	No	Type, index, or both
No	Yes	Name in the default namespace
*	Yes	Name
Yes	No	Namespace
Yes	Yes	Name and namespace

The IDENTITY clause takes a single path element in place of the TYPE, NAMESPACE, and NAME clauses and follows all the rules described in the topic for field references (see “ESQL field reference overview” on page 174).

When using MOVE with PREVIOUSIBLING or NEXTSIBLING, you can specify REPEAT, TYPE, and NAME keywords that move the target to the previous or next field with the same type and name as the current field. The REPEAT keyword is particularly useful when moving to a sibling of the same kind, because you do not have to write expressions to define the type and name.

Example

```
MOVE cursor FIRSTCHILD TYPE XML.Name 'Field1';
```

This example moves the reference variable cursor to the first child field of the field to which the cursor is currently pointing, that has the type XML.Name and the name *Field1*.

See “FIELDTYPE function” on page 321 for a list of the types you can use.

The MOVE statement never creates new fields.

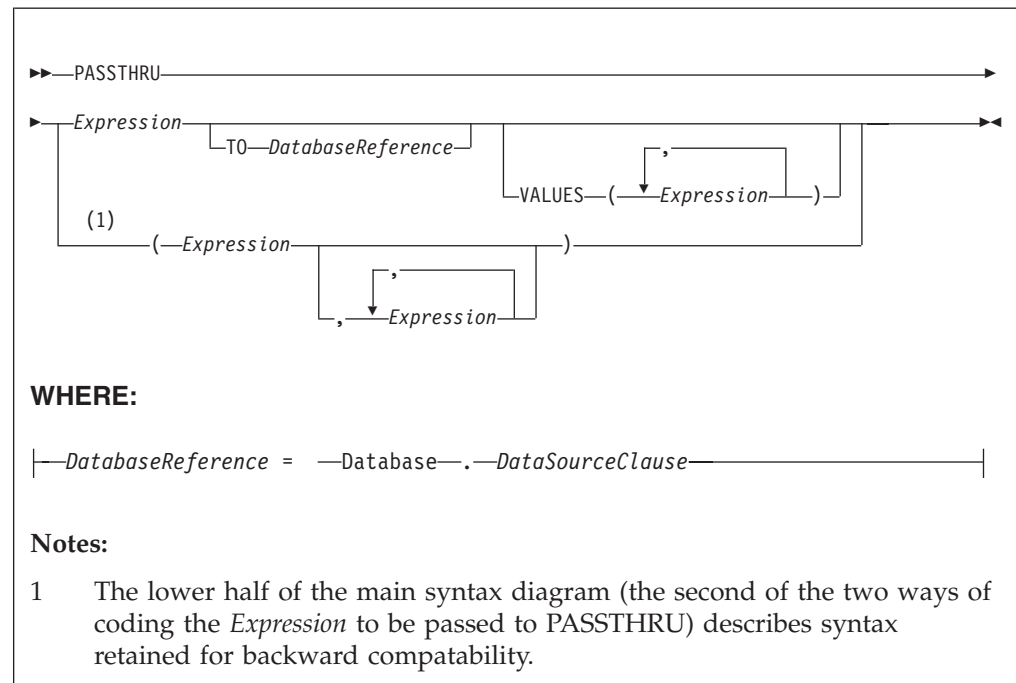
A common usage of the MOVE statement is to step from one instance of a repeating structure to the next. The fields within the structure can then be accessed by using a relative field reference. For example:

```
WHILE LASTMOVE(sourceCursor) DO
  SET targetCursor.ItemNumber = sourceCursor.item;
  SET targetCursor.Description = sourceCursor.name;
  SET targetCursor.Price      = sourceCursor.prc;
  SET targetCursor.Tax        = sourceCursor.prc * 0.175;
  SET targetCursor.quantity   = 1;
  CREATE NEXTSIBLING OF targetCursor AS targetCursor REPEAT;
  MOVE sourceCursor NEXTSIBLING REPEAT TYPE NAME;
END WHILE;
```

For more information about reference variables, and an example of moving a reference variable, see “Creating dynamic field references” on page 36.

PASSTHRU statement

The PASSTHRU statement evaluates an expression and runs the resulting character string as a database statement.



Usage

The main use of the PASSTHRU statement is to issue administrative commands to databases (for example, to create a table).

Note: Do not use PASSTHRU to call stored procedures; instead, use the CALL statement because PASSTHRU imposes limitations (you cannot use output parameters, for example).

The first expression is evaluated and the resulting character string is passed to the database pointed to by *DatabaseReference* (in the TO clause) for execution. If the TO clause is not specified, the database pointed to by the node's data source attribute is used.

Use question marks (?) in the database string to denote parameters. The parameter values are supplied by the VALUES clause.

If the VALUES clause is specified, its expressions are evaluated and passed to the database as parameters; (that is, the expressions' values are substituted for the question marks in the database statement).

If only one VALUE expression exists, the result might or might not be a list. If it is a list, the list's scalar values are substituted sequentially for the question marks. If it is not a list, the single scalar value is substituted for the (single) question mark in the database statement. If more than one VALUE expression exists, none of the expressions evaluate to a list; their scalar values are substituted sequentially for the question marks instead.

Because the database statement is constructed by the user program, it is not essential to use parameter markers (that is, the question marks) or the VALUES clause, because the whole of the database statement could be supplied, as a literal string, by the program. However, use parameter markers whenever possible because this reduces the number of different statements that need to be prepared and stored in the database and the broker.

Database reference

A database reference is a special instance of the field references that is used to refer to message trees. It consists of the word **Database** followed by the name of a data source (that is, the name of a database instance).

You can specify the data source name directly or by an expression enclosed in braces ({...}). A directly-specified data source name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see "DECLARE statement" on page 235).

If you have created a message flow that contains one of the following nodes, and the ESQL that is associated with this node includes a PASSTHRU statement and a database reference, you must specify a value for the Data source property of the relevant node:

- Compute
- Database
- Filter

Handling errors

It is possible for errors to occur during PASSTHRU operations. For example, the database might not be operational or the statement might be invalid. In these cases, an exception is thrown (unless the node has its Throw exception on database error property cleared). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the DECLARE HANDLER statement).

For further information about handling database errors, see “Capturing database state” on page 84.

Examples

The following example creates the table Customers in schema Shop in database DSN1:

```
PASSTHRU 'CREATE TABLE Shop.Customers (
  CustomerNumber INTEGER,
  FirstName      VARCHAR(256),
  LastName       VARCHAR(256),
  Street         VARCHAR(256),
  City           VARCHAR(256),
  Country        VARCHAR(256)
) ' TO Database.DSN1;
```

If, as in the last example, the ESQL statement is specified as a string literal, you must put single quotes around it. If, however, it is specified as a variable, omit the quotation marks. For example:

```
SET myVar = 'SELECT * FROM user1.stocktable';
SET OutputRoot.XMLNS.Data[] = PASSTHRU(myVar);
```

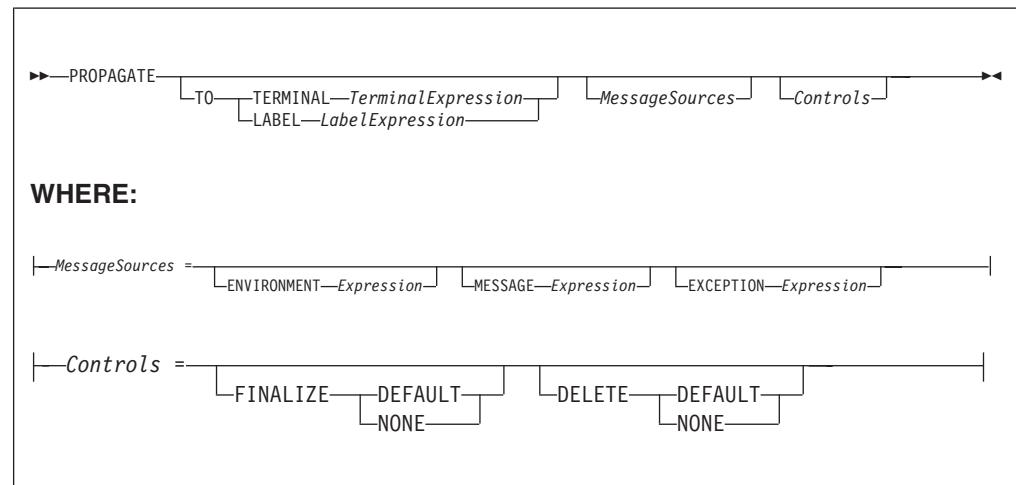
The following example “drops” (that is, deletes) the table Customers from schema Shop in database DSN1:

```
PASSTHRU 'DROP TABLE Shop.Customers' TO Database.DSN1;
```

PROPAGATE statement

The PROPAGATE statement propagates a message to the downstream nodes.

Syntax



You can use the PROPAGATE statement in Compute and Database nodes, but not in Filter nodes. The additions to this statement assist in error handling - see “Coding ESQL to handle errors” on page 79.

TO TERMINAL clause

If the TO TERMINAL clause is present, *TerminalExpression* is evaluated. If the result is of type CHARACTER, a message is propagated to a terminal according to the rule:

```
'nowhere' : no propagation
'failure' : Failure
'out'     : Out
'out1'    : Out1
'out2'    : Out2
'out3'    : Out3
'out4'    : Out4
```

Tip: Terminal names are case sensitive so, for example, Out1 does not match any terminal.

If the result of *TerminalExpression* is of type INTEGER, a message is propagated to a terminal according to the rule:

```
-2 : no propagation
-1 : Failure
0  : Out
1  : Out1
2  : Out2
3  : Out3
4  : Out4
```

If the result of *TerminalExpression* is neither a CHARACTER nor an INTEGER, the broker throws an exception.

If there is neither a TO TERMINAL nor a TO LABEL clause, the broker propagates a message to the Out terminal.

Tip: Using character values in terminal expressions leads to the most natural and readable code. Integer values, however, are easier to manipulate in loops and marginally faster.

TO LABEL clause

If the TO LABEL clause is present, *LabelExpression* is evaluated. If the result is of type CHARACTER **and** there is a Label node with a Label property that matches *LabelExpression*, in the same flow, the broker propagates a message to that node.

Tip: Labels, like terminals, are case sensitive. Also, note that, as with route to Label nodes, it is the Label Name property of the Label node that defines the target, not the node's label itself.

If the result of *LabelExpression* is NULL or not of type CHARACTER, or there is no matching Label node in the flow, the broker throws an exception.

If there is neither a TO TERMINAL nor a TO LABEL clause, the broker propagates a message to the out terminal.

MessageSources clauses

The MessageSources clauses select the message trees to be propagated. This clause is only applicable to the Compute node (it has no effect in the Database node).

The values that you can specify in MessageSources clauses are:

```
ENVIRONMENT :
  InputLocalEnvironment
  OutputLocalEnvironment
```

```
Message :
  InputRoot
  OutputRoot
```

ExceptionList :
 InputExceptionList
 OutputExceptionList

If there is no MessageSources clause, the node's Compute mode property is used to determine which messages are propagated.

FINALIZE clause

Finalization is a process that fixes header chains and makes the Properties folder match the headers. If present, the FINALIZE clause allows finalization to be controlled.

This clause is only applicable to the Compute node (it has no effect in a Database node).

The Compute node allows its output message to be changed by other nodes (by the other nodes changing their input message). However, a message created by a Compute node cannot be changed by another node after:

- It has been finalized
- It has reached any output or other node which generates a bit-stream

If FINALIZE is set to DEFAULT, or the FINALIZE clause is absent, the output message (but not the Environment, Local Environment or Exception List) is finalized before propagation.

If FINALIZE is set to NONE, no finalization takes place. This option is required if you want to preserve and allow updates of the entire output message tree by the nodes downstream in the message flow and is used with DELETE NONE as described in the next section.

DELETE clause

The DELETE clause allows the clearing of the output local environment, message, and exception list to be controlled.

The DELETE clause is only applicable to the Compute node (it has no effect in a Database node).

If DELETE is set to DEFAULT, or the DELETE clause is absent, the output local environment, message, and exception list are all cleared and their memory recovered immediately after propagation.

If DELETE is set to NONE, nothing is cleared. Use DELETE NONE if you want the downstream nodes to be able to see a single instance of output local environment message, and exception list trees. Each propagate starts with the content of these trees as created by the previous propagate rather than starting with empty trees. If you also want these nodes to update the output tree, DELETE NONE must be used with the FINALIZE NONE option described in the previous section.

Note that the output trees that are finalized are cleared, regardless of which ones are propagated.

Propagation is a synchronous process. That is, the next statement is not executed until all the processing of the message in downstream nodes has completed. Be aware that this processing might throw exceptions and, if these exceptions are not caught, they will prevent the statement following the PROPAGATE call being reached. This behavior might be what the logic of your flow requires but, if it is not, you can use a handler to catch the exception and perform the necessary actions. Note that exceptions thrown downstream of a propagate, if not caught, will also prevent the final automatic actions of a Compute or Database node (for example, issuing a COMMIT Transaction set to Commit) from taking place.

```

DECLARE i INTEGER 1;
DECLARE count INTEGER;
SET count = CARDINALITY(InputRoot.XMLNS.Invoice.Purchases."Item"[]);

WHILE i <= count DO
  --use the default tooling-generated procedure for copying message headers
  CALL CopyMessageHeaders();
  SET OutputRoot.XMLNS.BookSold.Item = InputRoot.XMLNS.Invoice.Purchases.Item[i];
  PROPAGATE;
  SET i = i+1;
END WHILE;
RETURN FALSE;

```

The following messages are produced on the Out terminal by the PROPAGATE statement:

```

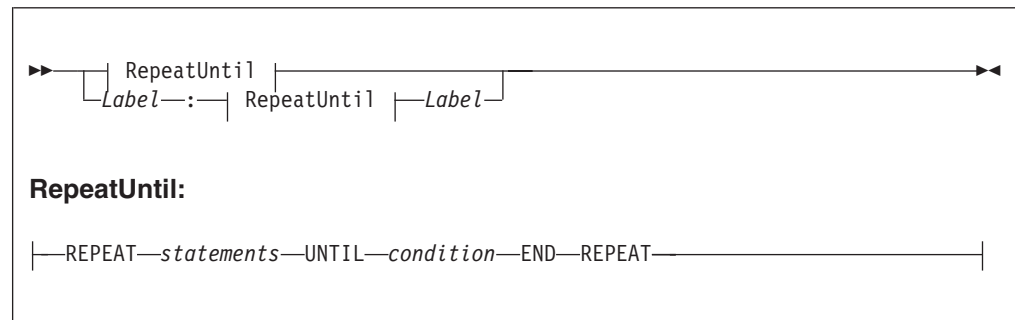
<BookSold>
  <Item>
    <Title Category="Computer" Form="Paperback" Edition="2">The XML Companion </Title>
    <ISBN>0201674866</ISBN>
    <Author>Neil Bradley</Author>
    <Publisher>Addison-Wesley</Publisher>
    <PublishDate>October 1999</PublishDate>
    <UnitPrice>27.95</UnitPrice>
    <Quantity>2</Quantity>
  </Item>
</BookSold>
<BookSold>
  <Item>
    <Title Category="Computer" Form="Paperback" Edition="2">A Complete Guide to
    DB2 Universal Database</Title>
    <ISBN>1558604820</ISBN>
    <Author>Don Chamberlin</Author>
    <Publisher>Morgan Kaufmann Publishers</Publisher>
    <PublishDate>April 1998</PublishDate>
    <UnitPrice>42.95</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
</BookSold>
<BookSold>
  <Item>
    <Title Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers
    Handbook</Title>
    <ISBN>0782121799</ISBN>
    <Author>Phillip Heller, Simon Roberts </Author>
    <Publisher>Sybex, Inc.</Publisher>
    <PublishDate>September 1998</PublishDate> <UnitPrice>59.99</UnitPrice>
    <Quantity>1</Quantity>
  </Item>
</BookSold>

```

REPEAT statement

The REPEAT statement processes a sequence of statements, then evaluates the condition expression.

Syntax



The `REPEAT` statement repeats the steps until condition is `TRUE`. Ensure that the logic of the program is such that the loop terminates. If the condition evaluates to `UNKNOWN`, the loop does **not** terminate.

If present, the `Label` gives the statement a name. This has no effect on the behavior of the `REPEAT` statement, but allows statements to include `ITERATE` and `LEAVE` statements or other labelled statements, which in turn include `ITERATE` and `LEAVE`. The second `Label` can be present only if the first `Label` is present and, if it is, the labels must be identical. Two or more labelled statements at the same level can have the same label, but this partly negates the advantage of the second `Label`. The advantage is that it unambiguously and accurately matches each `END` with its `REPEAT`. However, a labelled statement within statements cannot have the same label because this makes the behavior of the `ITERATE` and `LEAVE` statements ambiguous.

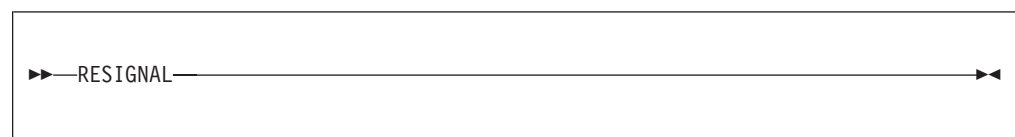
Example

```
DECLARE i INTEGER;  
SET i = 1;  
X : REPEAT  
  ...  
  SET i = i + 1;  
UNTIL  
  i >= 3  
END REPEAT X;
```

RESIGNAL statement

The `RESIGNAL` statement rethrows the current exception, if one exists.

Syntax



You can use `RESIGNAL` only in error handlers.

Typically, `RESIGNAL` is used when an error handler catches an exception that it cannot handle. The handler uses `RESIGNAL` to rethrow the original exception so that a handler in higher-level scope has the opportunity to handle it.

Because the handler throws the original exception, rather than a new (and therefore different) one:

1. The higher-level handler is not affected by the presence of the lower-level handler.
2. If no higher-level handler is present, you get a full error report in the event log.

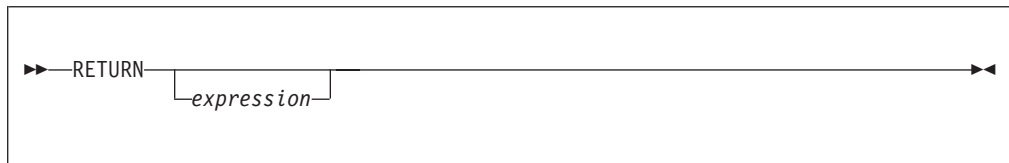
Example

```
RESIGNAL;
```

RETURN statement

The RETURN statement ends processing. What happens next depends on the programming context in which the RETURN statement is issued.

Syntax



Main Function

When used in the Main function, the RETURN statement stops processing of the module and returns control to the next node in a message flow. In the Main function the return statement must contain an *expression* of BOOLEAN type. The behavior of the RETURN statement in the Main function is dependant on the node. In the Compute node for example, if *expression* is anything other than TRUE, propagation of the message is stopped. In the Filter node, however, the message is propagated to the terminal matching the value of *expression*: TRUE, FALSE and UNKNOWN. The following table describes the differences between the RETURN statement when used in the Main function of the Compute, Filter, and Database nodes.

Node	RETURN TRUE;	RETURN FALSE;	RETURN UNKNOWN (if BOOLEAN type) or RETURN NULL;	RETURN;
Compute	Propagate message to Out terminal.	Stop propagation	Stop propagation	Deploy failure (BIP2912E: Type mismatch on RETURN)
Database	Propagate message to Out terminal.	Stop propagation	Stop propagation	Deploy failure (BIP2912E: Type mismatch on RETURN)
Filter	Propagate message to True terminal	Propagate message to False terminal	Propagate message to Unknown terminal	Deploy failure (BIP2912E: Type mismatch on RETURN)

User defined functions and procedures

When used in a function or a procedure, the RETURN statement stops processing of that function and returns control to the calling expression. The *expression*, which must be present if the function or procedure has been declared with a RETURNS clause, is evaluated and acts as the return value of the function. The data type of the returned value must be the same as that in the function's declaration. The following table describes the differences between the RETURN statement when used in user defined functions and procedures.

	RETURN <i>expression</i>;	RETURN NULL; (or return <i>expression</i> that evaluates to NULL)	RETURN;	No RETURN statement
User defined function or procedure with a RETURNS clause	Returns control to the calling expression with the value of <i>expression</i>	Returns control to the calling expression with NULL	Deploy failure (BIP2912E: Type mismatch on RETURN)	Returns control to the calling expression with NULL after all the statements in the function or procedure have been run
User defined function or procedure without a RETURNS clause	Deploy failure (BIP2401E: Syntax error: expected ; but found <i>expression</i>)	Deploy failure (BIP2401E: Syntax error: expected ; but found NULL)	Returns control to the calling expression	Returns control to the calling expression after all the statements in the function or procedure have been run

The RETURN statement must be used within the body of a function or procedure that has the RETURNS statement in its declaration. This function can be invoked using the CALL ... INTO statement. The RETURNS statement provides the datatype that the function or procedure returns to the "CALL statement" on page 196. The CALL ... INTO statement specifies the variable to which the return value is assigned. The example in this topic shows an example of how a RETURNS and CALL ... INTO statement are used together to assign the return statement. If you use the CALL ... INTO statement to call a function or procedure that does not have a RETURNS statement declared, a BIP2912E error message is generated.

Example

The following example, which is based on "Example message" on page 385, illustrates how the RETURN, RETURNS and CALL...INTO statements can be used:

```
CREATE FILTER MODULE ProcessOrder
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN

    DECLARE SpecialOrder BOOLEAN;
    SET OutputRoot.MQMD = InputRoot.MQMD;
    CALL IsBulkOrder(InputRoot.XMLNS.Invoice.Purchases) INTO SpecialOrder;

    --
    -- more processing could be inserted here
    -- before routing the order to the appropriate terminal
    --
    RETURN SpecialOrder;
```

```

END;

CREATE FUNCTION IsBulkOrder (P1 REFERENCE)
  RETURNS BOOLEAN
  BEGIN
    -- Declare and initialize variables--
    DECLARE a INT 1;
    DECLARE PriceTotal FLOAT 0.0;
    DECLARE NumItems INT 0;
    DECLARE iroot REFERENCE TO P1;

    -- Calculate value of order, however if this is a bulk purchase, the
    -- order will need to be handled differently (discount given) so return TRUE
    -- or FALSE depending on the size of the order
    WHILE a <= CARDINALITY(iroot.Item[]) DO
      SET NumItems = NumItems + iroot.Item[a].Quantity;
      SET PriceTotal = PriceTotal + iroot.Item[a].UnitPrice;
      SET a = a + 1;
    END WHILE;
    RETURN (PriceTotal/NumItems > 42);

  END;

END MODULE;

```

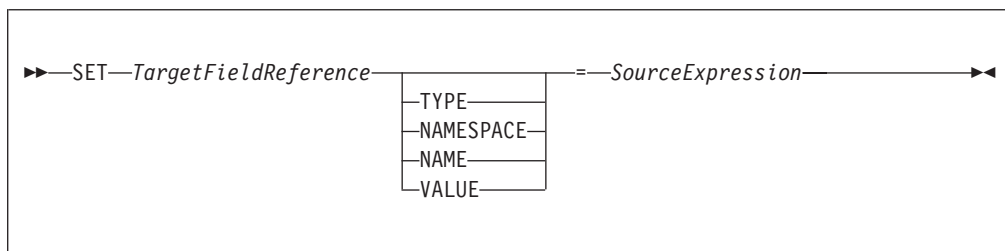
In the example, if the average price of items is greater than 42, TRUE is returned; otherwise FALSE is returned. Thus, a Filter node could route messages describing expensive items down a different path from messages describing inexpensive items. From the example, the CALL `IsBulkOrder(InputRoot.XMLNS.Invoice.Purchases)` INTO `SpecialOrder`; statement can also be written as `SpecialOrder = IsBulkOrder(InputRoot.XMLNS.Invoice.Purchases);`

If you are using the PROPAGATE statement in your node it is important that you use a RETURN FALSE; to prevent automatic propagation of the message to the next node in the message flow. See "PROPAGATE statement" on page 261 for an example of preventing the implicit propagate at the end of processing in a Compute node.

SET statement

The SET statement assigns a value to a variable.

Syntax



Introduction

TargetFieldReference identifies the target of the assignment. The target can be any of the following:

- A declared scalar variable
- A declared row variable

- One of the predefined row variables (for example, *InputRoot*)
- A field within any kind of row variable (that is, a sub tree or conceptual row)
- A list of fields within any kind of row variable (that is, a conceptual list)
- A declared reference variable that points to any of the above

The target cannot be any kind of database entity.

SourceExpression is an expression that supplies the value to be assigned. It can be any kind of expression and can return a scalar, row or list value.

Assignment to scalar variables

If the target is a declared scalar variable, *SourceExpression* is evaluated and assigned to the variable. If need be, its value is converted to the data type of the variable. If this conversion is not possible, there will be either an error at deploy time or an exception at run time.

Null values are handled in exactly the same way as any other value. That is, if the expression evaluates to null, the value “null” is assigned to the variable.

For scalar variables the TYPE, NAME, NAMESPACE, and VALUE clauses are meaningless and are not allowed.

Assignment to rows, lists, and fields

If the target is a declared row variable, one of the predefined row variables, a field within any kind of row variable, a list of fields within any kind of row variable, or a declared reference variable that points to any of these things, the ultimate target is a field. In these cases, the target field is navigated to (creating the fields if necessary).

If array indices are used in *TargetFieldReference*, the navigation to the target field can only create fields on the direct path from the root to the target field. For example, the following SET statement requires that at least one instance of Structure already exists in the message:

```
SET OutputRoot.XMLNS.Message.Structure[2].Field = ...
```

The target field's value is set according to a set of rules, based on:

1. The presence or absence of the TYPE, NAME, NAMESPACE, or VALUE clauses
 2. The data type returned by the source expression
1. If no TYPE, NAME, NAMESPACE, or VALUE clause is present (which is the most common case) the outcome depends on whether *SourceExpression* evaluates to a scalar, a row, or a list:
 - If *SourceExpression* evaluates to a scalar, the value of the target field is set to the value returned by *SourceExpression*, except that, if the result is null, the target field is discarded. Note that the new value of the field might not be of the same data type as its previous value.
 - If *SourceExpression* evaluates to a row:
 - a. The target field is identified.
 - b. The target field's value is set.
 - c. The target field's child fields are replaced by a new set, dictated by the structure and content of the list.
 - If *SourceExpression* evaluates to a list:
 - a. The set of target fields in the target tree are identified.

- b. If there are too few target fields, more are created; if there are too many, the extra ones are removed.
- c. The target fields' values are set.
- d. The target fields' child fields are replaced by a new set, dictated by the structure and content of the list.

For further information on working with elements of type `list` see “Working with elements of type `list`” on page 45

2. If a `TYPE` clause is present, the type of the target field is set to the value returned by *SourceExpression*. An exception is thrown if the returned value is not scalar, is not of type `INTEGER`, or is `NULL`.
3. If a `NAMESPACE` clause is present, the namespace of the target field is set to the value returned by *SourceExpression*. An exception is thrown if the returned value is not scalar, is not of type `CHARACTER`, or is `NULL`.
4. If a `NAME` clause is present, the name of the target field is set to the value returned by *SourceExpression*. An exception is thrown if the returned value is not scalar, is not of type `CHARACTER`, or is `NULL`.
5. If a `VALUE` clause is present, the value of the target field is changed to that returned by *SourceExpression*. An exception is thrown if the returned value is not scalar.

Notes

`SET` statements are particularly useful in Compute nodes that modify a message, either changing a field or adding a new field to the original message. `SET` statements are also useful in Filter and Database nodes, to set declared variables or the fields in the Environment tree or Local Environment trees. You can use statements such as the following in a Compute node that modifies a message:

```
SET OutputRoot = InputRoot;
SET OutputRoot.XMLNS.Order.Name = UPPER(InputRoot.XMLNS.Order.Name);
```

This example puts one field in the message into uppercase. The first statement constructs an output message that is a complete copy of the input message. The second statement sets the value of the `Order.Name` field to a new value, as defined by the expression on the right.

If the `Order.Name` field does not exist in the original input message, it does not exist in the output message generated by the first statement. The expression on the right of the second statement returns `NULL` (because the field referenced inside the `UPPER` function call does not exist). Assigning the `NULL` value to a field has the effect of deleting it if it already exists, and so the effect is that the second statement has no effect.

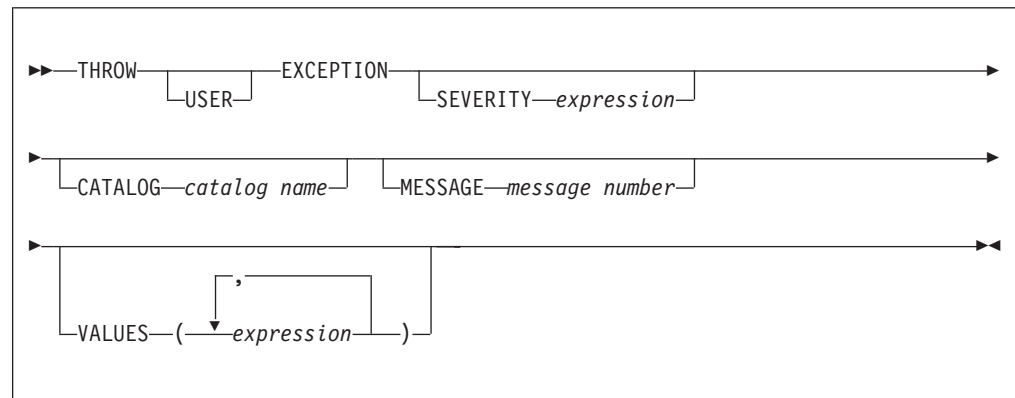
If you want to assign a `NULL` value to a field without deleting the field, use a statement like this:

```
SET OutputRoot.XMLNS.Order.Name VALUE = NULL;
```

THROW statement

Use the `THROW` statement to generate a user exception.

Syntax



The **USER** keyword indicates the type of exception being thrown. (Currently, only **USER** exceptions are supported, and if you omit the **USER** keyword the exception defaults to a **USER** exception anyway.) Specify the **USER** keyword, even though it currently has no effect, for the following reasons:

- If future broker releases support other types of exception, and the default type changes, your code will not need to be changed.
- It makes it clear that this is a user exception.

SEVERITY is an optional clause that determines the severity associated with the exception. The clause can contain any expression that returns a non-NULL, integer value. If you omit the clause, it defaults to 1.

On Windows you must set **SEVERITY** to 3, so that the Windows event log reports the error correctly.

CATALOG is an optional clause; if you omit it, **CATALOG** defaults to the WebSphere Message Broker current version catalog. To use the current version message catalog explicitly, use **BIPv610** on all operating systems.

MESSAGE is an optional clause; if you omit it, it defaults to the first message number of the block of messages provided for using **THROW** statements in the default catalog (2951). If you enter a message number in the **THROW** statement, you can use message numbers 2951 to 2999 from the default catalog. Alternatively, you can generate your own catalog by following the instructions in [Creating message catalogs](#).

Use the optional **VALUES** field to insert data into your message. You can insert any number of pieces of information, but the messages supplied (2951 - 2999) cater for eight inserts only.

Examples

Here are some examples of how you might use a **THROW** statement:

- ```
THROW USER EXCEPTION;
```
- ```
THROW USER EXCEPTION CATALOG 'BIPv610' MESSAGE  
2951 VALUES(1,2,3,4,5,6,7,8) ;
```
-

```

THROW USER EXCEPTION CATALOG 'BIPv610' MESSAGE
2951 VALUES('The SQL State: ',
            SQLSTATE, 'The SQL Code: ', SQLCODE, 'The SQLNATIVEERROR: ', SQLNATIVEERROR,
            'The SQL Error Text: ', SQLERRORTEXT ) ;

```

•

```

THROW USER EXCEPTION CATALOG 'BIPv610' MESSAGE
2951 ;

```

•

```

THROW USER EXCEPTION CATALOG 'MyCatalog' MESSAGE
2951 VALUES('Hello World') ;

```

•

```

THROW USER EXCEPTION MESSAGE
2951 VALUES('Insert text 1', 'Insert text 2') ;

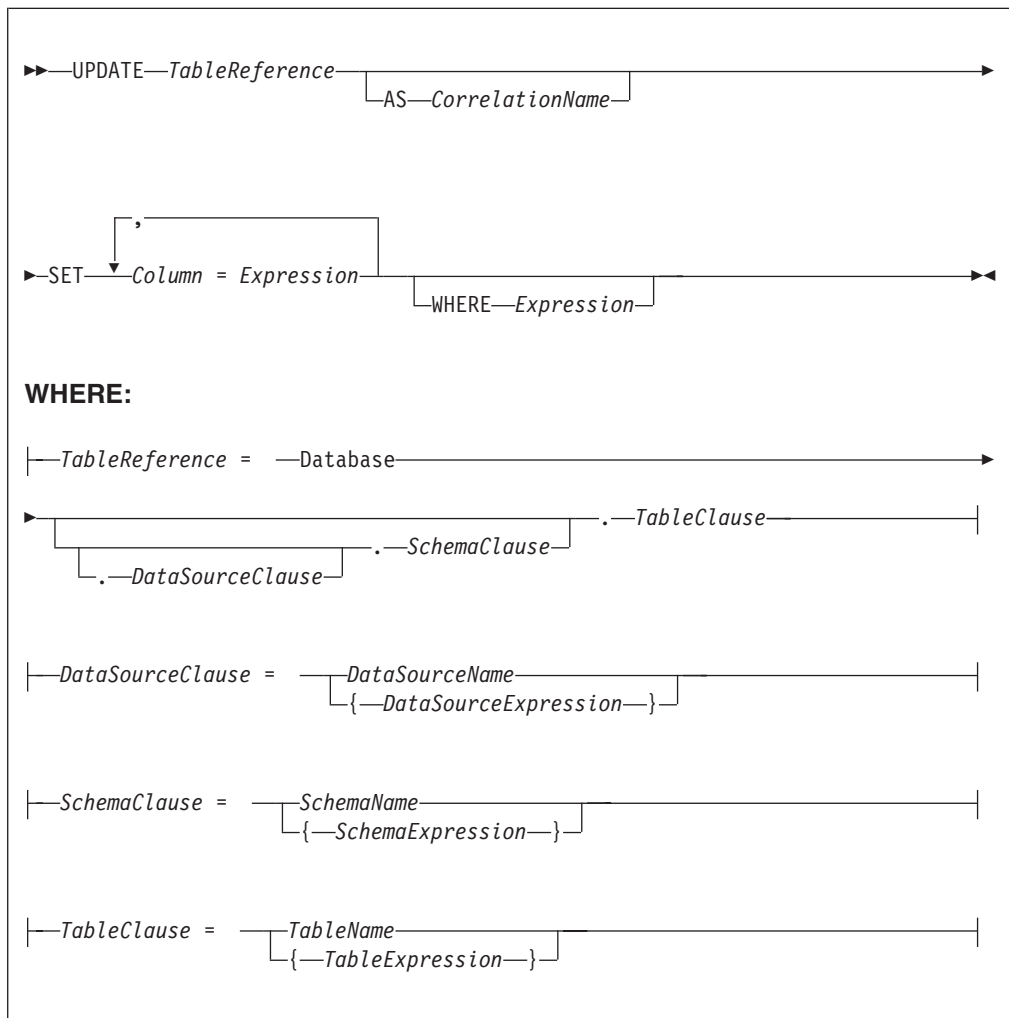
```

For more information about how to throw an exception, and details of SQLSTATE, SQLCODE, SQLNATIVEERROR, and SQLERRORTXT, see “ESQL database state functions” on page 279.

UPDATE statement

The UPDATE statement changes the values of specified columns, in selected rows, in a table in an external database.

Syntax



All rows for which the WHERE clause expression evaluates to TRUE are updated in the table identified by *TableReference*. Each row is examined in turn and a variable is set to point to the current row. Typically, the WHERE clause expression uses this variable to access column values and thus cause rows to be updated, or retained unchanged, according to their contents. The variable is referred to by *CorrelationName* or, in the absence of an AS clause, by *TableName*. When a row has been selected for updating, each column named in the SET clause is given a new value as determined by the corresponding expression. These expressions can, if you wish, refer to the current row variable.

Table reference

A table reference is a special case of the field references that are used to refer to message trees. It always starts with the word “Database” and can contain any of the following elements:

- A table name only
- A schema name and a table name
- A data source name (that is, the name of a database instance), a schema name, and a table name

In each case, the name can be specified directly or by an expression enclosed in braces ({...}). A directly-specified data source, schema, or table name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see “DECLARE statement” on page 235).

If a schema name is not specified, the default schema for the broker's database user is used.

If a data source name is not specified, the database pointed to by the node's data source attribute is used.

The WHERE clause

The WHERE clause expression can use any of the broker's operators and functions in any combination. It can refer to table columns, message fields, and any declared variables or constants.

However, be aware that the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be evaluated by the database. If it can, it is given to the database. In order to be evaluated by the database, it must use only those functions and operators supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECT functions, and to any other declared variables or constants within scope.

If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each sub-expression separately. It then attempts to give the database those sub-expressions that it can evaluate, leaving the broker to evaluate the rest. You need to be aware of this situation for two reasons:

1. Apparently trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.

2. Some databases' functions exhibit subtle differences of behavior from those of the broker.

Handling errors

It is possible for errors to occur during update operations. For example, the database might not be operational, or the table might have constraints defined that the new values would violate. In these cases, an exception is thrown (unless the node has its `throw exception on database error` property set to `FALSE`). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the `DECLARE HANDLER` statement).

For further information about handling database errors, see “Capturing database state” on page 84.

Examples

The following example assumes that the `dataSource` property of the Database node has been configured, and that the database it identifies has a table called `STOCKPRICES`, with columns called `COMPANY` and `PRICES`. It updates the `PRICE` column of the rows in the `STOCKPRICES` table whose `COMPANY` column matches the value given in the `Company` field in the message.

```
UPDATE Database.StockPrices AS SP
  SET PRICE = InputBody.Message.StockPrice
  WHERE SP.COMPANY = InputBody.Message.Company
```

In the following example (which make similar assumptions), the `SET` clause expression refers to the existing value of a column and thus decrements the value by an amount in the message:

```
UPDATE Database.INVENTORY AS INV
  SET QUANTITY = INV.QUANTITY - InputBody.Message.QuantitySold
  WHERE INV.ITEMNUMBER = InputBody.Message.ItemNumber
```

The following example updates multiple columns:

```
UPDATE Database.table AS T
  SET column1 = T.column1+1,
      column2 = T.column2+2;
```

Note that the column names (on the left of the `=`) are single identifiers. They must not be qualified with a table name or correlation name. In contrast, the references to database columns in the expressions (to the right of the `=`) must be qualified with the correlation name.

The next example shows the use of calculated data source, schema, and table names:

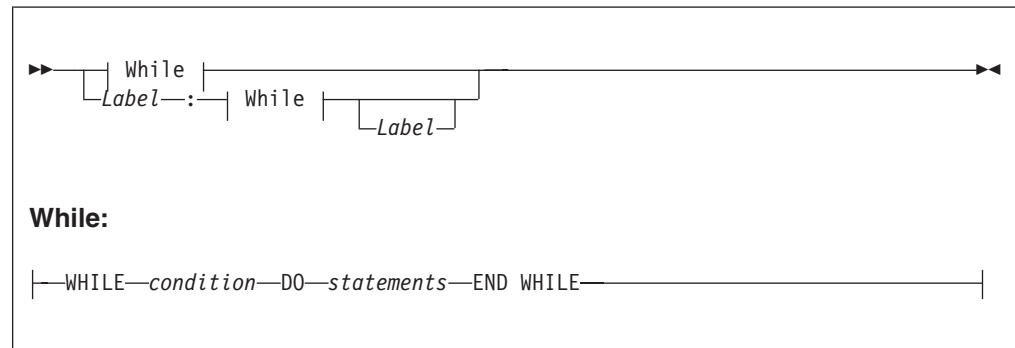
```
-- Declare variables to hold the data source, schema and table names
-- and set their default values
DECLARE Source CHARACTER 'Production';
DECLARE Schema CHARACTER 'db2admin';
DECLARE Table CHARACTER 'DynamicTable1';
-- Code which calculates their actual values comes here

-- Update rows in the table
UPDATE Database.{Source}.{Schema}.{Table} AS R SET Value = 0;
```

WHILE statement

The WHILE statement evaluates a condition expression, and if it is TRUE executes a sequence of statements.

Syntax



The WHILE statement repeats the steps specified in DO as long as *condition* is TRUE. It is your responsibility to ensure that the logic of the program is such that the loop terminates. If *condition* evaluates to UNKNOWN, the loop terminates immediately.

If present, *Label* gives the statement a name. This has no effect on the behavior of the WHILE statement itself, but allows statements to include ITERATE and LEAVE statements or other labelled statements, which in turn include them. The second *Label* can be present only if the first *Label* is present and if it is, the labels must be identical. It is not an error for two or more labelled statements at the same level to have the same *Label*, but this partly negates the advantage of the second *Label*. The advantage is that it unambiguously and accurately matches each END with its WHILE. However, it is an error for a labelled statement within statements to have the same label, because this makes the behavior of the ITERATE and LEAVE statements ambiguous.

Example

For example:

```

DECLARE i INTEGER;
SET i = 1;
X : WHILE i <= 3 DO
  ...
  SET i = i + 1;
END WHILE X;
  
```

ESQL functions: reference material, organized by function type

A table summarizing the functions that are available in ESQL, and what they do.

CATEGORY	FUNCTIONS	RELATED KEYWORDS
Variable manipulation		
Manipulation of all sources of variables		
Basic manipulation of all types of variable	<ul style="list-style-type: none"> • "CAST function" on page 331 	<ul style="list-style-type: none"> • ENCODING, CCSID, AS

Selective assignment to any variable	<ul style="list-style-type: none"> • “CASE function” on page 329 • “COALESCE function” on page 371 	<ul style="list-style-type: none"> • ELSE, WHEN, THEN, END
Creation of values	<ul style="list-style-type: none"> • “UIDASBLOB function” on page 376 • “UIDASCHAR function” on page 376 	
Manipulation of message trees		
Assignment to and deletion from a message tree	<ul style="list-style-type: none"> • “SELECT function” on page 345 (used with SET statement) • “ROW constructor function” on page 352 • “LIST constructor function” on page 353 	<ul style="list-style-type: none"> • FROM, AS, ITEM, THE, SUM, COUNT, MAX, MIN
Information relating to message trees or subtrees	<ul style="list-style-type: none"> • “ASBITSTREAM function” on page 316 • “BITSTREAM function (deprecated)” on page 319 • “FIELDNAME function” on page 320 • “FIELDNAMESPACE function” on page 321 • “FIELDTYPE function” on page 321 	
Processing Lists	<ul style="list-style-type: none"> • CARDINALITY, see “CARDINALITY function” on page 327 for details. • EXISTS, see “EXISTS function” on page 327 for details. • SINGULAR, see “SINGULAR function” on page 328 for details. • THE, see “THE function” on page 329 for details. 	
Processing repeating fields	<ul style="list-style-type: none"> • FOR • “SELECT function” on page 345 	<ul style="list-style-type: none"> • ALL, ANY, SOME • FROM, AS, ITEM, THE, SUM, COUNT, MAX, MIN
Processing based on data type		
String processing		
Numeric information about strings	<ul style="list-style-type: none"> • “LENGTH function” on page 305 • “POSITION function” on page 308 	IN
String conversion	<ul style="list-style-type: none"> • “UPPER and UCASE functions” on page 315 • “LOWER and LCASE functions” on page 306 	

String manipulation	<ul style="list-style-type: none"> • “CONTAINS function” on page 304 • “ENDSWITH function” on page 305 • “LEFT function” on page 305 • “LTRIM function” on page 306 • “OVERLAY function” on page 307 • “REPLACE function” on page 309 • “REPLICATE function” on page 309 • “RIGHT function” on page 310 • “RTRIM function” on page 310 • “SPACE function” on page 311 • “STARTSWITH function” on page 311 • “SUBSTRING function” on page 312 • “TRANSLATE function” on page 313 • “TRIM function” on page 314 	<ul style="list-style-type: none"> • LEADING, TRAILING, BOTH, FROM • PLACING, FROM, FOR • FROM FOR
Numeric processing		
Bitwise operations	<ul style="list-style-type: none"> • “BITAND function” on page 291 • “BITNOT function” on page 291 • “BITOR function” on page 292 • “BITXOR function” on page 292 	
General	<ul style="list-style-type: none"> • “ABS and ABSVAL functions” on page 289 • “ACOS function” on page 290 • “ASIN function” on page 290 • “ATAN function” on page 290 • “ATAN2 function” on page 290 • “COS function” on page 293 • “COSH function” on page 294 • “COT function” on page 294 • “DEGREES function” on page 294 • “EXP function” on page 294 • “FLOOR function” on page 295 • “LN and LOG functions” on page 295 • “LOG10 function” on page 296 • “MOD function” on page 296 • “POWER function” on page 296 • “RADIANS function” on page 297 • “RAND function” on page 297 • “ROUND function” on page 297 • “SIGN function” on page 301 • “SIN function” on page 301 • “SINH function” on page 301 • “SQRT function” on page 301 • “TAN function” on page 302 • “TANH function” on page 302 • “TRUNCATE function” on page 303 	
Date time processing		

	<ul style="list-style-type: none"> • “CURRENT_DATE function” on page 286 • “CURRENT_GMTDATE function” on page 287 • “CURRENT_GMTTIME function” on page 287 • “CURRENT_TIME function” on page 286 • “CURRENT_TIMESTAMP function” on page 286 • “CURRENT_GMTTIMESTAMP function” on page 287 • “LOCAL_TIMEZONE function” on page 288 • “EXTRACT function” on page 284 	YEAR, MONTH, DAY, HOUR, MINUTE, SECOND
Boolean evaluation for conditional statements		
Functions that return a Boolean value	<ul style="list-style-type: none"> • BETWEEN, see “ESQL simple comparison operators” on page 181 for details. • EXISTS, see “EXISTS function” on page 327 for details. • IN, see “ESQL simple comparison operators” on page 181 for details. • LIKE, see “ESQL simple comparison operators” on page 181 for details. • “NULLIF function” on page 373 • “LASTMOVE function” on page 326 • “SAMEFIELD function” on page 326 • SINGULAR, see “SINGULAR function” on page 328 for details. 	SYMMETRIC, ASYMMETRIC, AND
Database interaction		
Actions on tables	<ul style="list-style-type: none"> • “PASSTHRU function” on page 374 • “SELECT function” on page 345 	<ul style="list-style-type: none"> • FROM, AS, ITEM, THE, SUM, COUNT, MAX, MIN
Results of actions	<ul style="list-style-type: none"> • “SQLCODE function” on page 279 • “SQLERRORTEXT function” on page 279 • “SQLNATIVEERROR function” on page 280 • “SQLSTATE function” on page 281 	

Calling ESQL functions

Most ESQL functions belong to a schema called SQL and this is particularly useful if you have functions with the same name. For example, if you have created a function called SQRT, you can code:

```
/* call my SQRT function */
SET Variable1=SQRT (4);
/* call the SQL supplied function */
SET Variable2=SQL.SQRT (144);
```

Most of the functions described in this section impose restrictions on the data types of the arguments that can be passed to the function. If the values passed to

the functions do not match the required data types, errors are generated at node configuration time whenever possible. Otherwise runtime errors are generated when the function is evaluated.

ESQL database state functions

ESQL provides four functions to return database state. These are:

- “SQLCODE function”
- “SQLERRORTEXT function”
- “SQLNATIVEERROR function” on page 280
- “SQLSTATE function” on page 281

SQLCODE function

SQLCODE is a database state function that returns an INTEGER data type with a default value of 0 (zero).

Syntax

```
▶—SQLCODE—▶
```

Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If this check box is selected, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the ExceptionList. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception and you must include the THROW statement to throw an exception if a certain SQL state code is not expected. See “THROW statement” on page 270 for a description of THROW.

If you choose to handle database errors in a node, you can use the database state function SQLCODE to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node's ESQL to recognize and handle possible errors.

SQLERRORTEXT function

SQLERRORTEXT is a database state function that returns a CHARACTER data type with a default value of " (empty string).

Syntax

A rectangular box containing the text 'SQLERRORTEXT' with a double arrow pointing left at the start and a double arrow pointing right at the end, indicating the function name and its scope.

Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

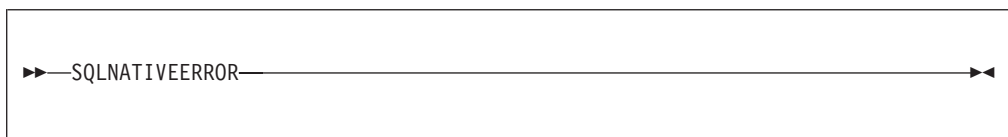
When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If you have selected this check box, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the ExceptionList. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception and you must include the THROW statement to throw an exception if a certain SQL state code is not expected. See "THROW statement" on page 270 for a description of THROW.

If you choose to handle database errors in a node, you can use the database state function SQLERRORTEXT to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node's ESQL to recognize and handle possible errors.

SQLNATIVEERROR function

SQLNATIVEERROR is a database state function that returns an INTEGER data type with a default value of 0 (zero).

Syntax

A rectangular box containing the text 'SQLNATIVEERROR' with a double arrow pointing left at the start and a double arrow pointing right at the end, indicating the function name and its scope.

Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Filter, Database, and Compute nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property *Throw exception on database error*. If you have selected this check box, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the ExceptionList. If you want to override the default behavior and handle a database error in the ESQL in the node, clear the *Throw exception on database error* check box. The broker does not throw an exception

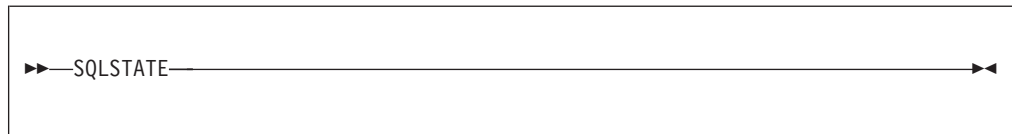
and you must include the `THROW` statement to throw an exception if a certain SQL state code is not expected. See “`THROW` statement” on page 270 for a description of `THROW`.

If you choose to handle database errors in a node, you can use the database state function `SQLNATIVEERROR` to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node's ESQL to recognize and handle possible errors.

SQLSTATE function

`SQLSTATE` is a database state function that returns a 5 character data type of `CHARACTER` with a default value of '00000' (five zeros as a string).

Syntax



Within a message flow, you can access and update an external database resource using the available ESQL database functions in the Compute, Database, and Filter nodes. When making calls to an external database, you might get errors, such as a table does not exist, a database is not available, or an insert for a key that already exists.

When these errors occur, the default action of the broker is to generate an exception. This behavior is determined by how you have set the property `Throw exception on database error`. If you select this property, the broker stops processing the node, propagates the message to the node's failure terminal, and writes the details of the error to the `ExceptionList`. If you want to override the default behavior and handle a database error in the ESQL in the node, clear `Throw exception on database error`. The broker does not throw an exception and you must include the `THROW` statement to throw an exception if a certain SQL state code is not expected. See “`THROW` statement” on page 270 for a description of `THROW`.

To handle database errors in a node, you can use the database state function `SQLSTATE` to receive information about the status of the DBMS call made in ESQL. You can include it in conditional statements in current node's ESQL to recognize and handle possible errors.

SQL states

In ESQL, SQL states are variable length character strings. By convention, they are six characters long and contain only the characters 0-9, A-Z. The significance of the six characters is:

Char 1

The origin of the exception

Chars 2 - 3

The class of the exception

Chars 4 - 6

The subclass of the exception

The SQL state of an exception is determined by a two stage process. In the **first stage**, the exception information is examined and any wrapping exceptions (that is,

information that says what the broker was doing at the time the exception occurred) is stepped over until the exception that describes the original error is located.

The **second stage** is as follows:

1. If the selected exception is a database exception, the SQL state is that supplied by the database, but prefixed by the letter "D" to avoid any confusion with exceptions arising in the broker. The SQL code, native error, and error text are those supplied by the database.
2. If the selected exception is a user exception (that is, it originated in a THROW statement), the SQL code, state, native error, and error text are taken from the first four inserts of the exception, in order. The resulting state value is taken as is (not prefixed by a letter such as "U"). The letter "U" is not used by the broker as an origin indicator. If you want to define a unique SQL state rather than to imitate an existing one, use SQL states starting with the letter "U". If you use SQL states that start with the letter "U", you can write an error handler to match all user-defined and thrown exceptions with a LIKE'U%' operator.
3. If the selected exception originated in the message transport or in the ESQL implementation itself, the SQL code, state, native error, and error text are as described in the list below.
4. For all other exceptions, the SQL state is "", indicating no origin, no class, and no subclass.

Some exceptions that currently give an empty SQL state might give individual states in future releases. If you want to catch unclassified exceptions, use the "all" wildcard ("%") for the SQL state on the last handler of a scope. This wildcard will continue to catch the same set of exceptions if previously unclassified exceptions are given new unique SQL states.

The following SQL states are defined:

Ddddd

dddd is the state returned by the database.

SqlState = 'S22003'

Arithmetic overflow. An operation whose result is a numeric type resulted in a value beyond the range supported.

SqlState = 'S22007'

Date time format not valid. A character string used in a cast from character to a datetime type had either the wrong basic format (for example, '01947-10-24') or had values outside the ranges allowed by the Gregorian calendar (for example, '1947-21-24').

SqlState = 'S22008'

Date time field overflow. An operation whose result is a datetime type resulted in a value beyond the range supported.

SqlState = 'S22012'

Divide by zero. A divide operation whose result data type has no concept of infinity had a zero right operand.

SqlState = 'S22015'

Interval field overflow. An operation whose result is of type INTERVAL resulted in a value beyond the range supported by the INTERVAL data type.

SqlState = 'S22018'

Character value for cast not valid.

SqlState = 'SPS001'

Target terminal not valid. A PROPAGATE to terminal statement attempted to use a terminal name that is not valid.

SqlState = 'SPS002'

Target label not valid. A PROPAGATE to label statement attempted to use a label that is not valid.

SqlState = 'MQW001', SqlNativeError = 0

The bit stream does not meet the requirements for WebSphere MQ messages. No attempt was made to put it to a queue. Retrying and queue administration does not resolve this problem.

SqlState = 'MQW002', SqlNativeError = 0

The target queue or queue manager names were not valid (that is, they could not be converted from Unicode to the queue manager's code page). Retrying and queue emptying does not resolve this problem.

SqlState = 'MQW003', SqlNativeError = 0

Request mode was specified but the "reply to" queue or queue manager names were not valid (that is, could not be converted from Unicode to the message's code page). Retrying and queue emptying does not resolve this problem.

SqlState = 'MQW004', SqlNativeError = 0

Reply mode was specified but the queue or queue manager names taken from the message were not valid (that is, they could not be converted from the given code page to Unicode). Retrying and queue emptying does not resolve this problem.

SqlState = 'MQW005', SqlNativeError = 0

Destination list mode was specified but the destination list supplied does not meet the basic requirements for destination lists. No attempt was made to put any message to a queue. Retrying and queue administration does not resolve this problem.

SqlState = 'MQW101', SqlNativeError = returned by WebSphere MQ

The target queue manager or queue could not be opened. Queue administration might resolve this problem but retrying does not.

SqlState = 'MQW102', SqlNativeError = returned by WebSphere MQ

The target queue manager or queue could not be written to. Retrying and queue administration might resolve this problem.

SqlState = 'MQW201', SqlNativeError = number of destinations with an error

More than one error occurred while processing a destination list. The message might have been put to zero or more queues. Retrying and queue administration might resolve this problem.

Anything that the user has used in a THROW statement

Use Uuuuuuu for user exceptions, unless imitating one of the exceptions defined above.

Empty string

All other errors.

ESQL datetime functions

This topic lists the ESQL datetime functions.

In addition to the functions described here, you can use arithmetic operators to perform various calculations on datetime values. For example, you can use the - (minus) operator to calculate the difference between two dates as an interval, or you can add an interval to a timestamp.

This section covers the following topics:

“EXTRACT function”

“CURRENT_DATE function” on page 286

“CURRENT_TIME function” on page 286

“CURRENT_TIMESTAMP function” on page 286

“CURRENT_GMTDATE function” on page 287

“CURRENT_GMTTIME function” on page 287

“CURRENT_GMTTIMESTAMP function” on page 287

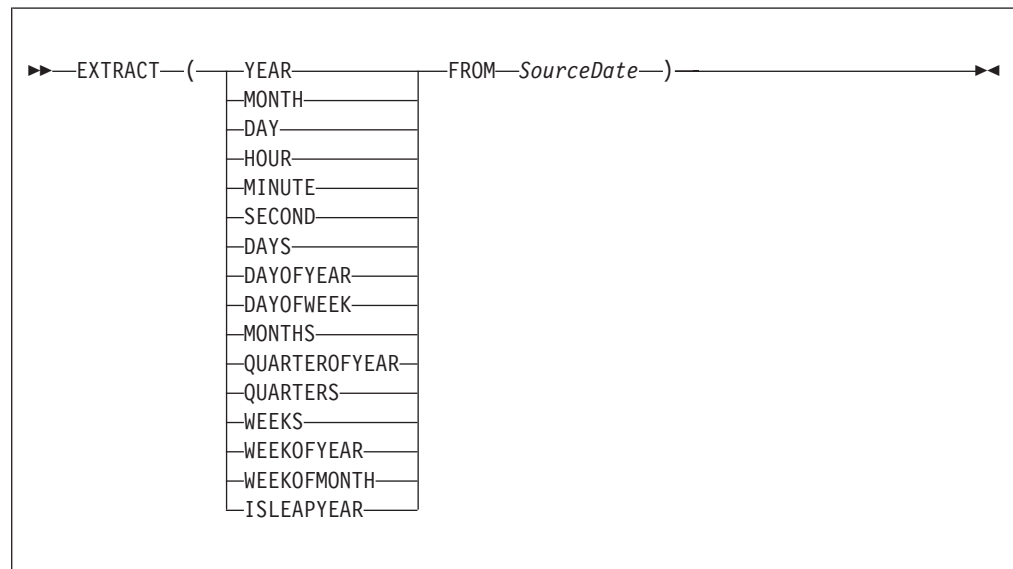
“LOCAL_TIMEZONE function” on page 288

EXTRACT function

The EXTRACT function extracts fields (or calculates values) from datetime values and intervals.

The result is INTEGER for YEAR, MONTH, DAY, HOUR, MINUTE, DAYS, DAYOFYEAR, DAYOFWEEK, MONTHS, QUARTEROFYEAR, QUARTERS, WEEKS, WEEKOFYEAR, and WEEKOFMONTH extracts, but FLOAT for SECOND extracts, and BOOLEAN for ISLEAPYEAR extracts. If the *SourceDate* is NULL, the result is NULL regardless of the type of extract.

Syntax



EXTRACT extracts individual fields from datetime values and intervals. You can extract a field only if it is present in the datetime value specified in the second parameter. Either a parse-time or a runtime error is generated if the requested field does not exist within the data type.

The following table describes the extracts that are supported:

Note: All new integer values start from 1.

Table 5.

Extract	Description
YEAR	Year
MONTH	Month
DAY	Day
HOUR	Hour
MINUTE	Minute
SECOND	Second
DAYS	Days encountered between 1st January 0001 and the <i>SourceDate</i> .
DAYOFYEAR	Day of year
DAYOFWEEK	Day of the week: Sunday = 1, Monday = 2, Tuesday = 3, Wednesday = 4, Thursday = 5, Friday = 6, Saturday = 7.
MONTHS	Months encountered between 1st January 0001 and the <i>SourceDate</i> .
QUARTEROFYEAR	Quarter of year: January to March = 1, April to June = 2, July to September = 3, October to December = 4.
QUARTERS	Quarters encountered between 1st January 0001 and the <i>SourceDate</i> .
WEEKS	Weeks encountered between 1st January 0001 and the <i>SourceDate</i> .
WEEKOFYEAR	Week of year
WEEKOFMONTH	Week of month
ISLEAPYEAR	Whether this is a leap year

Notes:

1. A week is defined as Sunday to Saturday, not any seven consecutive days. You must convert to an alternative representation scheme if required.
2. The source date time epoch is 1 January 0001. Dates before the epoch are not valid for this function.
3. The Gregorian calendar is assumed for calculation.

Example

`EXTRACT(YEAR FROM CURRENT_DATE)`

and

`EXTRACT(HOUR FROM LOCAL_TIMEZONE)`

both work without error, but

`EXTRACT(DAY FROM CURRENT_TIME)`

fails.

```
EXTRACT (DAYS FROM DATE '2000-02-29')
```

calculates the number of days encountered since year 1 to '2000-02-29' and

```
EXTRACT (DAYOFYEAR FROM CURRENT_DATE)
```

calculates the number of days encountered since the beginning of the current year but

```
EXTRACT (DAYOFYEAR FROM CURRENT_TIME)
```

fails because `CURRENT_TIME` does not contain date information.

CURRENT_DATE function

The `CURRENT_DATE` datetime function returns the current date.

Syntax

```
▶▶—CURRENT_DATE—◀◀
```

`CURRENT_DATE` returns a `DATE` value representing the current date in local time. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to `CURRENT_DATE` within the processing of one node are guaranteed to return the same value.

CURRENT_TIME function

The `CURRENT_TIME` datetime function returns the current local time.

Syntax

```
▶▶—CURRENT_TIME—◀◀
```

`CURRENT_TIME` returns a `TIME` value representing the current local time. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to `CURRENT_TIME` within the processing of one node are guaranteed to return the same value.

CURRENT_TIMESTAMP function

The `CURRENT_TIMESTAMP` datetime function returns the current date and local time.

Syntax

```
▶▶—CURRENT_TIMESTAMP—◀◀
```

CURRENT_TIMESTAMP returns a TIMESTAMP value representing the current date and local time. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_TIMESTAMP within the processing of one node are guaranteed to return the same value.

Example

To obtain the following XML output message:

```
<Body>
<Message>Hello World</Message>
<DateStamp>2006-02-01 13:13:56.444730</DateStamp>
</Body>
```

use the following ESQL:

```
SET OutputRoot.XMLNS.Body.Message = 'Hello World';
SET OutputRoot.XMLNS.Body.DateStamp = CURRENT_TIMESTAMP;
```

CURRENT_GMTDATE function

The CURRENT_GMTDATE datetime function returns the current date in the GMT time zone.

Syntax

```
▶▶—CURRENT_GMTDATE——▶▶
```

CURRENT_GMTDATE returns a DATE value representing the current date in the GMT time zone. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_GMTDATE within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIME function

The CURRENT_GMTTIME datetime function returns the current time in the GMT time zone.

Syntax

```
▶▶—CURRENT_GMTTIME——▶▶
```

It returns a GMTTIME value representing the current time in the GMT time zone. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to CURRENT_GMTTIME within the processing of one node are guaranteed to return the same value.

CURRENT_GMTTIMESTAMP function

The CURRENT_GMTTIMESTAMP datetime function returns the current date and time in the GMT time zone.

Syntax



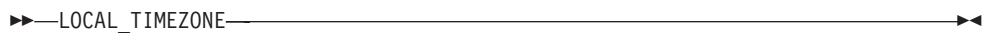
A rectangular box containing the text 'CURRENT_GMTTIMESTAMP' with a double-headed arrow pointing to the right, indicating the function name and its return type.

`CURRENT_GMTTIMESTAMP` returns a `GMTTIMESTAMP` value representing the current date and time in the GMT time zone. As with all SQL functions that take no parameters, no parentheses are required or accepted. All calls to `CURRENT_GMTTIMESTAMP` within the processing of one node are guaranteed to return the same value.

LOCAL_TIMEZONE function

The `LOCAL_TIMEZONE` datetime function returns the displacement of the local time zone from GMT.

Syntax



A rectangular box containing the text 'LOCAL_TIMEZONE' with a double-headed arrow pointing to the right, indicating the function name and its return type.

`LOCAL_TIMEZONE` returns an interval value representing the local time zone displacement from GMT. As with all SQL functions that take no parameters, no parentheses are required or accepted. The value returned is an interval in hours and minutes representing the displacement of the current time zone from Greenwich Mean Time. The sign of the interval is such that a local time can be converted to a time in GMT by subtracting the result of the `LOCAL_TIMEZONE` function.

ESQL numeric functions

This topic lists the ESQL numeric functions and covers the following:

“ABS and ABSVAL functions” on page 289

“ACOS function” on page 290

“ASIN function” on page 290

“ATAN function” on page 290

“ATAN2 function” on page 290

“BITAND function” on page 291

“BITNOT function” on page 291

“BITOR function” on page 292

“BITXOR function” on page 292

“CEIL and CEILING functions” on page 293

“COS function” on page 293

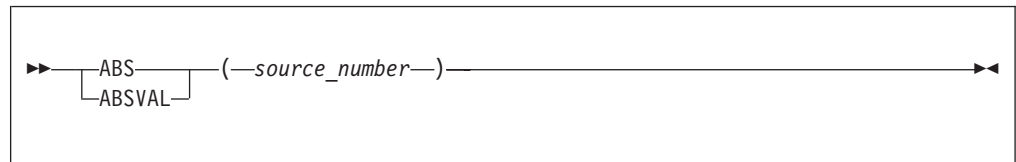
“COSH function” on page 294

"COT function" on page 294
 "DEGREES function" on page 294
 "EXP function" on page 294
 "FLOOR function" on page 295
 "LN and LOG functions" on page 295
 "LOG10 function" on page 296
 "MOD function" on page 296
 "POWER function" on page 296
 "RADIANS function" on page 297
 "RAND function" on page 297
 "ROUND function" on page 297
 "SIGN function" on page 301
 "SIN function" on page 301
 "SINH function" on page 301
 "SQRT function" on page 301
 "TAN function" on page 302
 "TANH function" on page 302
 "TRUNCATE function" on page 303

ABS and ABSVAL functions

The ABS and ABSVAL numeric functions return the absolute value of a supplied number.

Syntax



The absolute value of the source number is a number with the same magnitude as the source but without a sign. The parameter must be a numeric value. The result is of the same type as the parameter unless it is NULL, in which case the result is NULL.

For example:

```
ABS( -3.7 )
```

returns 3.7

```
ABS( 3.7 )
```

returns 3.7

```
ABS( 1024 )
```

returns 1024

ACOS function

The ACOS numeric function returns the angle of a given cosine.

Syntax

```
▶▶ ACOS (NumericExpression) ▶▶
```

The ACOS function returns the angle, in radians, whose cosine is the given *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ASIN function

The ASIN numeric function returns the angle of the given sine.

Syntax

```
▶▶ ASIN (NumericExpression) ▶▶
```

The ASIN function returns the angle, in radians, whose sine is the given *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ATAN function

The ATAN numeric function returns the angle of the given tangent.

Syntax

```
▶▶ ATAN (NumericExpression) ▶▶
```

The ATAN function returns the angle, in radians, whose tangent is the given *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ATAN2 function

The ATAN2 numeric function returns the angle subtended in a right angled triangle between an opposite and the base.

Syntax

```
▶▶ ATAN2 ( OppositeNumericExpression , BaseNumericExpression ) ▶▶
```

The ATAN2 function returns the angle, in radians, subtended (in a right angled triangle) by an opposite given by *OppositeNumericExpression* and the base given by *BaseNumericExpression*. The parameters can be any built-in numeric data type. The result is FLOAT unless either parameter is NULL, in which case the result is NULL

BITAND function

The BITAND numeric function performs a bitwise AND on the binary representation of two or more numbers.

Syntax

```
▶▶ BITAND ( source_integer , source_integer ) ▶▶
```

BITAND takes two or more integer values and returns the result of performing the bitwise AND on the binary representation of the numbers. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

```
BITAND(12, 7)
```

returns 4 as shown by this worked example:

	Binary	Decimal
	1100	12
AND	0111	7

	0100	4

BITNOT function

The BITNOT numeric function performs a bitwise complement on the binary representation of a number.

Syntax

```
▶▶ BITNOT ( source_integer ) ▶▶
```

BITNOT takes an integer value and returns the result of performing the bitwise complement on the binary representation of the number. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

```
BITNOT(7)
```

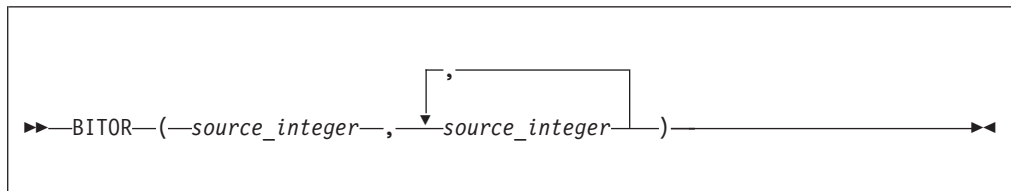
returns -8, as shown by this worked example:

	Binary	Decimal
	00...0111	7
NOT	<u>11...1000</u>	-8

BITOR function

The BITOR numeric function performs a bitwise OR on the binary representation of two or more numbers.

Syntax



BITOR takes two or more integer values and returns the result of performing the bitwise OR on the binary representation of the numbers. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

```
BITOR(12, 7)
```

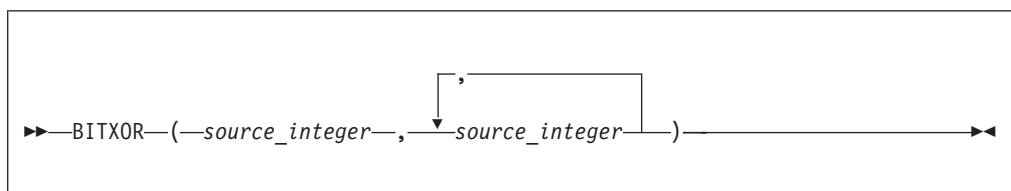
returns 15, as shown by this worked example:

	Binary	Decimal
	1100	12
OR	0111	7
	<u>1111</u>	15

BITXOR function

The BITXOR numeric function performs a bitwise XOR on the binary representation of two or more numbers.

Syntax



BITXOR takes two or more integer values and returns the result of performing the bitwise XOR on the binary representation of the numbers. The result is INTEGER unless either parameter is NULL, in which case the result is NULL.

For example:

```
BITXOR(12, 7)
```

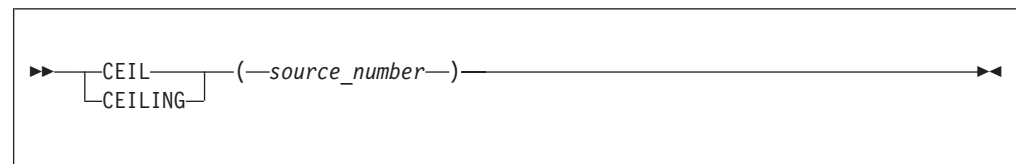
returns 11, as shown by this worked example:

	Binary	Decimal
	1100	12
XOR	0111	7
<hr/>		
	1011	11

CEIL and CEILING functions

The CEIL and CEILING numeric functions return the smallest integer equivalent of a decimal number.

Syntax



CEIL and CEILING return the smallest integer value greater than or equal to `source_number`. The parameter can be any numeric data type. The result is of the same type as the parameter unless it is NULL, in which case the result is NULL.

For example:

```
CEIL(1)
```

returns 1

```
CEIL(1.2)
```

returns 2.0

```
CEIL(-1.2)
```

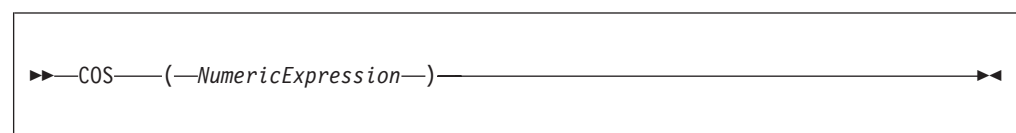
returns -1.0

If possible, the scale is changed to zero. If the result cannot be represented at that scale, it is made sufficiently large to represent the number.

COS function

The COS numeric function returns the cosine of a given angle.

Syntax



The COS function returns the cosine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

COSH function

The COSH numeric function returns the hyperbolic cosine of a given angle.

Syntax

```
►► COSH (—NumericExpression—) ◀◀
```

The COSH function returns the hyperbolic cosine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

COT function

The COT numeric function returns the cotangent of a given angle.

Syntax

```
►► COT (—NumericExpression—) ◀◀
```

The COT function returns the cotangent of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

DEGREES function

The DEGREES numeric function returns the angle of the radians supplied.

Syntax

```
►► DEGREES (—NumericExpression—) ◀◀
```

The DEGREES function returns the angle, in degrees, specified by *NumericExpression* in radians. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

EXP function

The EXP numeric function returns the exponential value of a given number.

Syntax



►► EXP (—*NumericExpression*—) ◄◄

The EXP function returns the exponential of the value specified by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

FLOOR function

The FLOOR numeric function returns the largest integer equivalent to a given decimal number.

Syntax



►► FLOOR (—*source_number*—) ◄◄

FLOOR returns the largest integer value less than or equal to *source_number*. The parameter can be any numeric data type. The result is of the same type as the parameter unless it is NULL, in which case the result is NULL.

For example:

FLOOR(1)

returns 1

FLOOR(1.2)

returns 1.0

FLOOR(-1.2)

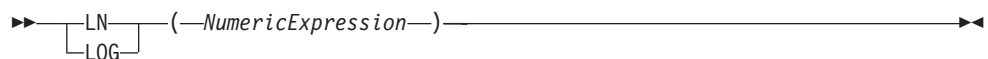
returns -2.0

If possible, the scale is changed to zero. If the result cannot be represented at that scale, it is made sufficiently large to represent the number.

LN and LOG functions

The LN and LOG equivalent numeric functions return the natural logarithm of a given value.

Syntax



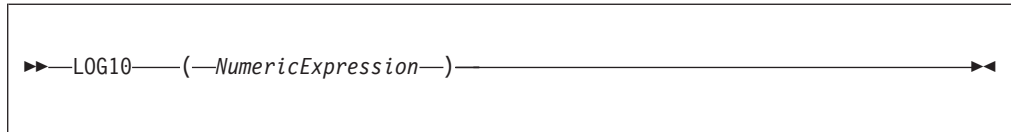
►► LN (—*NumericExpression*—) ◄◄
LOG (—*NumericExpression*—) ◄◄

The LN and LOG functions return the natural logarithm of the value specified by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

LOG10 function

The LOG10 numeric function returns the logarithm to base 10 of a given value.

Syntax

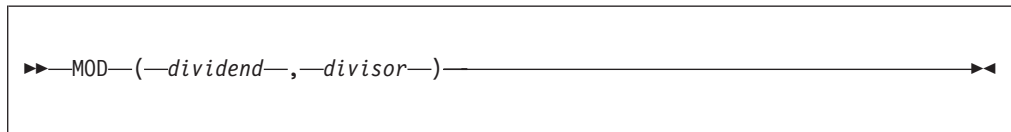


The LOG10 function returns the logarithm to base 10 of the value specified by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

MOD function

The MOD numeric function returns the remainder when dividing two numbers.

Syntax



MOD returns the remainder when the first parameter is divided by the second parameter. The result is negative only if the first parameter is negative. Parameters must be integers. The function returns an integer. If any parameter is NULL, the result is NULL.

For example:

```
MOD(7, 3)
```

returns 1

```
MOD(-7, 3)
```

returns -1

```
MOD(7, -3)
```

returns 1

```
MOD(6, 3)
```

returns 0

POWER function

The POWER numeric function raises a value to the power supplied.

Syntax

```
▶▶ POWER ( ValueNumericExpression , PowerNumericExpression ) ▶▶
```

POWER returns the given value raised to the given power. The parameters can be any built-in numeric data type. The result is FLOAT unless any parameter is NULL, in which case the result is NULL

An exception occurs, if the value is either:

- Zero and the power is negative, or
- Negative and the power is not an integer

RADIANS function

The RADIANS numeric function returns a given radians angle in degrees.

Syntax

```
▶▶ RADIANS ( NumericExpression ) ▶▶
```

The RADIANS function returns the angle, in radians, specified by *NumericExpression* in degrees. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

RAND function

The RAND numeric function returns a pseudo random number.

Syntax

```
▶▶ RAND ( IntegerExpression ) ▶▶
```

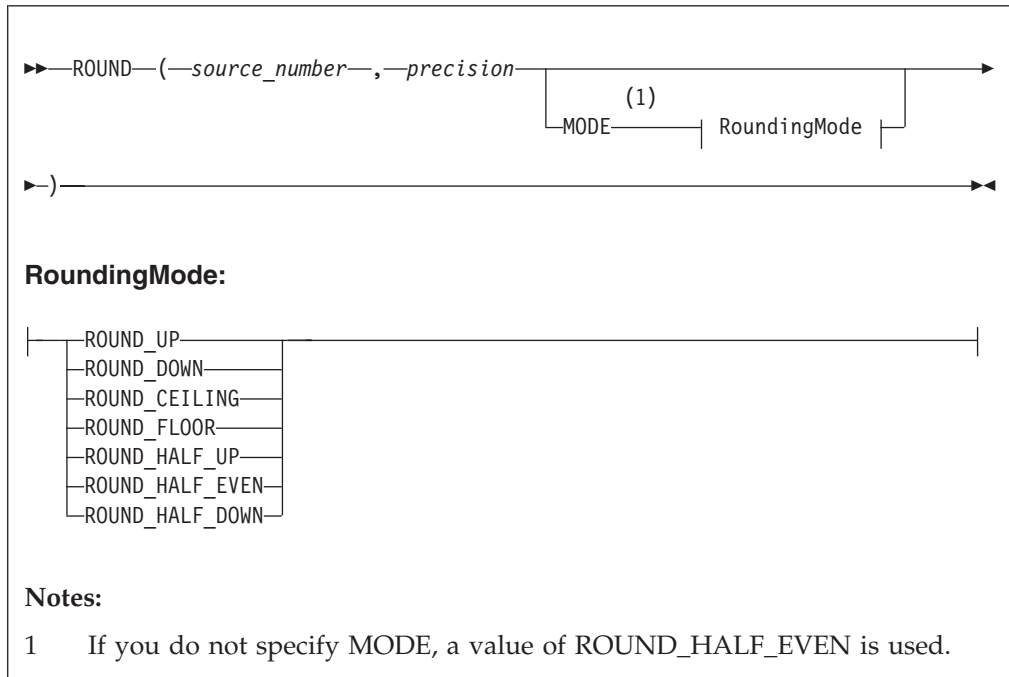
The RAND function returns a pseudo random number in the range 0.0 to 1.0. If supplied, the parameter initializes the pseudo random sequence.

The parameter can be of any numeric data type, but any fractional part is ignored. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

ROUND function

The ROUND numeric function rounds a supplied value to a given number of places.

Syntax



If *precision* is a positive number, *source_number* is rounded to *precision* places right of the decimal point. If *precision* is negative, the result is *source_number* rounded to the absolute value of *precision* places to the left of the decimal point.

source_number can be any built-in numeric data type; *precision* must be an integer. The result is of the same data type as the *source_number* parameter unless *source_number* is NULL, in which case the result is NULL.

This means that the result of the function is:

- INTEGER if *source_number* is INTEGER
- FLOAT if *source_number* is FLOAT
- DECIMAL if *source_number* is DECIMAL

When rounding, the *banker's* or *half-even symmetric* rounding rules are used by default, unless a *RoundingMode* is specified.

RoundingMode

RoundingMode can take one of the following values:

ROUND_UP

Round away from zero. Always increments the digit prior to a nonzero discarded fraction. This rounding mode never decreases the magnitude of the calculated value.

ROUND_DOWN

Round towards zero. Never increments the digit prior to a discarded fraction, that is, truncates. This rounding mode never increases the magnitude of the calculated value.

ROUND_CEILING

Round towards positive infinity. If the decimal is positive, behaves as for

ROUND_UP; if negative, behaves as for ROUND_DOWN. This rounding mode never decreases the calculated value.

ROUND_FLOOR

Round towards negative infinity. If the decimal is positive, behaves as for ROUND_DOWN; if negative, behaves as for ROUND_UP. This rounding mode never increases the calculated value.

ROUND_HALF_UP

Round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up. Behaves as for ROUND_UP if the discarded fraction is greater than, or equal to, 0.5; otherwise, behaves as for ROUND_DOWN. This is the rounding mode that is typically taught in schools.

ROUND_HALF_DOWN

Round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down. Behaves as for ROUND_UP if the discarded fraction is greater than 0.5; otherwise, behaves as for ROUND_DOWN.

ROUND_HALF_EVEN

Round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor. Behaves as for ROUND_HALF_UP if the digit to the left of the discarded fraction is odd; behaves as for ROUND_HALF_DOWN if it is even. This is the rounding mode that minimizes cumulative error when applied repeatedly over a sequence of calculations, and is sometimes referred to as Banker's rounding.

The following table gives a summary of rounding operations, with a precision of zero, under different rounding modes.

Input number	ROUND UP	ROUND DOWN	ROUND CEILING	ROUND FLOOR	ROUND HALF UP	ROUND HALF DOWN	ROUND HALF EVEN
5.5	6	5	6	5	6	5	6
2.5	3	2	3	2	3	2	2
1.6	2	1	2	1	2	2	2
1.1	2	1	2	1	1	1	1
1.0	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1
-1.1	-2	-1	-1	-2	-1	-1	-1
-1.6	-2	-1	-1	-2	-2	-2	-2
-2.5	-3	-2	-2	-3	-3	-2	-2
-5.5	-6	-5	-5	-6	-6	-5	-6

Examples using the default rounding mode (ROUND_HALF_EVEN):

ROUND(27.75, 2)

returns 27.75

ROUND(27.75, 1)

returns 27.8

ROUND(27.75, 0)

returns 28
ROUND(27.75, -1)

returns 30

Examples using a rounding mode with a precision of zero:

ROUND(5.5, 0 MODE ROUND_UP);

returns 6
ROUND(5.5, 0 MODE ROUND_DOWN);

returns 5
ROUND(5.5, 0 MODE ROUND_CEILING);

returns 6
ROUND(5.5, 0 MODE ROUND_FLOOR);

returns 5
ROUND(5.5, 0 MODE ROUND_HALF_UP);

returns 6
ROUND(5.5, 0 MODE ROUND_HALF_DOWN);

returns 5
ROUND(5.5, 0 MODE ROUND_HALF_EVEN);

returns 6
ROUND(2.5, 0 MODE ROUND_UP);

returns 3
ROUND(2.5, 0 MODE ROUND_DOWN);

returns 2
ROUND(2.5, 0 MODE ROUND_CEILING);

returns 3
ROUND(2.5, 0 MODE ROUND_FLOOR);

returns 2
ROUND(2.5, 0 MODE ROUND_HALF_UP);

returns 3
ROUND(2.5, 0 MODE ROUND_HALF_DOWN);

returns 2
ROUND(2.5, 0 MODE ROUND_HALF_EVEN);

returns 3

If possible, the scale is changed to the given value. If the result cannot be represented within the given scale, it is INFINITY.

SIGN function

The SIGN numeric function tells you whether a given number is positive, negative, or zero.

Syntax

```
▶▶SIGN(—NumericExpression—)▶▶
```

The SIGN function returns -1, 0, or +1 when the *NumericExpression* value is negative, zero, or positive respectively. The parameter can be any built-in numeric data type and the result is of the same type as the parameter. If the parameter is NULL, the result is NULL.

SIN function

The SIN numeric function returns the sine of a given angle.

Syntax

```
▶▶SIN(—NumericExpression—)▶▶
```

The SIN function returns the sine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

SINH function

The SINH numeric function returns the hyperbolic sine of a given angle.

Syntax

```
▶▶SINH(—NumericExpression—)▶▶
```

The SINH function returns the hyperbolic sine of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

SQRT function

The SQRT numeric function returns the square root of a given number.

Syntax

```
▶▶—Sqrt—(—source_number—)————▶▶
```

Sqrt returns the square root of *source_number*. The parameter can be any built-in numeric data type. The result is a FLOAT. If the parameter is NULL, the result is NULL.

For example:

```
Sqrt(4)
```

returns 2E+1

```
Sqrt(2)
```

returns 1.414213562373095E+0

```
Sqrt(-1)
```

throws an exception.

TAN function

The TAN numeric function returns the tangent of a given angle.

Syntax

```
▶▶—Tan—(—NumericExpression—)————▶▶
```

The TAN function returns the tangent of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

TANH function

The TANH numeric function returns the hyperbolic tangent of an angle.

Syntax

```
▶▶—Tanh—(—NumericExpression—)————▶▶
```

The TANH function returns the hyperbolic tangent of the angle, in radians, given by *NumericExpression*. The parameter can be any built-in numeric data type. The result is FLOAT unless the parameter is NULL, in which case the result is NULL.

TRUNCATE function

The TRUNCATE numeric function truncates a supplied decimal number a specified number of places.

Syntax

```
▶▶—TRUNCATE—(—source_number—,—precision—)————▶▶
```

If *precision* is positive, the result of the TRUNCATE function is *source_number* truncated to *precision* places right of the decimal point. If *precision* is negative, the result is *source_number* truncated to the absolute value of *precision* places to the left of the decimal point.

source_number can be any built-in numeric data type. *precision* must evaluate to an INTEGER. The result is of the same data type as *source_number*. If any parameter is NULL, the result is NULL.

For example:

```
TRUNCATE(27.75, 2)
```

returns 27.75

```
TRUNCATE(27.75, 1)
```

returns 27.7

```
TRUNCATE(27.75, 0)
```

returns 27.0

```
TRUNCATE(27.75, -1)
```

returns 20.0

If possible, the scale is changed to the given value. If the result cannot be represented within the given scale, it is INF.

ESQL string manipulation functions

A list of the ESQL string manipulation functions that you can use.

Most of the following functions manipulate all string data types (BIT, BLOB, and CHARACTER). Exceptions to this are UPPER, LOWER, LCASE, UCASE, and SPACE, which operate only on character strings.

In these descriptions, the term *singleton* refers to a single part (BIT, BLOB, or CHARACTER) within a string of that type.

In addition to the functions that are described here, you can use the logical OR operator to perform various calculations on ESQL string manipulation values.

To concatenate two strings, use the “ESQL string operator” on page 187.

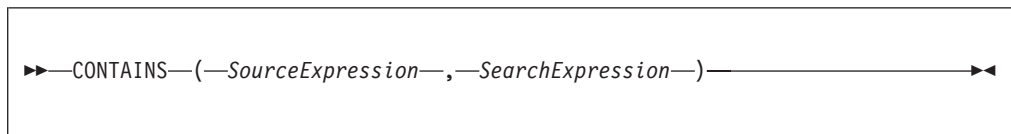
This section covers the following topics:

- “CONTAINS function”
- “ENDSWITH function” on page 305
- “LEFT function” on page 305
- “LENGTH function” on page 305
- “LOWER and LCASE functions” on page 306
- “LTRIM function” on page 306
- “OVERLAY function” on page 307
- “POSITION function” on page 308
- “REPLACE function” on page 309
- “REPLICATE function” on page 309
- “RIGHT function” on page 310
- “RTRIM function” on page 310
- “SPACE function” on page 311
- “STARTSWITH function” on page 311
- “SUBSTRING function” on page 312
- “TRANSLATE function” on page 313
- “TRIM function” on page 314
- “UPPER and UCASE functions” on page 315

CONTAINS function

CONTAINS is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and returns a Boolean value to indicate whether one string is present within another.

Syntax



CONTAINS returns TRUE if the *SearchExpression* is present within the *SourceExpression*, otherwise it returns FALSE.

The parameter strings for both *SourceExpression* and *SearchExpression* can be of the CHARACTER, BLOB, or BIT data type, but must be of the same data type.

If any parameter is NULL, the result is NULL.

Examples

```
CONTAINS('Hello World!', 'ello');
```

returns TRUE.

```
CONTAINS('Hello World!', 'daisy');
```

returns FALSE.

ENDSWITH function

ENDSWITH is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and returns a Boolean value to indicate whether one string ends with another.

Syntax

```
▶▶—ENDSWITH—(—SourceExpression—,—SearchExpression—)————▶▶
```

ENDSWITH returns TRUE if *SourceExpression* ends with *SearchExpression*, otherwise it returns FALSE.

The parameter strings for both *SearchExpression* and *SourceExpression* can be of the CHARACTER, BLOB, or BIT data type, but must be of the same data type.

If any parameter is NULL, the result is NULL.

Examples

```
ENDSWITH('Hello World!', 'World!');
```

returns TRUE.

```
ENDSWITH('Hello World!', 'World');
```

returns FALSE.

LEFT function

LEFT is a string manipulation function that returns a string consisting of the source string truncated to the length given by the length expression.

Syntax

```
▶▶—LEFT—(—source_string—,—LengthIntegerExpression—)————▶▶
```

The source string can be of the CHARACTER, BLOB or BIT data type and the length must be of type INTEGER. The truncation discards the final characters of the *source_string*

The result is of the same type as the source string. If the length is negative or zero, a zero length string is returned. If either parameter is NULL, the result is NULL

LENGTH function

The LENGTH function is used for string manipulation on all string data types (BIT, BLOB, and CHARACTER) and returns an integer value giving the number of singletons in *source_string*.

Syntax



If the *source_string* is NULL, the result is the NULL value. The term *singleton* refers to a single part (BIT, BYTE, or CHARACTER) within a string of that type.

For example:

```
LENGTH('Hello World!');
```

returns 12.

```
LENGTH('');
```

returns 0.

LOWER and LCASE functions

The LOWER and LCASE functions are equivalent, and manipulate CHARACTER string data; they both return a new character string, which is identical to *source_string*, except that all uppercase letters are replaced with the corresponding lowercase letters.

Syntax



For example:

```
LOWER('Mr Smith')
```

returns 'mr smith'.

```
LOWER('22 Railway Cuttings')
```

returns '22 railway cuttings'.

```
LCASE('ABCD')
```

returns 'abcd'.

LTRIM function

LTRIM is a string manipulation function, used for manipulating all data types (BIT, BLOB, and CHARACTER), that returns a character string value of the same data type and content as *source_string*, but with any leading default singletons removed.

Syntax

```
▶▶LTRIM(—source_string—)▶▶
```

The term *singleton* is used to refer to a single part (BIT, BLOB, or CHARACTER) within a string of that type.

The LTRIM function is equivalent to TRIM(LEADING FROM *source_string*).

If the parameter is NULL, the result is NULL.

The default singleton depends on the data type of *source_string*:

Table 6.

Character	' ' (space)
BLOB	X'00'
Bit	B'0'

OVERLAY function

OVERLAY is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER) and replaces part of a string with a substring.

Syntax

```
▶▶OVERLAY(—source_string— PLACING —source_string2—  
▶ FROM —start_position—  
    └── FOR —string_length—)▶▶
```

OVERLAY returns a new string of the same type as the source and is identical to *source_string*, except that a given substring in the string, starting from the specified numeric position and of the given length, has been replaced by *source_string2*. When the length of the substring is zero, nothing is replaced.

For example:

```
OVERLAY ('ABCDEFGHIJ' PLACING '1234' FROM 4 FOR 3)
```

returns the string 'ABC1234GHIJ'

If any parameter is NULL, the result is NULL. If *string_length* is not specified, it is assumed to be equal to LENGTH(*source_string2*).

The result of the OVERLAY function is equivalent to:

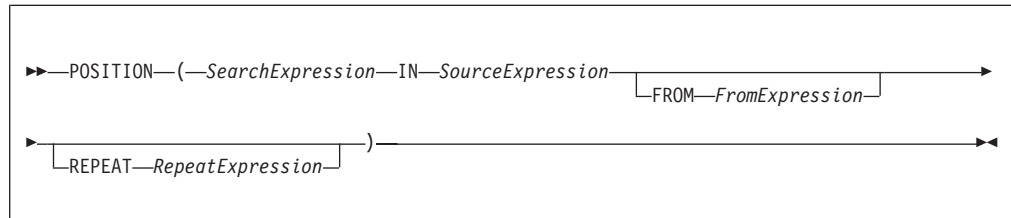
```
SUBSTRING(source_string FROM 1 FOR start_position -1 )  
|| source_string2 ||  
SUBSTRING(source_string FROM start_position + string_length)
```

where || is the concatenation operator.

POSITION function

POSITION is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and returns the position of one string within another.

Syntax



POSITION returns an integer giving the position of one string (*SearchExpression*) in a second string (*SourceExpression*). A position of one corresponds to the first character of the source string.

If present, the FROM clause gives a position within the search string at which the search commences. In the absence of a FROM clause, the source string is searched from the beginning.

If present, the REPEAT clause gives a repeat count, returning the position returned to be that of the nth occurrence of the search string within the source string. If the repeat count is negative, the source string is searched from the end.

In the absence of a REPEAT clause, a repeat count of +1 is assumed; that is, the position of the first occurrence, searching from the beginning is returned. If the search string has a length of zero, the result is one.

If the search string cannot be found, the result is zero: if the FROM clause is present, this applies only to the section of the source string being searched; if the REPEAT clause is present this applies only if there are insufficient occurrences of the string.

If any parameter is NULL, the result is NULL.

The search and source strings can be of the CHARACTER, BLOB, or BIT data types but they must be of the same type.

For example:

```
POSITION('Village' IN 'Hursley Village'); returns 9
POSITION('Town' IN 'Hursley Village'); returns 0

POSITION ('B' IN 'ABCABCABCABCABC'); -> returns 2
POSITION ('D' IN 'ABCABCABCABCABC'); -> returns 0

POSITION ('A' IN 'ABCABCABCABCABC' FROM 4); -> returns 4
POSITION ('C' IN 'ABCABCABCABCABC' FROM 2); -> returns 3

POSITION ('B' IN 'ABCABCABCABCABC' REPEAT 2); -> returns 5
POSITION ('C' IN 'ABCABCABCABCABC' REPEAT 4); -> returns 12

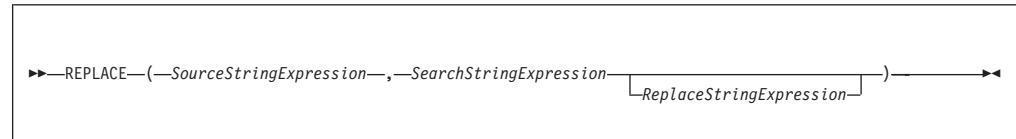
POSITION ('A' IN 'ABCABCABCABCABC' FROM 4 REPEAT 2); -> returns 7
POSITION ('AB' IN 'ABCABCABCABCABC' FROM 2 REPEAT 3); -> returns 10
```

```
POSITION ('A' IN 'ABCABCABCABC' REPEAT -2); -> returns 10
POSITION ('BC' IN 'ABCABCABCABC' FROM 2 REPEAT -3); -> returns 5
```

REPLACE function

REPLACE is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and replaces parts of a string with supplied substrings.

Syntax



REPLACE returns a string consisting of the source string, with each occurrence of the search string replaced by the replace string. The parameter strings can be of the CHARACTER, BLOB, or BIT data types, but all three must be of the same type.

If any parameter is NULL, the result is NULL.

The search process is single pass from the left and disregards characters that have already been matched.

If you do not specify the replace string expression, the replace string uses the default value of an empty string, and the behavior of the function is to delete all occurrences of the search string from the result.

The following examples give the results shown:

```
REPLACE('ABCDABCDABCD', 'A', 'AA')  
-- RESULT = AABCDAAABCDAA
```

The above example shows that replacement is single pass. Each occurrence of A is replaced by AA but these are not then expanded further.

```
REPLACE('AAAABCDEFHAAAABCDEFH', 'AA', 'A')  
-- RESULT = AABCDEFHABCDEFH
```

This example shows that after characters are matched, they are not considered further. Each occurrence of AA is replaced by A. The resulting AA pairs are not matched.

```
REPLACE('AAAAABCDEFHAAAABCDEFH', 'AA', 'XYZ')  
-- RESULT = XYZXYZABCDEFHXYZXYZBCDEFH
```

This last example shows that matching is from the left. The first four As are matched as two pairs and replaced. The fifth A is not matched.

REPLICATE function

REPLICATE is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER) and returns a string made up of multiple copies of a supplied string.

Syntax

```
►► REPLICATE ( PatternStringExpression , CountNumericExpression ) ◄◄
```

REPLICATE returns a string consisting of the pattern string given by *PatternStringExpression* repeated the number of times given by *CountNumericExpression*.

The pattern string can be of the CHARACTER, BLOB, or BIT datatype and the count must be of type INTEGER. The result is of the same data type as the pattern string.

If the count is negative or zero, a zero length string is returned. If either parameter is NULL, the result is NULL.

The count is limited to 32*1024*1024 to protect the broker from erroneous programs. If this limit is exceeded, an exception condition is issued.

RIGHT function

RIGHT is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and truncates a string.

Syntax

```
►► RIGHT ( SourceStringExpression , LengthIntegerExpression ) ◄◄
```

RIGHT returns a string consisting of the source string truncated to the length given by the length expression. The truncation discards the initial characters of the source string.

The source string can be of the CHARACTER, BLOB, or BIT data type and the length must be of type INTEGER.

If the length is negative or zero, a zero length string is returned. If either parameter is NULL, the result is NULL

RTRIM function

RTRIM is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and removes trailing singletons from a string.

Syntax

```
►► RTRIM ( source_string ) ◄◄
```

RTRIM returns a string value of the same data type and content as *source_string* but with any trailing default singletons removed. The term *singleton* refers to a single part (BIT, BLOB, or CHARACTER) within a string of that type.

The RTRIM function is equivalent to TRIM(TRAILING FROM *source_string*).

If the parameter is NULL, the result is NULL.

The default singleton depends on the data type of *source_string*:

Character	' ' (space)
BLOB	X'00'
Bit	B'0'

SPACE function

SPACE is a string manipulation function that manipulates all data types (BIT, BLOB, and CHARACTER), and creates a string consisting of a defined number of blank spaces.

Syntax

```
▶▶ SPACE (NumericExpression) ▶▶
```

SPACE returns a character string consisting of the number of blank spaces given by *NumericExpression*. The parameter must be of type INTEGER; the result is of type CHARACTER.

If the parameter is negative or zero, a zero length character string is returned. If the parameter is NULL, the result is NULL.

The string is limited to 32*1024*1024 to protect the broker from erroneous programs. If this limit is exceeded, an exception condition is issued.

STARTSWITH function

STARTSWITH is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and returns a Boolean value to indicate whether one string begins with another.

Syntax

```
▶▶ STARTSWITH (SourceExpression, SearchExpression) ▶▶
```

STARTSWITH returns TRUE if *SourceExpression* begins with *SearchExpression*, otherwise it returns FALSE.

The parameter strings for both *SearchExpression* and *SourceExpression* can be of the CHARACTER, BLOB, or BIT data type, but must be of the same data type.

If any parameter is NULL, the result is NULL.

Examples

```
STARTSWITH('Hello World!', 'Hello');
```

returns TRUE.

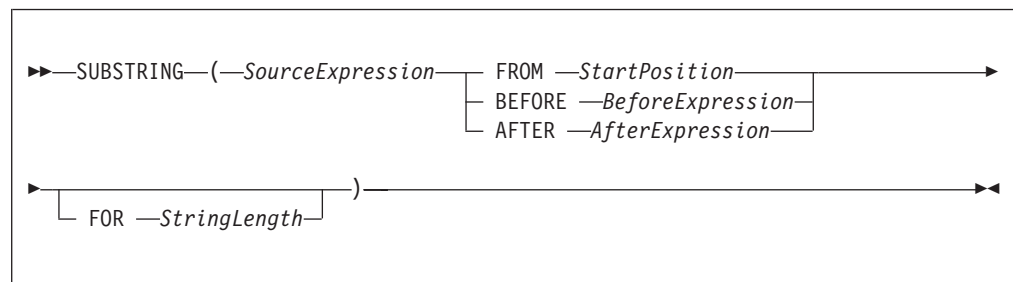
```
STARTSWITH('Hello World!', 'World');
```

returns FALSE.

SUBSTRING function

SUBSTRING is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and extracts characters from a string to create another string.

Syntax



Parameters must be of the following types:

- *SourceExpression*, *BeforeExpression*, and *AfterExpression* can be BIT, BLOB, or CHARACTER.
- *StartPosition* and *StringLength* can be INTEGER only.

StartPosition

If you specify *StartPosition*, SUBSTRING returns a new string of the same type as *SourceExpression* containing one contiguous sequence of characters that are extracted from *SourceExpression*, as specified by *StartPosition* and *StringLength*. If you do not specify *StringLength*, the sequence runs from *StartPosition* until the end of *SourceExpression*. The *StartPosition* can be negative, and together, the *StartPosition* and *StringLength* define a range. The result is the overlap between this range and the *SourceExpression*; the *StringLength* cannot be less than the *StartPosition*.

BeforeExpression

If you specify *BeforeExpression*, SUBSTRING returns a new string of the same type as *SourceExpression* containing one contiguous sequence of characters that are extracted from *StringLength* characters before the first occurrence of *BeforeExpression* within *SourceExpression*, up to (but not including) the first character of the first occurrence of *BeforeExpression*. If you do not specify *StringLength*, the sequence of characters is taken from the beginning of *SourceExpression* up to (but not including) the first character of the first occurrence of *BeforeExpression*. If the *BeforeExpression* string does not occur in *SourceExpression*, an empty (zero length) string is returned.

The *BeforeExpression* string must be of the same data type as *SourceExpression*.

AfterExpression

If you specify *AfterExpression*, SUBSTRING returns a new string of the same type as *SourceExpression*, containing one contiguous sequence of characters that are extracted from *SourceExpression*, beginning with the first character after the end of the first occurrence of *AfterExpression* until the end of *SourceExpression* (or *StringLength* characters, if specified). If the *AfterExpression* string does not occur in *SourceExpression*, an empty (zero length) string is returned.

The *AfterExpression* string must be of the same data type as *SourceExpression*.

If any parameter is NULL, the result is NULL. This is not a zero length string.

Examples:

```
SUBSTRING('Hello World!' FROM 7 FOR 4)
```

returns 'Worl'.

```
SUBSTRING('Hello World!' BEFORE 'World');
```

returns 'Hello '.

```
SUBSTRING('Hello World!' BEFORE 'World' FOR 3);
```

returns 'lo '.

```
SUBSTRING('Hello World!' BEFORE 'e');
```

returns 'H'.

```
SUBSTRING('Hello World!' AFTER 'World');
```

returns '! '.

```
SUBSTRING('Hello World!' AFTER 'W' FOR 2);
```

returns 'or'.

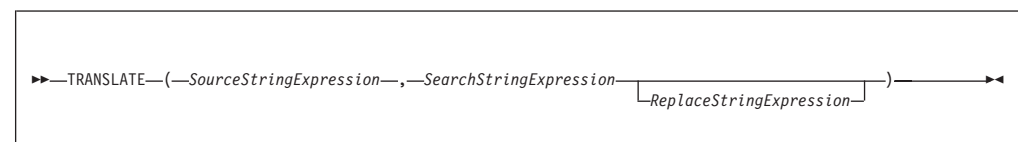
```
SUBSTRING('Hello World!' AFTER 'P');
```

returns ''.

TRANSLATE function

TRANSLATE is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and replaces specified characters in a string.

Syntax



TRANSLATE returns a string consisting of the source string, with each occurrence of any character that occurs in the search string being replaced by the corresponding character from the replace string.

The parameter strings can be of the CHARACTER, BLOB, or BIT data type but all three must be of the same type. If any parameter is NULL, the result is NULL.

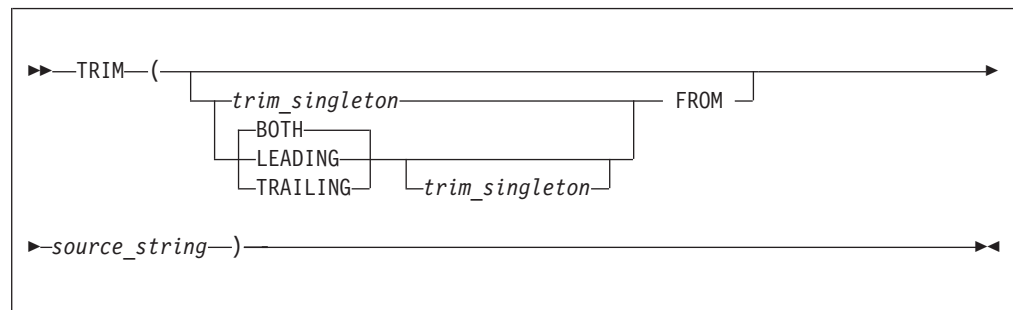
If the replace string is shorter than the search string, there are characters in the search string for which there is no corresponding character in the replace string. This is treated as an instruction to delete these characters and any occurrences of these characters in the source string are absent from the returned string

If the replace string expression is not specified, the replace string is assumed to be an empty string, and the function deletes all occurrences of any characters in the search string from the result.

TRIM function

TRIM is a string manipulation function that manipulates all string data types (BIT, BLOB, and CHARACTER), and removes trailing and leading singletons from a string.

Syntax



TRIM returns a new string of the same type as *source_string*, in which the leading, trailing, or both leading and trailing singletons have been removed. The term *singleton* refers to a single part (BIT, BYTE, or CHARACTER) within a string of that type.

If *trim_singleton* is not specified, a default singleton is assumed. The default singleton depends on the data type of *source_string*:

Character	' ' (space)
BLOB	X'00'
Bit	B'0'

If any parameter is NULL, the result is NULL.

It is often unnecessary to strip trailing blanks from character strings before comparison, because the rules of character string comparison mean that trailing blanks are not significant.

The following examples illustrate the behavior of the TRIM function:

```
TRIM(TRAILING 'b' FROM 'aaabB')
```

returns 'aaabB'.

```
TRIM(' a ')
```

returns 'a'.
TRIM(LEADING FROM ' a ')

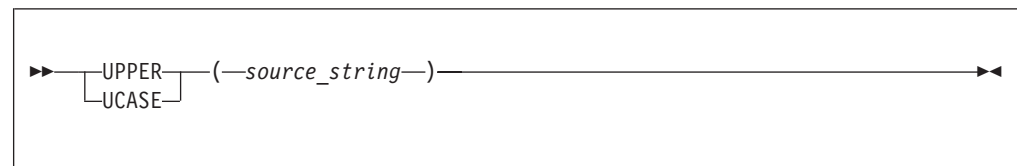
returns 'a '
TRIM('b' FROM 'bbbaaabb')

returns 'aaa'.

UPPER and UCASE functions

UPPER and UCASE are equivalent string manipulation functions that manipulate CHARACTER string data and convert lowercase characters in a string to uppercase.

Syntax



UPPER and UCASE both return a new character string, which is identical to *source_string*, except that all lowercase letters are replaced with the corresponding uppercase letters.

For example:

```
UPPER('ABCD')
```

returns 'ABCD'.

```
UCASE('abc123')
```

returns 'ABC123'.

Converting characters from different code pages to uppercase

If you are using certain code pages, characters with no uppercase equivalent in your code page might be converted when you use the UPPER or UCASE function. This conversion happens because the bitstream is converted to a Unicode message tree by the message parser. Even though characters might have no uppercase equivalent in the source code page, they can still have an uppercase equivalent in the Unicode code page, and are converted by the UPPER or UCASE function. When the bitstream is converted back to the original code page, these characters cannot be converted back, and a substitution character is inserted into the output message for each character. The substitution character inserted depends on the original code page. For example, conversion to an EBCDIC code page inserts an X'3F' byte and conversion to a Japanese code page inserts an X'7F' byte.

A solution to this problem is to use the TRANSLATE function to convert selected characters to uppercase, instead of using the UPPER or UCASE function. Any characters that have no uppercase equivalent in the code page are excluded from the conversion.

In the following example, the input message is in code page 284, and the InputRoot.XML.MSG.APPDATA element contains characters that do not have an uppercase equivalent in code page 284, but do have uppercase equivalents in the

Unicode code page. The TRANSLATE function is used to convert only the lowercase characters 'a' to 'z' to their equivalent uppercase characters. Any other characters in InputRoot.XML.MSG.APPDATA are excluded from the conversion.

```
DECLARE char1 CHAR;  
SET char1 = TRANSLATE(InputRoot.XML.MSG.APPDATA, 'abcdefghijklmnopqrstuvwxyz', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');  
SET OutputRoot.MQMD.CodedCharSetId = 284;  
SET OutputRoot.XML.TEST.translated = char1;
```

ESQL field functions

This topic lists the ESQL field functions and covers the following:

“ASBITSTREAM function”

“BITSTREAM function (deprecated)” on page 319

“FIELDNAME function” on page 320

“FIELDNAMESPACE function” on page 321

“FIELDTYPE function” on page 321

“FIELDVALUE function” on page 323

“FOR function” on page 324

“LASTMOVE function” on page 326

“SAMEFIELD function” on page 326

ASBITSTREAM function

The ASBITSTREAM field function generates a bit stream for the subtree of a given field according to the rules of the parser that owns the field.

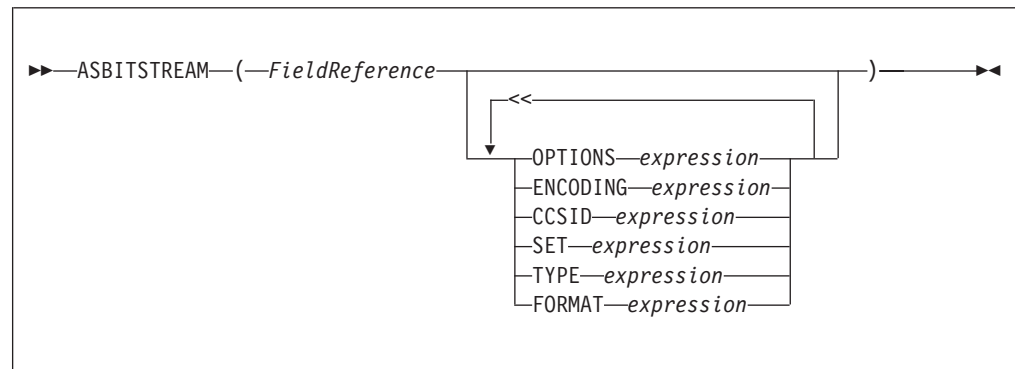
The ASBITSTREAM field function uses parameters supplied by the caller for:

- Encoding
- CCSID
- Message set
- Message type
- Message format
- Options

The ASBITSTREAM function removes the limitation of the existing BITSTREAM function, which can be used only on a tree produced by a parser that belongs to an input node.

The BITSTREAM function is retained only for compatibility with earlier versions.

Syntax



Note that each clause can occur once only.

ASBITSTREAM returns a value of type BLOB that contains a bitstream representation of the field that is pointed to by *FieldReference* and its children.

The algorithm for doing this varies from parser to parser, and according to the options specified. All parsers support the following modes:

- RootBitStream, in which the algorithm that generates the bit stream is the same as the algorithm that is used by an output node. In this mode, a meaningful result is obtained only if the field pointed to is at the head of a subtree with an appropriate structure.
- EmbeddedBitStream, in which not only is the algorithm that generates the bit stream the same as the algorithm that is used by an output node, but also the
 - Encoding
 - CCSID
 - Message set
 - Message type
 - Message format

are determined, if not explicitly specified, in the same way as the output node. That is, they are determined by searching the previous siblings of *FieldReference* on the assumption that they represent headers.

In this way, the algorithm for determining these properties is essentially the same as that used for the BITSTREAM function.

Some parsers also support another mode, FolderBitStream, which generates a meaningful bit stream for any subtree, provided that the field that is pointed to represents a folder.

In all cases, the bit stream obtained can be given to a CREATE statement with a PARSE clause, using the same DOMAIN and OPTIONS to reproduce the original subtree.

When the function is called, any clause expressions are evaluated. An exception is thrown if any of the expressions do not result in a value of the appropriate type.

If any parameter is NULL the result is NULL.

Clause	Type	Default value
OPTIONS	Integer	RootBitStream & ValidateNone
ENCODING	Integer	0
CCSID	Integer	0
SET	Character	Zero length string
TYPE	Character	Zero length string
FORMAT	Character	Zero length string

For details of the syntax of the TYPE clause, refer to Specifying namespaces in the Message Type property.

Although the OPTIONS clause accepts any expression that returns a value of type integer, it is only meaningful to generate option values from the list of supplied constants, using the BITOR function if more than one option is required.

The generated value becomes an integer and can be saved in a variable, passed as a parameter to a function, or used directly in an ASBITSTREAM call. The list of globally-defined constants is:

```

Validate master options...
ValidateContentAndValue
ValidateValue    -- Can be used with ValidateContent
ValidateContent  -- Can be used with ValidateValue
ValidateNone

Validate failure action options...
ValidateException
ValidateExceptionList
ValidateLocalError
ValidateUserTrace

```

Note:

1. For full details of the validation options, refer to Validation properties.

C and Java equivalent APIs

Note that equivalent options are not available on:

- The Java plugin node API MBElement methods createElementAsLastChildFromBitstream() and toBitstream()
- The C plugin node API methods cniCreateElementAsLastChildFromBitstream() and cniElementAsBitstream.

Only one option from each group can be specified, with the exception of ValidateValue and ValidateContent, which can be used together to obtain the content and value validation. If you do not specify an option within a group, the option in bold is used.

The ENCODING clause accepts any expression that returns a value of type integer. However, it is only meaningful to generate encoding values from the list of supplied constants:

```

0
MQENC_INTEGER_NORMAL
MQENC_INTEGER_REVERSED
MQENC_DECIMAL_NORMAL

```

```

MQENC_DECIMAL_REVERSED
MQENC_FLOAT_IEEE_NORMAL
MQENC_FLOAT_IEEE_REVERSED
MQENC_FLOAT_S390

```

0 uses the queue manager's encoding.

The values that are used for the CCSID clause follow the normal numbering system. For example, 1200 = UCS-2, 1208 = UTF-8.

In addition the special values 0 and -1 are supported:

- 0 uses the queue manager's CCSID
- -1 uses the CCSID's as determined by the parser itself. This value is reserved.

For absent clauses, the given default values are used. Use the CCSID and encoding default values, because they take their values from the queue manager's encoding and CCSID settings.

Similarly, use the default values for each of the message set, type, and format options, because many parsers do not require message set, type, or format information; any valid value is sufficient.

When any expressions have been evaluated, the appropriate bit stream is generated.

Note: Because this function has a large number of clauses, an alternative syntax is supported in which the parameters are supplied as a comma-separated list rather than by named clauses. In this case, the expressions must be in the following order:

```
ENCODING -> CCSID -> SET -> TYPE -> FORMAT -> OPTIONS
```

The list can be truncated at any point and you can use an empty expression for any clauses for which you do not supply a value.

Examples

```

DECLARE options INTEGER BITOR(FolderBitStream, ValidateContent,
                             ValidateValue);
SET result = ASBITSTREAM(cursor OPTIONS options CCSID 1208);
SET Result = ASBITSTREAM(Environment.Variables.MQRFH2.Data,,1208
                          ,,,,options);

```

BITSTREAM function (deprecated)

The BITSTREAM field function returns a value that represents the bit stream that is described by the given field and its children. Its use is deprecated; use the newer ASBITSTREAM function instead.

The BITSTREAM function can be used only on a tree produced by a parser belonging to an input node. The ASBITSTREAM function does not suffer from this limitation.

Syntax

```

▶▶ BITSTREAM (—field_reference—) ▶▶

```

The BITSTREAM function returns a value of type BLOB that represents the bit stream that is described by the given field and its children. For incoming messages, the appropriate portion of the incoming bit stream is used. For messages that are constructed by Compute nodes, the following algorithm is used to establish the ENCODING, CCSID, message set, message type, and message format:

- If the addressed field has a previous sibling, and this sibling is the root of a subtree that belongs to a parser capable of providing an ENCODING and CCSID, these values are obtained and used to generate the requested bit stream. Otherwise, the broker's default ENCODING and CCSID (that is, those of its queue manager) are used.
- Similarly, if the addressed field has a previous sibling, and this sibling is the root of a subtree that belongs to a parser capable of providing a message set, message type, and message format, these values are obtained and used to generate the requested bit stream. Otherwise, zero length strings are used.

This function is typically used for message warehouse scenarios, where the bit stream of a message needs to be stored in a database. The function returns the bit stream of the physical portion of the incoming message, identified by the parameter. In some cases, it does not return the bit stream that represents the actual field identified. For example, the following two calls return the same value:

```
BITSTREAM(Root.MQMD);
BITSTREAM(Root.MQMD.UserIdentifier);
```

because they lie in the same portion of the message.

FIELDNAME function

The FIELDNAME field function returns the name of a given field.

Syntax

```
►►—FIELDNAME—(—source_field_reference—)—————►
```

FIELDNAME returns the name of the field identified by *source_field_reference* as a character value. If the parameter identifies a nonexistent field, NULL is returned.

For example:

- FIELDNAME(InputRoot.XMLNS) returns XMLNS.
- FIELDNAME(InputBody) returns the name of the last child of InputRoot, which could be XMLNS.
- FIELDNAME(InputRoot.*[<]) returns the name of the last child of InputRoot, which could be XMLNS.

This function does not show any namespace information; this must be obtained by a separate call to the "FIELDNAMESPACE function" on page 321.

Whereas the following ESQL sets X to "F1":

```
SET X=FIELDNAME(InputBody.*[<]);
```

The following ESQL sets Y to null:

```
SET Y=FIELDNAME(InputBody.F1.*[<]);
```


However, the following ESQL sets Z to the (expected) child of F1:

```
SET Z=FIELDNAME(InputBody.*[<].*[<]);
```

This is because F1 belongs to a namespace and needs to be explicitly referenced by, for example:

```
DECLARE ns NAMESPACE 'urn:nid:xxxxxx';
```

```
SET Y=FIELDNAME(InputBody.ns:F1.*[<]);
```

FIELDNAMESPACE function

The FIELDNAMESPACE field function returns the namespace of a given field.

Syntax

```
►►—FIELDNAMESPACE—(—FieldReference—)—————►►
```

FIELDNAMESPACE takes a field reference as a parameter and returns a value of type CHARACTER containing the namespace of the addressed field. If the parameter identifies a nonexistent field, NULL is returned.

FIELDTYPE function

The FIELDTYPE field function returns the type of a given field.

Syntax

```
►►—FIELDTYPE—(—source_field_reference—)—————►►
```

FIELDTYPE returns an integer representing the type of the field identified by *source_field_reference*; this is the type of the field, not the data type of the field that the parameter identifies. If the parameter identifies a nonexistent entity, NULL is returned.

The mapping of integer values to field types is not published, and might change from release to release. Compare the results of the FIELDTYPE function with named field types.

For example:

```
IF FIELDTYPE(source_field_reference) = NameValue  
THEN ...
```

The named field types that you can use in this context are listed below.

The following conditions apply:

- *Name*, *Value*, *NameValue* and *MQRFH2.BitStream* are domain independent.
- The *XML.** types are applicable to the XML, XMLNS, JMSMap, and JMSStream domains, except for *XML.Namespace* which is specific to the XMLNS domain.
- The *XMLNSC.** types are applicable to the XMLNSC domain.

You must use these types with the capitalization shown:

- *Name*
- *Value*
- *NameValue*
- *MQRFH2.BitStream*
- *MQRFH2.Field*
- *MQRFH2C.Field*
- *XML.AsisElementContent*
- *XML.Attribute*
- *XML.AttributeDef*
- *XML.AttributeDefDefaultType*
- *XML.AttributeDefType*
- *XML.AttributeDefValue*
- *XML.AttributeList*
- *XML.BitStream*
- *XML.CDataSection*
- *XML.Comment*
- *XML.Content*
- *XML.DocTypeComment*
- *XML.DocTypeDecl*
- *XML.DocTypePI*
- *XML.DocTypeWhiteSpace*
- *XML.Element*
- *XML.ElementDef*
- *XML.Encoding*
- *XML.EntityDecl*
- *XML.EntityDeclValue*
- *XML.EntityReferenceStart*
- *XML.EntityReferenceEnd*
- *XML.ExternalEntityDecl*
- *XML.ExternalParameterEntityDecl*
- *XML.ExtSubset*
- *XML.IntSubset*
- *XML.NamespaceDecl*
- *XML.NotationDecl*
- *XML.NotationReference*
- *XML.ParameterEntityDecl*
- *XML.ParserRoot*
- *XML.ProcessingInstruction*
- *XML.PublicId*
- *XML.RequestedDomain*
- *XML.Standalone*
- *XML.SystemId*
- *XML.UnparsedEntityDecl*
- *XML.Version*

- *XML.WhiteSpace*
- *XML.XmlDecl*
- *XMLNSC.Attribute*
- *XMLNSC.BitStream*
- *XMLNSC.CDataField*
- *XMLNSC.CDataValue*
- *XMLNSC.Comment*
- *XMLNSC.DocumentType*
- *XMLNSC.DoubleAttribute*
- *XMLNSC.DoubleEntityDefinition*
- *XMLNSC.EntityDefinition*
- *XMLNSC.EntityReference*
- *XMLNSC.Field*
- *XMLNSC.Folder*
- *XMLNSC.HybridField*
- *XMLNSC.HybridValue*
- *XMLNSC.PCDataField*
- *XMLNSC.PCDataValue*
- *XMLNSC.ProcessingInstruction*
- *XMLNSC.SingleAttribute*
- *XMLNSC.SingleEntityDefinition*
- *XMLNSC.Value*
- *XMLNSC.XmlDeclaration*

You can also use this function to determine whether a field in a message exists. To do this, use the form:

```
FIELDTYPE(SomeFieldReference) IS NULL
```

If the field exists, an integer value is returned to the function that indicates the field type (for example, string). When this is compared to NULL, the result is FALSE. If the field does not exist, NULL is returned and therefore the result is TRUE. For example:

```
IF FIELDTYPE(InputRoot.XMLNS.Message1.Name)
  IS NULL THEN
  // Name field does not exist, take error
  action....
  ... more ESQL ...
ELSE
  // Name field does exist, continue....
  ... more ESQL ...
END IF
```

FIELDVALUE function

The FIELDVALUE field function returns the scalar value of a given field.

Syntax

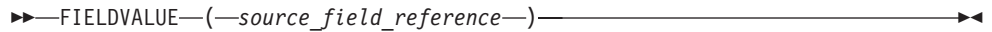


Diagram showing the syntax for the FIELDVALUE function: `FIELDVALUE(source_field_reference)`. The function name is followed by a single argument in parentheses, which is the source field reference.

FIELDVALUE returns the scalar value of the field identified by *source_field_reference*. If it identifies a non-existent field, NULL is returned.

For example, consider the following XML input message:

```
<Data>
  <Qty Unit="Gallons">1234</Qty>
</Data>
```

The ESQL statement

```
SET OutputRoot.XML.Data.Quantity =
  FIELDVALUE(InputRoot.XML.Data.Qty);
```

gives the result:

```
<Data><Quantity>1234</Quantity></Data>
```

whereas this ESQL statement (without the FIELDVALUE function):

```
SET OutputRoot.XML.Data.Quantity =
  InputRoot.XML.Data.Qty;
```

causes a tree copy, with the result:

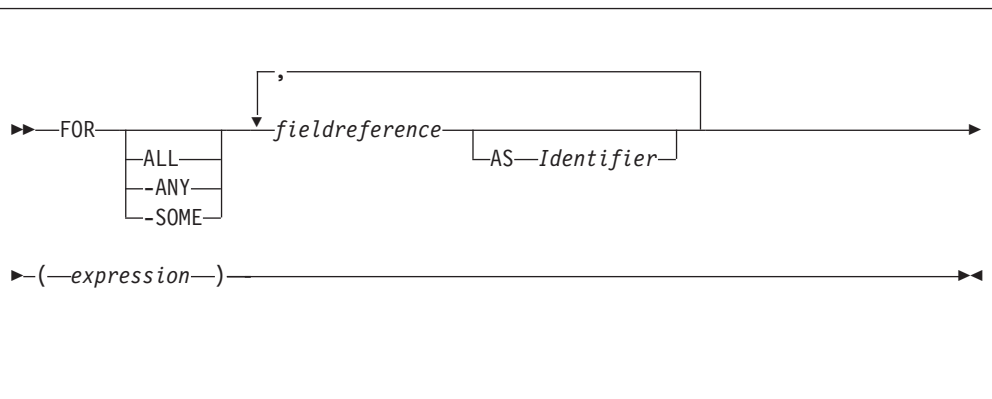
```
<Data><Quantity Unit="Gallons">1234</Quantity></Data>
```

because the field Qty is not a leaf field.

FOR function

The FOR field function evaluates an expression and assigns a resulting value of TRUE, FALSE, or UNKNOWN

Syntax



FOR enables you to write an expression that iterates over all instances of a repeating field. For each instance it processes a boolean expression and collates the results.

For example:

```
FOR ALL Body.Invoice.Purchases."Item"[] AS I (I.Quantity <= 50)
```

Note:

1. With the quantified predicate , the first thing to note is the [] on the end of the field reference after the FOR ALL. The square brackets define iteration over all instances of the Item field.

In some cases, this syntax appears unnecessary, because you can get that information from the context, but it is done for consistency with other pieces of syntax.

- 2.

The ASclause associates the name I in the field reference with the current instance of the repeating field. This is similar to the concept of iterator classes used in some object oriented languages such as C++. The expression in parentheses is a predicate that is evaluated for each instance of the Item field.

If you specify the **ALL** keyword, the function iterates over all instances of the field Item inside Body.Invoice.Purchases and evaluates the predicate I.Quantity <= 50. If the predicate evaluates to:

- TRUE (if the field is empty, or for all instances of Item) return TRUE.
- FALSE (for any instance of Item) return FALSE.
- Anything else, return UNKNOWN.

The **ANY** and **SOME** keywords are equivalent. If you use either, the function iterates over all instances of the field Item inside Body.Invoice.Purchases and evaluates the predicate I.Quantity <= 50. If the predicate evaluates to:

- FALSE (if the field is empty, or for all instances of Item) return FALSE.
- TRUE (for any instance of Item) return TRUE.
- Anything else, return UNKNOWN.

To further illustrate this, the following examples are based on the message described in "Example message" on page 385. In the following filter expression:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Title = 'The XML Companion')
```

the sub-predicate evaluates to TRUE. However, this next expression returns FALSE:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Title = 'C Primer')
```

because the C Primer is not included on this invoice. If in this instance some of the items in the invoice do not include a book title field, the sub-predicate returns UNKNOWN, and the quantified predicate returns the value UNKNOWN.

Take great care to deal with the possibility of null values appearing. Write this filter with an explicit check on the existence of the field, as follows:

```
FOR ANY Body.Invoice.Purchases."Item"[] AS I (I.Book IS NOT NULL AND I.Book.Title = 'C Primer')
```

The IS NOT NULL predicate ensures that, if an Item field does not contain a Book, a FALSE value is returned from the sub-predicate.

LASTMOVE function

The LASTMOVE field function tells you whether the last MOVE function succeeded.

Syntax

```
▶▶—LASTMOVE—(—source_dynamic_reference—)————▶▶
```

LASTMOVE returns a Boolean value indicating whether the last MOVE function applied to *source_dynamic_reference* was successful (TRUE) or not (FALSE).

See “MOVE statement” on page 257 for an example of using the MOVE statement, and the LASTMOVE function to check its success.

See “Creating dynamic field references” on page 36 for information about dynamic references.

SAMEFIELD function

The SAMEFIELD field function tells you whether two field references point to the same target.

Syntax

```
▶▶—SAMEFIELD—(—source_field_reference1—,—source_field_reference2—)————▶▶
```

SAMEFIELD returns a BOOLEAN value indicating whether two field references point to the same target. If they do, SAMEFIELD returns TRUE; otherwise SAMEFIELD returns FALSE.

For example:

```
DECLARE ref1 REFERENCE TO OutputRoot.XMLNS.Invoice.Purchases.Item[1];  
MOVE ref1 NEXTSIBLING;  
SET Result = SAMEFIELD(ref1,OutputRoot.XMLNS.Invoice.Purchases.Item[2]);
```

Result is TRUE.

See “Creating dynamic field references” on page 36 for information about dynamic references.

ESQL list functions

ESQL provides several functions to work with lists.

“CARDINALITY function” on page 327

“EXISTS function” on page 327

“SINGULAR function” on page 328

“THE function” on page 329

CARDINALITY function

The CARDINALITY function returns the number of elements in a list.

Syntax

```
►►—CARDINALITY—(—ListExpression—)—————►◄
```

CARDINALITY returns an integer value giving the number of elements in the list specified by *ListExpression*.

ListExpression is any expression that returns a list. All the following, for example, return a list:

- A LIST constructor
- A field reference with the [] array indicator
- Some SELECT expressions (not all return a list)

A common use of this function is to determine the number of fields in a list before iterating over them.

Examples

```
-- Determine the number of F1 fields in the message.
-- Note that the [ ] are required
DECLARE CountF1 INT CARDINALITY(OutputRoot.XMLNS.Data.Source.F1[]);

-- Determine the number of fields called F1 with the value 'F12' in the message.
-- Again note that the [ ] are required
DECLARE CountF1F12 INT
  CARDINALITY(SELECT F.* FROM OutputRoot.XMLNS.Data.Source.F1[] AS F
              where F = 'F12');

-- Use the value returned by CARDINALITY to refer to a specific element
-- in a list or array:
-- Array indices start at 1, so this example refers to the third-from-last
-- instance of the Item field
Body.Invoice.Item[CARDINALITY(Body.Invoice.Item[]) - 2].Quantity
```

EXISTS function

The EXISTS function returns a Boolean value to indicate whether a list contains at least one element (that is, whether the list exists).

Syntax

```
►►—EXISTS—(—ListExpression—)—————►◄
```

If the list specified by *ListExpression* contains one or more elements, EXISTS returns TRUE. If the list contains no elements, EXISTS returns FALSE.

ListExpression is any expression that returns a list. All the following expressions, for example, return a list:

- A LIST constructor
- A field reference with the [] array indicator
- Some SELECT expressions (not all return a list)

If you want to know only whether a list contains at least one elements or none, EXISTS executes more quickly than an expression involving the CARDINALITY function (for example, `CARDINALITY(ListExpression) <> 0`).

A typical use of this function is to determine whether a field exists.

Examples

```
-- Determine whether the F1 array exists in the message. Note that the [ ]
-- are required
DECLARE Field1Exists BOOLEAN EXISTS(OutputRoot.XMLNS.Data.Source.F1[]);
-- Determine whether the F1 array contains an element with the value 'F12'.
-- Again note that the [ ] are required
DECLARE Field1F12Exists BOOLEAN
  EXISTS(SELECT F.* FROM OutputRoot.XMLNS.Data.Source.F1[] AS F where F = 'F12');
```

SINGULAR function

The SINGULAR function returns a Boolean value that indicates whether a list contains exactly one element.

Syntax

► SINGULAR(—*ListExpression*—) ◄

If the list specified by *ListExpression* contains exactly one element, SINGULAR returns TRUE. If the list contains more or fewer elements, SINGULAR returns FALSE.

ListExpression is an expression that returns a list. All the following expressions, for example, return a list:

- A LIST constructor
- A field reference with the [] array indicator
- Some SELECT expressions (not all return a list)

If you want to know only whether a list contains just one element or some other number, SINGULAR executes more quickly than an expression involving the CARDINALITY function (for example, `CARDINALITY(ListExpression) = 1`).

A typical use of this function is to determine whether a field is unique.

Examples

```
-- Determine whether there is just one F1 field in the message.
-- Note that the [ ] are required
DECLARE Field1Unique BOOLEAN SINGULAR(OutputRoot.XMLNS.Data.Source.F1[]);
-- Determine whether there is just one field called F1 with the value 'F12'
-- in the message. Again note that the [ ] are required
DECLARE Field1F12Unique BOOLEAN
  SINGULAR(SELECT F.* FROM OutputRoot.XMLNS.Data.Source.F1[] AS F where F = 'F12');
```


THE function

The THE function returns the first element of a list.

Syntax

The diagram shows the syntax for the THE function. It consists of the word 'THE' followed by an opening parenthesis, the text '*ListExpression*', and a closing parenthesis. A double-headed arrow is positioned above the opening parenthesis, and a single-headed arrow points to the right from the closing parenthesis, indicating the scope of the list expression.

If *ListExpression* contains one or more elements; THE returns the first element of the list. In all other cases, it returns an empty list.

Restrictions

ListExpression must be a SELECT expression.

Complex ESQL functions

This topic lists the complex ESQL functions and covers the following:

“CASE function”

“CAST function” on page 331

“SELECT function” on page 345

“ROW constructor function” on page 352

“LIST constructor function” on page 353

“ROW and LIST combined” on page 354

“ROW and LIST comparisons” on page 354

“Supported casts” on page 357

“Implicit casts” on page 365

“Implicit CASTs for comparisons” on page 365

“Implicit CASTs for arithmetic operations” on page 367

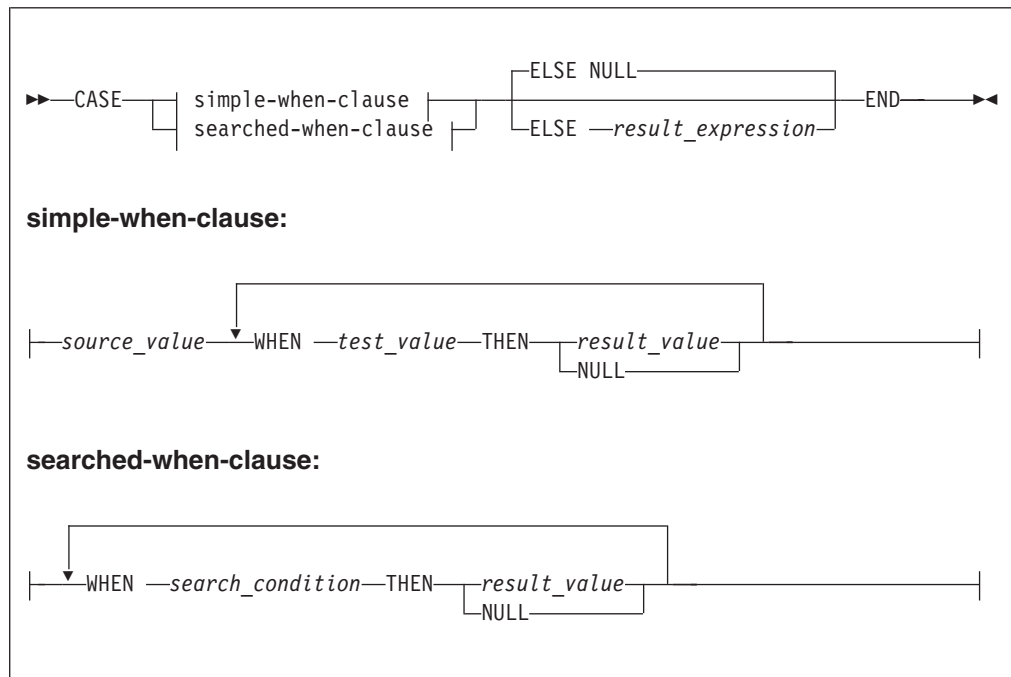
“Implicit CASTs for assignment” on page 369

“Data types of values from external sources” on page 370

CASE function

CASE is a complex function that has two forms; the simple-when form and the searched-when form. In either form CASE returns a result, the value of which controls the path of subsequent processing.

Syntax



In the simple-when form, *source_value* is compared with each *test_value* until a match is found. The result of the CASE function is the value of the corresponding *result_value*. The data type of *source_value* must therefore be comparable to the data type of each *test_value*.

The CASE function must have at least one WHEN clause. The ELSE expression is optional. The default ELSE expression is NULL. A CASE expression is delimited by END. The test values do not have to be literal values.

The searched-when form is similar, but has the additional flexibility of allowing a number of different values to be tested.

The following example shows a CASE function with a simple WHEN clause. In this example, the CASE can be determined only by one variable that is specified next to the CASE keyword.

```
DECLARE CurrentMonth CHAR;
DECLARE MonthText CHAR;
SET CurrentMonth = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2);

SET MonthText =
CASE CurrentMonth
  WHEN '01' THEN 'January'
  WHEN '02' THEN 'February'
  WHEN '03' THEN 'March'
  WHEN '04' THEN 'April'
  WHEN '05' THEN 'May'
  WHEN '06' THEN 'June'
  ELSE 'Second half of year'
END;
```

The following example shows a CASE function with a searched-when-clause. This example is still determined by one variable *CurrentMonth*:

```

DECLARE CurrentMonth CHAR;
DECLARE MonthText CHAR;
SET CurrentMonth = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2);

SET MonthText =
CASE
  WHEN Month = '01' THEN 'January'
  WHEN Month = '02' THEN 'February'
  WHEN Month = '03' THEN 'March'
  WHEN Month = '04' THEN 'April'
  WHEN Month = '05' THEN 'May'
  WHEN Month = '06' THEN 'June'
  ELSE 'Second half of year'
END;

```

In a searched-when-clause, different variables can be used in the WHEN clauses to determine the result. This is demonstrated in the following example of the searched-when-clause:

```

DECLARE CurrentMonth CHAR;
DECLARE CurrentYear CHAR;
DECLARE MonthText CHAR;
SET CurrentMonth = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 6 FOR 2);
SET CurrentYear = SUBSTRING(InputBody.Invoice.InvoiceDate FROM 1 FOR 4);

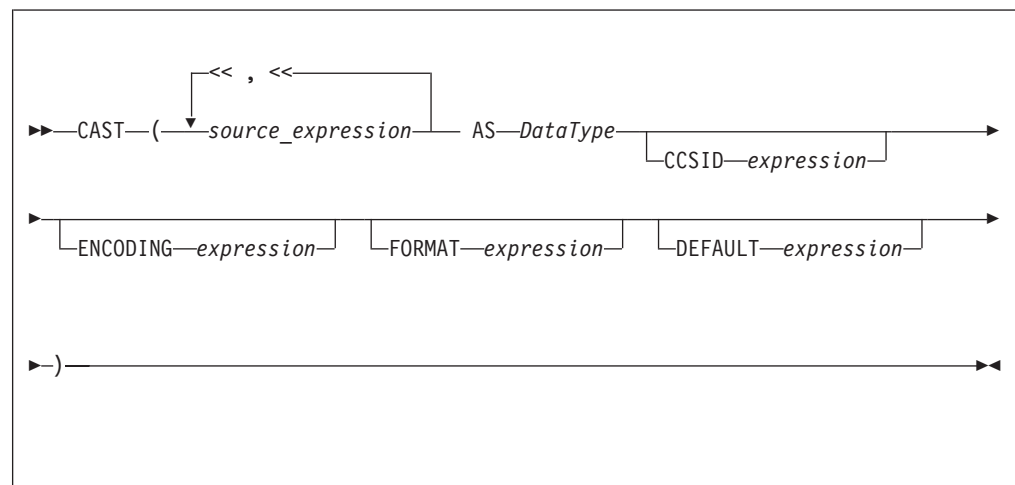
SET MonthText =
CASE
  WHEN CurrentMonth = '01' THEN 'January'
  WHEN CurrentMonth = '02' THEN 'February'
  WHEN CurrentMonth = '03' THEN 'March'
  WHEN CurrentYear = '2000' THEN 'A month in the Year 2000'
  WHEN CurrentYear = '2001' THEN 'A month in the Year 2001'
  ELSE 'Not first three months of any year or a month in the Year 2000 or 2001'
END;

```

CAST function

CAST is a complex function that transforms one or more values from one data type into another.

Syntax



In practice, you cannot specify all of the above parameters at the same time. For example, **CCSID** and **ENCODING** parameters take effect only on string-to-string

conversions, while **FORMAT** applies only to string-numeric and string-datetime conversions (in either direction).

The **CAST** function transforms one or more values from one data type into another data type. For example, you can use **CAST** to process generic XML messages. All fields in an XML message have character values, so to perform an arithmetic calculation or a date/time comparison on a field, for example, use **CAST** to convert the string value of the field into a value of the appropriate type.

Not all conversions are supported; see “Supported casts” on page 357 for a list of supported conversions.

Parameters:

Source expression

CAST returns its first parameter (**source_expression**), which can contain more than one value, as the data type that is specified by its second parameter (**DataType**). In all cases, if the source expression is **NULL**, the result is **NULL**. If the evaluated source expression is not compatible with the target data type, or if the source expression is of the wrong format, a runtime error is generated.

CCSID

The **CCSID** parameter is used only for conversions to or from one of the string data types. Use the **CCSID** parameter to specify the code page of the source or target string.

The **CCSID** parameter can be any expression that evaluates to a value of type **INT**. The expression is interpreted according to normal WebSphere Message Broker rules for **CCSIDs**. See Supported code pages for a list of valid values.

DataType

The **DataType** parameter is the data type into which the source value is transformed. The possible values are:

- String types:
 - BIT
 - BLOB
 - CHARACTER
- Numeric types:
 - DECIMAL
 - FLOAT
 - INTEGER
- Date/Time types:
 - DATE
 - GMTTIME
 - GMTTIMESTAMP
 - INTERVAL
 - TIME
 - TIMESTAMP
- Boolean:
 - BOOLEAN

Ensure that you specify a valid **ESQL** interval subtype after a Date/Time type of **INTERVAL**. For valid **ESQL** interval subtypes, see “**ESQL INTERVAL** data type” on page 162. For example commands that show how to specify a valid **ESQL** interval

subtype, see examples 12, 13, and 14 below.

DEFAULT

The **DEFAULT** parameter provides a method of avoiding exceptions being thrown from **CAST** statements by providing a last-resort value to return.

The **DEFAULT** parameter must be a valid **ESQL** expression that returns the same data type as that specified on the **DataType** parameter, otherwise an exception is thrown.

The **CCSID**, **ENCODING**, and **FORMAT** parameters are not applied to the result of the **DEFAULT** parameter; the expression must, therefore, be of the correct **CCSID**, **ENCODING**, and **FORMAT**.

ENCODING

Use the **ENCODING** parameter to specify the encoding for certain conversions. The **ENCODING** value can be any expression that evaluates to a value of type **INT**, and is interpreted according to normal WebSphere Message Broker rules for encoding. Valid values are:

- **MQENC_NATIVE** (0x00000222L)
- **MQENC_INTEGER_NORMAL** (0x00000001L)
- **MQENC_INTEGER_REVERSED** (0x00000002L)
- **MQENC_DECIMAL_NORMAL** (0x00000010L)
- **MQENC_DECIMAL_REVERSED** (0x00000020L)
- **MQENC_FLOAT_IEEE_NORMAL** (0x00000100L)
- **MQENC_FLOAT_IEEE_REVERSED** (0x00000200L)
- **MQENC_FLOAT_S390** (0x00000300L)

FORMAT

Use the **FORMAT** parameter for conversions between string data types and numerical or date/time data types. For conversions *from* string types, **FORMAT** defines how the source string should be parsed to fill the target data type. For conversions *to* string types, it defines how the data in the source expression is formatted in the target string.

FORMAT takes different types of expression for date/time and numerical conversions. However, the same **FORMAT** expression can be used irrespective of whether the conversion is to a string or from a string.

You can specify a **FORMAT** parameter when casting:

- From any of the string data types (**BIT**, **BLOB**, or **CHARACTER**) to:
 - **DECIMAL**
 - **FLOAT**
 - **INTEGER**
 - **DATE**
 - **GMTTIMESTAMP**
 - **TIMESTAMP**
 - **GMTTIME**
 - **TIME**

- To any of the string data types (BIT, BLOB, or CHARACTER) from any of the numerical and date/time data types in the previous list.

Specifying **FORMAT** for an unsupported combination of source and target data types causes error message BIP3205 to be issued.

For more information about conversion to and from numerical data types, see “Formatting and parsing numbers as strings” on page 336. For more information about conversion to and from date/time data types, see “Formatting and parsing dateTimes as strings” on page 339.

The **FORMAT** parameter is equivalent to those used in many other products, such as ICU and Microsoft Excel.

Examples:

Example 1. Formatted CAST from DECIMAL to CHARACTER

```
DECLARE source DECIMAL 31415.92653589;
DECLARE target CHARACTER;
DECLARE pattern CHARACTER '#,##0.00';
SET target = CAST(source AS CHARACTER FORMAT pattern);
-- target is now '31,415.93'
```

Example 2. Formatted CAST from DATE to CHARACTER

```
DECLARE now CHARACTER;
SET now = CAST(CURRENT_TIMESTAMP AS CHARACTER FORMAT 'yyyyMMdd-HHmss');
-- target is now '20041007-111656' (in this instance at least)
```

Example 3. Formatted CAST from CHARACTER to DATE

```
DECLARE source CHARACTER '01-02-03';
DECLARE target DATE;
DECLARE pattern CHARACTER 'dd-MM-yy';
SET target = CAST(source AS DATE FORMAT pattern);
-- target now contains Year=2003, Month=02, Day=01
```

Example 4. Formatted CAST from CHARACTER to TIMESTAMP

```
DECLARE source CHARACTER '12 Jan 03, 3:45pm';
DECLARE target TIMESTAMP;
DECLARE pattern CHARACTER 'dd MMM yy, h:mma';
SET target = CAST(source AS TIMESTAMP FORMAT pattern);
-- target now contains Year=2003, Month=01, Day=03, Hour=15, Minute=45,
--                               Seconds=58
-- (seconds taken from CURRENT_TIME since not present in input)
```

Example 5. Formatted CAST from DECIMAL to CHARACTER, with negative pattern

```
DECLARE source DECIMAL -54231.122;
DECLARE target CHARACTER;
DECLARE pattern CHARACTER '#,##0.00;(#,##0.00)';
SET target = CAST(source AS CHARACTER FORMAT pattern);
-- target is now '£(54,231.12)'
```

Example 6. Formatted CAST from CHARACTER to TIME

```
DECLARE source CHARACTER '16:18:30';
DECLARE target TIME;
DECLARE pattern CHARACTER 'hh:mm:ss';
SET target = CAST(source AS TIME FORMAT pattern);
-- target now contains Hour=16, Minute=18, Seconds=30
```

Example 7. CASTs from the numeric types to DATE

```
CAST(7, 6, 5 AS DATE);
CAST(7.4e0, 6.5e0, 5.6e0 AS DATE);
CAST(7.6, 6.51, 5.4 AS DATE);
```

Example 8. CASTs from the numeric types to TIME

```
CAST(9, 8, 7 AS TIME);
CAST(9.4e0, 8.6e0, 7.1234567e0 AS TIME);
CAST(9.6, 8.4, 7.7654321 AS TIME);
```

Example 9. CASTs to TIMESTAMP

```
CAST(DATE '0001-02-03', TIME '04:05:06' AS TIMESTAMP);
CAST(2, 3, 4, 5, 6, 7.8 AS TIMESTAMP);
```

Example 10. CASTs to GMTTIMESTAMP

```
CAST(DATE '0002-03-04', GMTTIME '05:06:07' AS GMTTIMESTAMP);
CAST(3, 4, 5, 6, 7, 8 AS GMTTIMESTAMP);
CAST(3.1e0, 4.2e0, 5.3e0, 6.4e0, 7.5e0, 8.6789012e0 AS GMTTIMESTAMP);
CAST(3.2, 4.3, 5.4, 6.5, 7.6, 8.7890135 AS GMTTIMESTAMP);
```

Example 11. CASTs to INTERVAL from INTEGER

```
CAST(1234 AS INTERVAL YEAR);
CAST(32, 10 AS INTERVAL YEAR TO MONTH);
CAST(33, 11 AS INTERVAL DAY TO HOUR);
CAST(34, 12 AS INTERVAL HOUR TO MINUTE);
CAST(35, 13 AS INTERVAL MINUTE TO SECOND);
CAST(36, 14, 10 AS INTERVAL DAY TO MINUTE);
CAST(37, 15, 11 AS INTERVAL HOUR TO SECOND);
CAST(38, 16, 12, 10 AS INTERVAL DAY TO SECOND);
```

Example 12. CASTs to INTERVAL from FLOAT

```
CAST(2345.67e0 AS INTERVAL YEAR);
CAST(3456.78e1 AS INTERVAL MONTH);
CAST(4567.89e2 AS INTERVAL DAY);
CAST(5678.90e3 AS INTERVAL HOUR);
CAST(6789.01e4 AS INTERVAL MINUTE);
CAST(7890.12e5 AS INTERVAL SECOND);
CAST(7890.1234e0 AS INTERVAL SECOND);
```

Example 13. CASTs to INTERVAL from DECIMAL

```
CAST(2345.67 AS INTERVAL YEAR);
CAST(34567.8 AS INTERVAL MONTH);
CAST(456789 AS INTERVAL DAY);
CAST(5678900 AS INTERVAL HOUR);
CAST(67890100 AS INTERVAL MINUTE);
CAST(789012000 AS INTERVAL SECOND);
CAST(7890.1234 AS INTERVAL SECOND);
```

Example 14. CASTs to FLOAT from INTERVAL

```
CAST(INTERVAL '1234' YEAR AS FLOAT);
CAST(INTERVAL '2345' MONTH AS FLOAT);
CAST(INTERVAL '3456' DAY AS FLOAT);
CAST(INTERVAL '4567' HOUR AS FLOAT);
CAST(INTERVAL '5678' MINUTE AS FLOAT);
CAST(INTERVAL '6789.01' SECOND AS FLOAT);
```

Example 15. CASTs DECIMAL from INTERVAL

```
CAST(INTERVAL '1234' YEAR AS DECIMAL);
CAST(INTERVAL '2345' MONTH AS DECIMAL);
CAST(INTERVAL '3456' DAY AS DECIMAL);
```

```
CAST(INTERVAL '4567' HOUR AS DECIMAL);
CAST(INTERVAL '5678' MINUTE AS DECIMAL);
CAST(INTERVAL '6789.01' SECOND AS DECIMAL);
```

Example 16. A ternary cast that fails and results in the substitution of a default value

```
CAST(7, 6, 32 AS DATE DEFAULT DATE '1947-10-24');
```

Example 17. A sexternary cast that fails and results in the substitution of a default value

```
CAST(2, 3, 4, 24, 6, 7.8 AS TIMESTAMP DEFAULT TIMESTAMP '1947-10-24 07:08:09');
```

Example 18. A ternary cast that fails and throws an exception

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE LIKE '%' BEGIN
    SET OutputRoot.XMLNS.Data.Date.FromIntegersInvalidCast = 'Exception thrown';
  END;

  DECLARE Dummy CHARACTER CAST(7, 6, 32 AS DATE);
END;
```

Example 19. A sexternary cast that fails and throws an exception

```
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE LIKE '%' BEGIN
    SET OutputRoot.XMLNS.Data.Timestamp.FromIntegersInvalidCast = 'Exception thrown';
  END;

  DECLARE Dummy CHARACTER CAST(2, 3, 4, 24, 6, 7.8 AS TIMESTAMP);
END;
```

Formatting and parsing numbers as strings:

For conversions between string data types and numeric data types, you can supply, on the `FORMAT` parameter of the `CAST` function, an optional formatting expression.

For conversions *from* string types, the formatting expression defines how the source string should be parsed to fill the target data type.

For conversions *to* string types, the formatting expression defines how the data in the source expression is to be formatted in the target string.

You can specify a `FORMAT` expression for the following numeric conversions (Specifying a `FORMAT` expression for date/time conversions is described in “Formatting and parsing dateTimes as strings” on page 339).

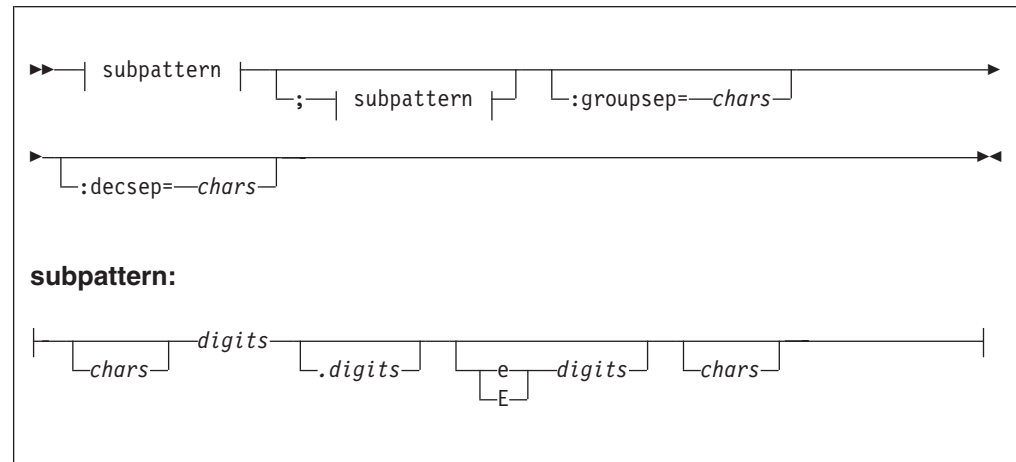
- From any of the string data types (`BIT`, `BLOB`, or `CHARACTER`) to:
 - `DECIMAL`
 - `FLOAT`
 - `INTEGER`
- To any of the string data types (`BIT`, `BLOB`, or `CHARACTER`) from any of the numeric data types that are in the previous list.

The formatting expression consists of three parts:

1. A subpattern that defines positive numbers.
2. An optional subpattern that defines negative numbers. (If only one subpattern is defined, negative numbers use the positive pattern, prefixed with a minus sign.)

3. The optional parameters `groupsep` and `decsep`.

Syntax



Parameters:

chars

A sequence of zero or more characters. All characters can be used, *except* the special characters that are listed under “subpattern” on page 338.

decsep

One or more characters to be used as the separator between the whole and decimal parts of a number (the decimal separator). The default decimal separator is a period (.).

digits

A sequence of one or more of the numeric tokens (0 # - + , .) that are listed under “subpattern” on page 338.

groupsep

One or more characters to be used as the separator between clusters of integers, to make large numbers more readable (the grouping separator). The default grouping separator is nothing (that is, there is no grouping of digits or separation of groups).

Grouping is commonly done in thousands, but it can be redefined by either the pattern or the locale. There are two grouping sizes:

The primary grouping size

Used for the least significant integer digits.

The secondary grouping size

Used for all other integer digits.

In most cases, the primary and secondary grouping sizes are the same, but they can be different. For example, if the pattern used is `#,##,##0`, the primary grouping size is 3 and the secondary is 2. The number 123456789 would become the string “12,34,56,789”.

If multiple grouping separators are used (as in the previous example), the rightmost separator defines the primary size, and the penultimate rightmost separator defines the secondary size.

subpattern

The subpattern consists of:

1. An optional prefix (*chars*)
2. A mandatory pattern representing a whole number
3. An optional pattern representing decimal places
4. An optional pattern representing an exponent (the power by which the preceding number is raised)
5. An optional suffix (*chars*)

Parts 2, 3, and 4 of the subpattern are defined by the tokens in the following table.

Token	Represents
0	Any digit, including a leading zero.
#	Any digit, excluding a leading zero. (See the explanation of the difference between 0 and # that follows this table.)
.	Decimal separator.
+	Prefix of positive numbers.
-	Prefix of negative numbers.
,	Grouping separator.
E/e	Separates the number from the exponent.
;	Subpattern boundary.
'	Quotation mark, applied to special characters. If a quotation mark is needed in output, it must be doubled (").
*	Padding specifier. The character following the asterisk is used to pad the number to fit the length of the format pattern.

The # and 0 characters are used for digit substitution, the difference between them being that a # character is removed if there is no number to replace it with. For example, 10 formatted by the pattern #,##0.00 gives "10.00", but formatted by 0,000.00 gives "0,010.00".

To specify padding characters, use an asterisk. When an asterisk is placed in either of the two *chars* regions (the prefix and suffix), the character immediately following it is used to pad the output. Padding can be specified only once. For example, a pattern of *x#,###,##0.00 applied to 1234 gives "xxx1,234.00". But applied to 1234567, it gives "1,234,567.00".

Examples of formatting patterns:

The following table shows formatting patterns and the strings that are generated from sample numeric input.

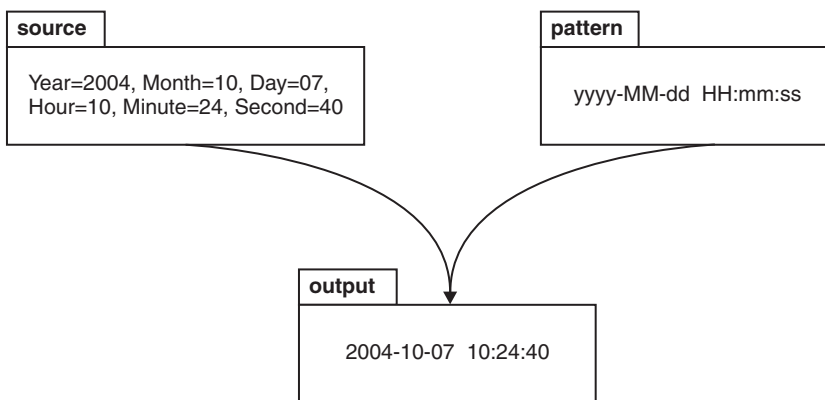
Pattern	Input number	Output string
+++ ,##0.00;- ###,###,##0.00:groupsep=' ':decsep=,	123456789.123	"+123'456'789,12"
##0.00	1000000	"1000000.00"
##0.00	3.14159265	"3.14"

Formatting and parsing dateTimes as strings:

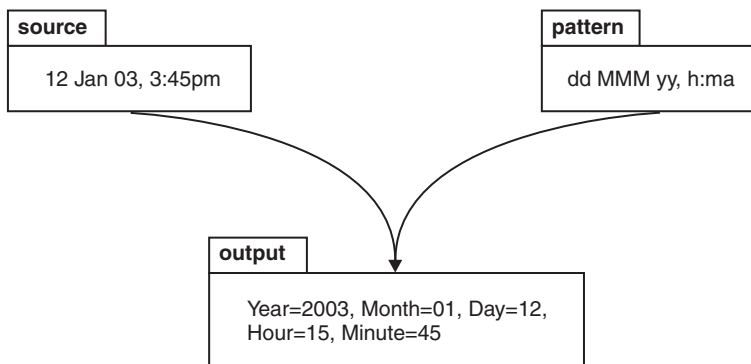
This section gives information on how you can specify the date`Time` format using a string of pattern letters.

When you convert a date or time into a string, a format pattern must be applied that directs the conversion. Apply the format pattern to convert a date or time into a string, or to parse a string into a date or time.

During the conversion (for example, of a date`Time` into a string), a pattern or a set of tokens is replaced with the equivalent source. The following diagram shows how a pattern is used to format a date`Time` source to produce a character string output.

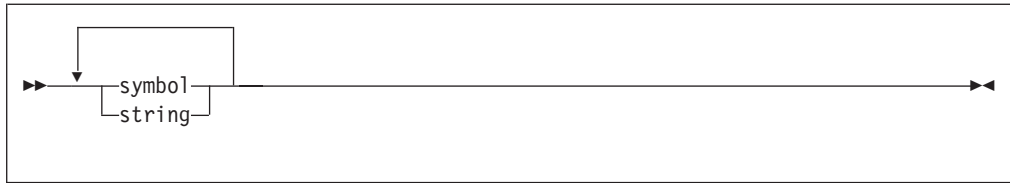


When a string is parsed (for example, when converting the string to a date`Time`), the pattern or set of tokens is used to determine which part of the target date`Time` is represented by which part of the string. The following diagram shows how this is done.



Syntax

The expression pattern is defined by:



Where:

symbol

is a character in the set **adDeEFGhHIkKmmSSTUwWyYzZ**.

string is a sequence of characters enclosed in single quotation marks. If a single quotation mark is required within the string, use two single quotation marks ("").

Characters for formatting a dateTime as a string

The following table lists the characters that you can use in a pattern for formatting or parsing strings in relation to a dateTime. The table is followed by some notes that explain more about some of the examples in the table.

Symbol	Meaning	Presentation	Examples
a	am or pm marker	Text	Input am, AM, pm, PM. Output AM or PM
d	day in month (1-31)	Number	1, 20
dd	day in month (01-31)	Number	01, 31
D	day in year (1-366)	Number	3, 80, 100
DD	day in year (01-366)	Number	03, 80, 366
DDD	day in year (001-366)	Number	003
e	day in week (1-7) ¹	Number	2
EEE	day in week ¹	Text	Tue
EEEE	day in week ¹	Text	Tuesday
F	day of week in month (1-5) ²	Number	2
G	Era	Text	BC or AD
h	hour in am or pm (1-12)	Number	6
hh	hour in am or pm (01-12)	Number	06
H	hour of day in 24 hour form (0-23) ³	Number	7
HH	hour of day in 24 hour form (00-23) ³	Number	07
I	ISO8601 Date/Time (up to yyyy-MM-dd'THH:mm:ss.SSSZZZ) ⁴	Text	2006-10-07T12:06:56.568+01:00
IU	ISO8601 Date/Time (similar to I, but ZZZ with output "Z" if the time zone is +00:00) ⁴	Text	2006-10-07T12:06:56.568+01:00, 2003-12 -15T15:42:12.000Z
k	hour of day in 24 hour form (1-24) ³	Number	8

Symbol	Meaning	Presentation	Examples
kk	hour of day in 24 hour form (01-24) ³	Number	08
K	hour in am or pm (0-11)	Number	9
KK	hour in am or pm (00-11)	Number	09
m	minute	Number	4
mm	minute	Number	04
M	numeric month	Number	5, 12
MM	numeric month	Number	05, 12
MMM	named month	Text	Jan, Feb
MMMM	named month	Text	January, February
s	seconds	Number	5
ss	seconds	Number	05
S	decisecond ⁵	Number	7
SS	centisecond ⁵	Number	70
SSS	millisecond ⁵	Number	700
SSSS	0.0001 second ⁵	Number	7000
SSSSS	0.00001 second ⁵	Number	70000
SSSSSS	0.000001 second ⁵	Number	700000
T	ISO8601 Time (up to HH:mm:ss.SSSZZZ) ⁴	Text	12:06:56.568+01:00
TU	ISO8601 Time (similar to T, but a time zone of +00:00 is replaced with 'Z') ⁴	Text	12:06:56.568+01:00, 15:42:12.000Z
w	week in year ⁶	Number	7, 53
ww	week in year ⁶	Number	07, 53
W	week in month ⁷	Number	2
yy	year ⁸	Number	06
yyyy	year ⁸	Number	2006
YY	year: use with week in year only ⁶	Number	06
YYYY	year: use with week in year only ⁶	Number	2006
zzz	time zone (abbreviated name) ⁹	Text	GMT
zzzz	time zone (full name)	Text	Greenwich Mean Time
Z	time zone (+/-n)	Text	+3
ZZ	time zone (+/-nn)	Text	+03
ZZZ	time zone (+/-nn:nn)	Text	+03:00
ZZZU	time zone (as ZZZ, "+00:00" is replaced by "Z")	Text	+03:00, Z
ZZZZ	time zone (GMT+/-nn:nn)	Text	GMT+03:00
ZZZZZ	time zone (as ZZZ, but no colon) (+/-nnnn)	Text	+0300

Symbol	Meaning	Presentation	Examples
'	escape for text		'User text'
"	(two single quotation marks) single quotation mark within escaped text		'o'clock'

The presentation of the `dateTime` object depends on what symbols you specify.

- **Text.** If you specify four or more of the symbols, the full form is presented. If you specify less than four symbols, the short or abbreviated form, if it exists, is presented. For example, `EEEE` produces `Monday`, `EEE` produces `Mon`.
- **Number.** The number of characters for a numeric `dateTime` component must be within the bounds of the corresponding formatting symbols. Repeat the symbol to specify the minimum number of digits that are required. The maximum number of digits allowed is the upper bound for a particular symbol. For example, `day in month` has an upper bound of 31; therefore, a format string of `d` allows the values 2 or 21 to be parsed but disallows the values 32 and 210. On output, numbers are padded with zeros to the specified length. A year is a special case; see note 8 in the list below. Fractional seconds are also special case; see note 5 below.
- Any characters in the pattern that are not in the ranges of `[a..'z']` and `[A..'Z']` are treated as quoted text. For example, characters like colon (`:`), comma (`,`), period (`.`), the number sign (hash or pound, `#`), the at sign (`@`), and space appear in the resulting time text even if they are not enclosed within single quotes.
- You can create formatting strings that produce unpredictable results; therefore, you must use these symbols with care. For example, if you specify `dMyyyy`, you cannot distinguish between day, month, and year. `dMyyyy` tells the broker that a minimum of one character represents the day, a minimum of one character represents the month, and four characters represent the year. Therefore, `3111999` can be interpreted as either `3/11/1999` or `31/1/1999`.

Notes: The following notes apply to the table above.

1. You can specify the following values in the `day in week` field:
 - 1 - Sunday
 - 2 - Monday
 - 3 - Tuesday
 - 4 - Wednesday
 - 5 - Thursday
 - 6 - Friday
 - 7 - Saturday
2. 12th July 2006 is the second Wednesday in July and can be expressed as `2006 July Wednesday 2` using the format string `yyyy MMMM EEEE F`. Note that this format does not represent the Wednesday in week 2 of July 2006, which is 5th July 2006; the format string for this is `yyyy MMMM EEEE W`.
3. 24-hour fields might result in an ambiguous time, if specified with a conflicting `am/pm` field.
4. See "ISO8601, I and T Date/Time tokens" on page 343.
5. Fractional seconds are represented by uppercase `S`. The length must implicitly match the number of format symbols on input. The format string `ss SSS` or `ss.SSS`, for example, represents seconds and

milliseconds. However, the format string `ss.sss` represents a repeated field (of seconds); the value after the period (`.`) is taken as a seconds field, not as fractional seconds. The output is truncated to the specified length.

6. In ESQL, the first day of the year is assumed to be in the first week; therefore, January 1 is always in week 1. As a result, dates that are specified relative to one year might actually be in a different year. For example, "Monday week 1 2005" parsed using "EEEE' week 'w' 'YYYY'" gives a date of 2004-12-27, because the Monday of the first week in 2005 is actually a date in 2004.

If you use the `y` symbol, the adjustment is not done and unpredictable results might occur for dates around the end of the year. For example, if the string "2005 01 Monday" is formatted:

- Monday of week 1 in 2005 using format string "YYYY ww EEEE" is correctly interpreted as 27th December 2004
 - Monday of week 1 in 2005 using format string "yyyy ww EEEE" is incorrectly interpreted as 27th December 2005
7. The first and last week in a month might include days from neighboring months. For example, Monday 31st July 2006 can be expressed as *Monday in week one of August 2006*, which is 2006 08 1 Monday using format string `yyyy MM W EEEE`.
 8. Year is handled as a special case.
 - On output, if the count of `y` is 2, the year is truncated to 2 digits. For example, if `yyyy` produces 1997, `yy` produces 97.
 - On input, for 2 digit years the century window is fixed to 53. For example, an input date of 52 results in a year value of 2052, whereas an input date of 53 gives an output year of 1953, and 97 gives 1997.
 9. Using the `zzz` option can have ambiguous results. For example, `BST` can be interpreted as Bangladesh Standard Time or British Summer Time. For compatibility reasons, WebSphere Message Broker uses the former interpretation.

To avoid these problems, use the `zzzz` option with a well-defined name; for example, `Europe/London`, `Asia/Dhaka`, or `America/Los_Angeles`.

ISO8601, I and T DateTime tokens

If your `dateTime` values comply with the ISO8601:2000 'Representation of dates and times' standard, consider using the formatting symbols `I` and `T`, which match the following subset of the ISO8601 standard.

- The restricted profile as proposed by the W3C at <http://www.w3.org/TR/NOTE-datetime>
- Truncated representations of calendar dates, as specified in section 5.2.1.3 of ISO8601:2000
 - Basic format (subsections c, e, and f)
 - Extended format (subsections a, b, and d)

Use the formatting symbols `I` and `T` only on their own:

- The `I` formatting symbol matches any `dateTime` string that conforms to the supported subset.
- The `T` formatting symbol matches any `dateTime` string that conforms to the supported subset that consists of a time portion only.

The following table shows how the output form relates to the logical data type.

Logical model data type	ESQL data type	Output form
xsd:dateTime	TIMESTAMP or GMTTIMESTAMP	yyyy-MM-dd'T'HH:mm:ss.SSSZZZ
xsd:date	DATE	yyyy-MM-dd
xsd:gYear	INTERVAL	yyyy
xsd:gYearMonth	INTERVAL	yyyy-MM
xsd:gMonth	INTERVAL	--MM
xsd:gmonthDay	INTERVAL	--MM-dd
xsd:gDay	INTERVAL	---dd
xsd:time	TIME / GMTTIME	'T'HH:mm:ss.SSSZZZ

Note:

- On input, both I and T accept both '+00:00' and 'Z' to indicate a zero time difference from Coordinated Universal Time (UTC), but on output they always generate '+00:00'. If you want 'Z' to always be generated on output, use the IU or TU formatting symbols instead.
- ZZZ always writes '+00:00' to indicate a zero time difference from Coordinated Universal Time (UTC). If you want 'Z' to always be generated on output, use ZZZU instead.

Using the input UTC format on output

An element or attribute of logical type xsd:dateTime or xsd:time that contains a dateTime as a string can specify Coordinated Universal Time (UTC) by using either the Z symbol or time zone +00:00. On input, the MRM parser remembers the UTC format of such elements and attributes. On output, you can specify whether Z or +00:00 is displayed by using the Default DateTime Format property of the element or attribute. Alternatively, you can preserve the input UTC format by selecting the message set property Use input UTC format on output. If this property is selected, the UTC format is preserved in the output message and overrides the format that is implied by the dateTime format property.

Understanding daylight saving time and the CAST function

When the broker is running in a time zone other than GMT, it calculates the daylight saving time (DST) offset on times that are supplied to it by the CAST function. For CAST to calculate the offset correctly, the time passed into CAST must have a time zone associated with it, as a Z parameter. If no time zone is associated with the value passed, the time is converted into GMT time; it is not treated as a local time stamp.

Also, when you use CAST to cast a string to a time value, the DST offset is calculated using the current system date. To cast a string to a time variable and calculate DST for a specific date, you must also specify the date.

For example, if timeValue='10:00:00', the following code, run on a broker that is in the Central Daylight Time zone, converts the time to GMT, because no time zone identifier is specified:

```
DECLARE castTime TIME;
SET castTime = CAST (timeValue AS TIME FORMAT timePattern)
```


The time is not converted into GMT again if the castTime variable is used in any subsequent code, for example

```
CAST(castDate, castTime AS GMTTIMESTAMP);
```

Examples

The following table shows a few examples of dateTime formats.

Format pattern	Result
"yyyy.MM.dd 'at' HH:mm:ss ZZZ"	2006.07.10 at 15:08:56 -05:00
"EEE, MMM d, 'yy"	Wed, July 10, '06
"h:mm a"	8:08 PM
"hh o'clock a, ZZZZ"	09 o'clock AM, GMT+09:00
"K:mm a, ZZZ"	9:34 AM, -05:00
"yyyy.MMMMMM.dd hh:mm aaa"	1996.July.10 12:08 PM

Use within an MRM domain

In MRM it is possible to define an element that has the logical type of dateTime.

When a dateTime element is parsed, a field is created in the message tree that has the ESQL datatype of CURRENT_TIME or CURRENT_TIMESTAMP. However, the CURRENT_TIME and CURRENT_TIMESTAMP data types do not have the functionality to store timezone information, and the MRM does not adjust the time according to the input timezone and the timezone of the broker.

Although the CURRENT_TIME and CURRENT_TIMESTAMP data types cannot store timezone information, the MRM stores this information as part of the underlying field. This means that if the field is copied between message trees, the timezone information is copied with it, allowing this information to be preserved on output.

Note that the information is preserved only if the field is copied to a field of the same name.

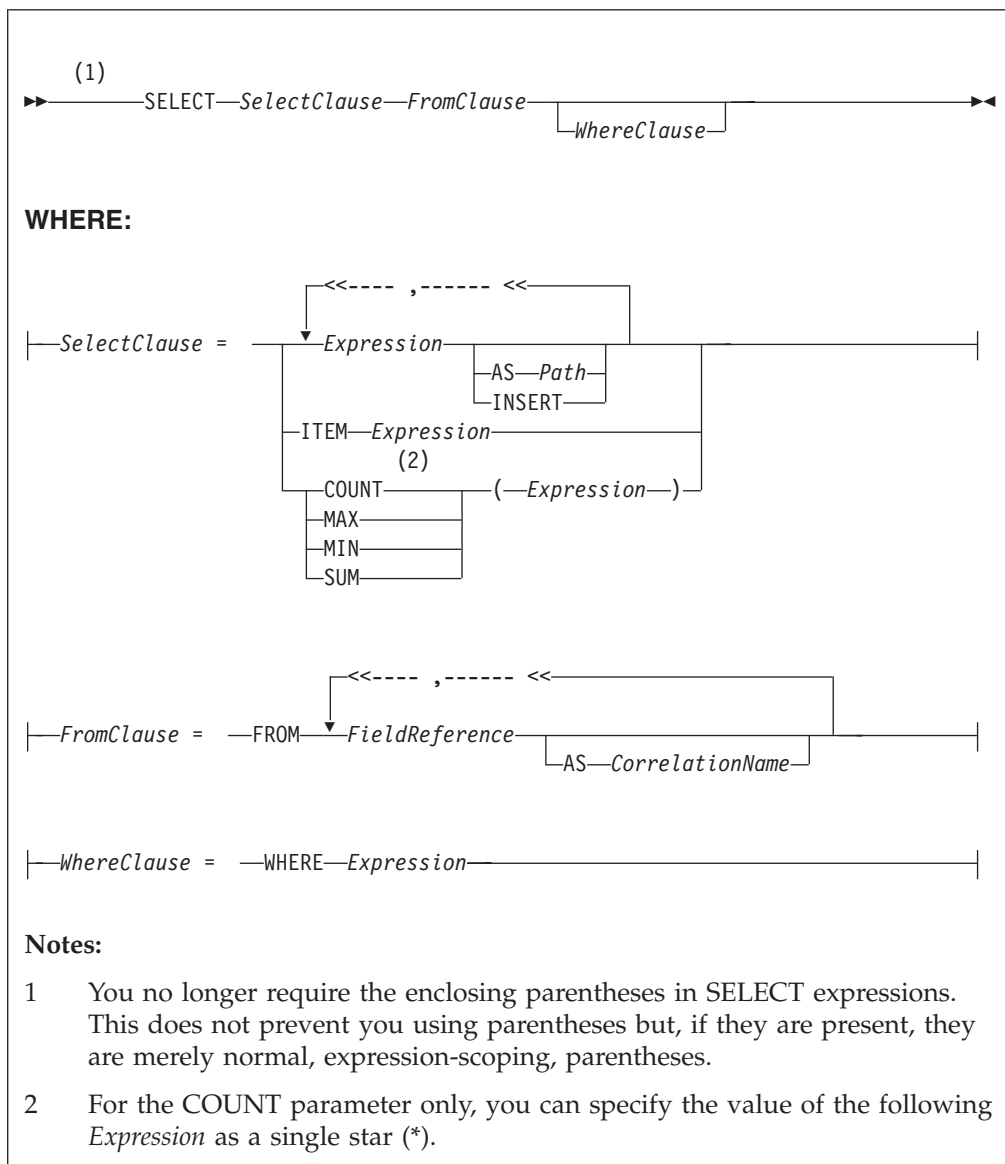
However, if any new field is derived from the original field, the new field does not have the timezone information. This means that if such a field is cast as a character, the new field assumes the timezone of the broker, but its value is not adjusted for any difference between the input timezone and the timezone of the broker.

For example, an input dateTime element containing 2009-02-20T06:08:07-08:00 could be copied from the input message tree to the output message tree and appear in an output message in exactly the same format. However, if the element is cast as character, using format IU, by a broker running GMT the result would be 2009-02-20T06:08:07.000Z.

SELECT function

The SELECT function combines, filters, and transforms complex message and database data.

Syntax



Usage

The SELECT function is the usual and most efficient way of transforming messages. You can use SELECT to:

- Comprehensively reformat messages
- Access database tables
- Make an output array that is a subset of an input array
- Make an output array that contains only the values of an input array
- Count the number of entries in an array
- Select the minimum or maximum value from a number of entries in an array
- Sum the values in an array

Introduction to SELECT

The SELECT function considers a message tree (or sub-tree) to consist of a number of rows and columns, rather like a database table. A *FieldReference* in a FROM clause identifies a field in a message tree. The identified field is regarded in the following ways:

- The identified field is regarded as a row in a table.
- The field's siblings are regarded as other rows of the same table.
- The field's children are regarded as the table's columns.

Note: The *FieldReference* in a FROM clause can also be a table reference that refers directly to a real database table.

The return value of the SELECT function is typically another message tree that contains rows whose structure and content is determined by the *SelectClause*. The number of rows in the result is the sum of all the rows pointed to by all the field references and table references in the FROM clause, filtered by the WHERE clause; only those fields for which the WHERE clause evaluates to TRUE are included.

The return value of the SELECT function can also be scalar (see “ITEM selections” on page 349).

You can specify the *SelectClause* in several ways; see:

- “Simple selections”
- “INSERT selections” on page 349
- “ITEM selections” on page 349
- “Column function selections” on page 349

If you have created a message flow that contains one of the following nodes, and the ESQL that is associated with this node includes a SELECT function and a database reference, you must specify a value for the Data source property of the relevant node:

- Compute
- Database
- Filter

Simple selections

To understand the SELECT function in more detail, first consider the following simple case:

- The *SelectClause* consists of a number of expressions, each with an AS *Path* clause.
- The FROM clause contains a single *FieldReference* and an AS *CorrelationName* clause.

The SELECT function creates a local, reference, correlation variable, whose name is given by the AS *CorrelationName* clause, and then steps, in turn, through each row of the list of rows derived from the FROM clause. For each row:

1. The correlation variable is set to point to the current row.
2. The WHERE clause (if present) is evaluated. If it evaluates to FALSE or unknown (null), nothing is added to the result tree and processing proceeds to the next row of the input. Otherwise processing proceeds to the next step.
3. A new member is added to the result list.

4. The SELECT clause expressions are evaluated and assigned to fields named as dictated by the AS *Path* clause. These fields are child fields of the new member of the result list.

Typically, both the *SelectClause* and the WHERE clause expressions use the correlation variable to access column values (that is, fields in the input message tree) and thus to build a new message tree containing data from the input message. The correlation variable is referred to by the name specified in the AS *CorrelationName* clause or, if an AS clause is not specified, by the final name in the FROM *FieldReference* (that is, the name after the last dot).

Note that:

- Despite the analogy with a table, you are not restricted to accessing or creating messages with a flat, table-like, structure; you can access and build trees with arbitrarily deep folder structures.
- You are not restricted to a column being a single value; a column can be a repeating list value or a structure.

These concepts are best understood by reference to the examples.

If the field reference is actually a *TableReference*, the operation is very similar. In this case, the input is a real database table and is thus restricted to the flat structures supported by databases. The result tree is still not so restricted, however.

If the FROM clause contains more than one field reference, the rightmost reference steps through each of its rows for each row in the next-to-rightmost reference, and so on. The total number of rows in the result is thus the product of the number of rows in each table. Such selects are known as *joins* and commonly use a WHERE clause that excludes most of these rows from the result. Joins are commonly used to add database data to messages.

The AS *Path* clause is optional. If it is unspecified, the broker generates a default name according to the following rules:

1. If the *SelectClause* expression is a reference to a field or a cast of a reference to a field, the name of the field is used.
2. Otherwise the broker uses the default names Column1, Column2, and so on.

Examples

The following example performs a SELECT on the table Parts in the schema Shop in the database DSN1. Because no WHERE clause exists, all rows are selected. Because the select clause expressions (for example, P.PartNumber) contain no AS clauses, the fields in the result adopt the same names:

```
SET PartsTable.Part[] = SELECT
  P.PartNumber,
  P.Description,
  P.Price
FROM Database.DSN1.Shop.Parts AS P;
```

If the target of the SET statement (PartsTable) is a variable of type ROW, after the statement is executed PartsTable will have, as children of its root element, a field called Part for each row in the table. Each of the Part fields will have child fields called PartNumber, Description, and Price. The child fields will have values dictated by the contents of the table. (PartsTable could also be a reference into a message tree).

The next example performs a similar SELECT. This case differs from the last in that the SELECT is performed on the message tree produced by the first example (rather than on a real database table). The result is assigned into a subfolder of OutputRoot:

```
SET OutputRoot.XMLNS.Data.TableData.Part[] = SELECT
    P.PartNumber,
    P.Description,
    P.Price
FROM PartsTable.Part[] AS P;
```

INSERT selections

The INSERT clause is an alternative to the AS clause. It assigns the result of the *SelectClause* expression (which must be a row) to the current new row itself, rather than to a child of it. The effect of this is to merge the row result of the expression into the row being generated by the SELECT. This differs from the AS clause, in that the AS clause always generates at least one child element before adding a result, whereas INSERT generates none. INSERT is useful when inserting data from other SELECT operations, because it allows the data to be merged without extra folders.

ITEM selections

The *SelectClause* can consist of the keyword ITEM and a single expression. The effect of this is to make the results nameless. That is, the result is a list of values of the type returned by the expression, rather than a row. This option has several uses:

- In conjunction with a scalar expression and the THE function, it can be used to create a SELECT query that returns a single scalar value (for example, the price of a particular item from a table).
- In conjunction with a CASE expression and ROW constructors, it can be used to create a SELECT query that creates or handles messages in which the structure of some rows (that is, repeats in the message) is different from others. This is useful for handling messages that have a repeating structure but in which the repeats do not all have the same structure.
- In conjunction with a ROW constructor, it can be used to create a SELECT query that collapses levels of repetition in the input message.

Column function selections

The *SelectClause* can consist of one of the functions COUNT, MAX, MIN, and SUM operating on an expression. These functions are known as column functions. They return a single scalar value (not a list) giving the count, maximum, minimum, or sum of the values that *Expression* evaluated to in stepping through the rows of the FROM clause. If *Expression* evaluates to NULL for a particular row, the value is ignored, so that the function returns the count, maximum, minimum, or sum of the remaining rows.

For the COUNT function only, *Expression* can consist of a single star (*). This form counts the rows regardless of null values.

To make the result a useful reflection of the input message, *Expression* typically includes the correlation variable.

Typically, *Expression* evaluates to the same data type for each row. In these cases, the result of the MAX, MIN, and SUM functions are of the same data type as the

operands. The returned values are not required to be all of the same type however, and if they are not, the normal rules of arithmetic apply. For example, if a field in a repeated message structure contains integer values for some rows and float values for others, the sum follows the normal rules for addition. The sum is of type float because the operation is equivalent to adding a number of integer and float values.

The result of the COUNT function is always an integer.

Differences between message and database selections

FROM expressions in which a correlation variable represents a row in a message behave slightly differently from those in which the correlation variable represents a row in a real database table.

In the message case, a path involving a star (*) has the normal meaning; it ignores the field's name and finds the first field that matches the other criteria (if any).

In the database case a star (*) has, for historical reasons, the special meaning of all fields. This special meaning requires advance knowledge of the definition of the database table and is supported only when querying the default database (that is, the database pointed to by the node's data source attribute). For example, the following queries return column name and value pairs only when querying the default database:

```
SELECT * FROM Database.Datasource.SchemaName.Table As A
SELECT A.* FROM Database.Datasource.SchemaName.Table As A
SELECT A FROM Database.Datasource.SchemaName.Table AS A
```

Specifying the SELECT expressions

SelectClause

SelectClause expressions can use any of the broker's operators and functions in any combination. They can refer to the tables' columns, message fields, correlation names declared by containing SELECTs, and to any other declared variables or constants that are in scope.

AS Path

An *AS Path* expression is a relative path (that is, there is no correlation name) but is otherwise unrestricted in any way. For example, it can contain:

- Indexes (for example, A.B.C[i])
- Field-type specifiers (for example, A.B.(XML.Attribute)C)
- Multipart paths (for example, A.B.C)
- Name expressions (for example, A.B.{var})

Any expressions in these paths can also use any of the broker's operators and functions in any combination. The expressions can refer to the tables' columns, message fields, correlation names declared by containing SELECTs, and any declared variables or constants.

FROM clause

FROM clause expressions can contain multiple database references, multiple message references, or a mixture of the two. You can join tables with tables, messages with messages, or tables with messages.

FROM clause *FieldReferences* can contain expressions of any kind (for example, Database.{DataSource}.{Schema}.Table1).

You can calculate a field, data source, schema, or table name at run time.

WHERE clause

The WHERE clause expression can use any of the broker's operators and functions in any combination. It can refer to table columns, message fields, and any declared variables or constants.

However, be aware that the broker treats the WHERE clause expression by examining the expression and deciding whether the whole expression can be evaluated by the database. If it can, it is given to the database. In order to be evaluated by the database, it must use only those functions and operators supported by the database.

The WHERE clause can, however, refer to message fields, correlation names declared by containing SELECT functions, and to any other declared variables or constants within scope.

If the whole expression cannot be evaluated by the database, the broker looks for top-level AND operators and examines each sub-expression separately. It then attempts to give the database those sub-expressions that it can evaluate, leaving the broker to evaluate the rest. You need to be aware of this situation for two reasons:

1. Apparently trivial changes to WHERE clause expressions can have large effects on performance. You can determine how much of the expression was given to the database by examining a user trace.
2. Some databases' functions exhibit subtle differences of behavior from those of the broker.

Relation to the THE function

You can use the function THE (which returns the first element of a list) in conjunction with SELECT to produce a non-list result. This is useful, for example, when a SELECT query is required to return no more than one item. It is particularly useful in conjunction with ITEM (see "ITEM selections" on page 349).

Differences from the SQL standard

ESQL SELECT differs from database SQL SELECT in the following ways:

- ESQL can produce tree-structured result data
- ESQL can accept arrays in SELECT clauses
- ESQL has the THE function and the ITEM and INSERT parameters
- ESQL has no SELECT ALL function in this release
- ESQL has no ORDER BY function in this release
- ESQL has no SELECT DISTINCT function in this release
- ESQL has no GROUP BY or HAVING parameters in this release
- ESQL has no AVG column function in this release

Restrictions

The following restrictions apply to the current release:

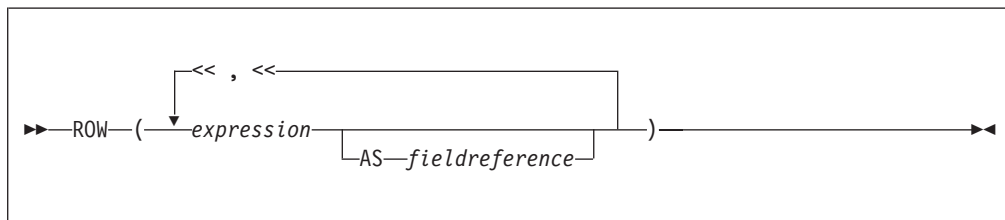
- When a SELECT command operates on more than one database table, all the tables must be in the same database instance. (That is, the *TableReferences* must not specify different data source names.)
- If the FROM clause refers to both messages and tables, the tables must precede the messages in the list.

- Using dynamic DSN, SCHEMA and TABLE names with 'SELECT *' statements is not supported. If you use a schema, table or datasource name as a variable (dynamic variables) in 'SELECT *' queries, the variables are not resolved to the correct set of schema or table names.

ROW constructor function

ROW constructor is a complex function used to explicitly generate rows of values that can be assigned to fields in an output message.

Syntax



A ROW consists of a sequence of named values. When assigned to a field reference it creates that sequence of named values as child fields of the referenced field. A ROW cannot be assigned to an array field reference.

Examples:

Example 1

```
SET OutputRoot.XMLNS.Data = ROW('granary' AS bread,
                                'riesling' AS wine,
                                'stilton' AS cheese);
```

produces:

```
<Data>
  <bread>granary</bread>
  <wine>riesling</wine>
  <cheese>stilton</cheese>
</Data>
```

Example 2

Given the following XML input message body:

```
<Proof>
  <beer>5</beer>
  <wine>12</wine>
  <gin>40</gin>
</Proof>
```

the following ESQL:

```
SET OutputRoot.XMLNS.Data = ROW(InputBody.Proof.beer,
                                InputBody.Proof.wine AS vin,
                                (InputBody.Proof.gin * 2) AS special);
```

produces the following result:


```

<Data>
  <beer>5</beer>
  <vin>12</vin>
  <special>80</special>
</Data>

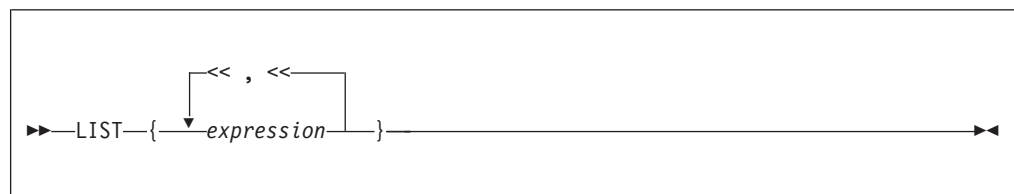
```

Because the values in this case are derived from field references that already have names, it is not necessary to explicitly provide a name for each element of the row, but you might choose to do so.

LIST constructor function

The LIST constructor complex function is used to explicitly generate lists of values that can be assigned to fields in an output message.

Syntax



A LIST consists of a sequence of unnamed values. When assigned to an array field reference (indicated by [] suffixed to the last element of the reference), each value is assigned in sequence to an element of the array. A LIST cannot be assigned to a non-array field reference.

Examples:

Example 1

Given the following XML message input body:

```

<Car>
  <size>big</size>
  <color>red</color>
</Car>

```

The following ESQL:

```

SET OutputRoot.XMLNS.Data.Result[] = LIST{InputBody.Car.colour,
                                           'green',
                                           'blue'};

```

produces the following results:

```

<Data>
  <Result>red</Result>
  <Result>green</Result>
  <Result>blue</Result>
</Data>

```

In the case of a LIST, there is no explicit name associated with each value. The values are assigned in sequence to elements of the message field array specified as the target of the assignment. Curly braces rather than parentheses are used to surround the LIST items.

Example 2

Given the following XML input message body:

```
<Data>
  <Field>Keats</Field>
  <Field>Shelley</Field>
  <Field>Wordsworth</Field>
  <Field>Tennyson</Field>
  <Field>Byron</Field>
</Data>
```

the following ESQL:

```
-- Copy the entire input message to the output message,
-- including the XML message field array as above
SET OutputRoot = InputRoot;
SET OutputRoot.XMLNS.Data.Field[] = LIST{'Henri','McGough','Patten'};
```

Produces the following output:

```
<Data>
  <Field>Henri</Field>
  <Field>McGough</Field>
  <Field>Patten</Field>
</Data>
```

The previous members of the `Data.Field[]` array have been discarded. Assigning a new list of values to an already existing message field array removes all the elements in the existing array before the new ones are assigned.

ROW and LIST combined

ROW and LIST combined form a complex function.

A ROW might validly be an element in a LIST. For example:

```
SET OutputRoot.XMLNS.Data.Country[] =
  LIST{ROW('UK' AS name, 'pound' AS currency),
       ROW('US' AS name, 'dollar' AS currency),
       'default'};
```

produces the following result:

```
<Data>
  <Country>
    <name>UK</name>
    <currency>pound</currency>
  </Country>
  <Country>
    <name>US</name>
    <currency>dollar</currency>
  </Country>
  <Country>default</Country>
</Data>
```

ROW and non-ROW values can be freely mixed within a LIST.

A LIST cannot be a member of a ROW. Only named scalar values can be members of a ROW.

ROW and LIST comparisons

You can compare ROWs and LISTs against other ROWs and LISTs.

Examples:

Example 1

```
IF ROW(InputBody.Data.*[1],InputBody.Data.*[2]) =  
    ROW('Raf' AS Name,'25' AS Age) THEN ...  
IF LIST{InputBody.Data.Name, InputBody.Data.Age} = LIST{'Raf','25'} THEN ...
```

With the following XML input message body both the IF expressions in both the above statements evaluate to TRUE:

```
<Data>  
  <Name>Raf</Name>  
  <Age>25</Age>  
</Data>
```

In the comparison between ROWs, both the name and the value of each element are compared; in the comparison between LISTS only the value of each element is compared. In both cases, the cardinality and sequential order of the LIST or ROW operands being compared must be equal in order for the two operands to be equal. Therefore, the following examples are false because either the sequential order or the cardinality of the operands being compared do not match:

```
ROW('alpha' AS A, 'beta' AS B) =  
    ROW('alpha' AS A, 'beta' AS B, 'delta' AS D)  
ROW('alpha' AS A, 'beta' AS B) =  
    ROW('beta' AS B, 'alpha' AS A)  
LIST{1,2,3} = LIST{1,2,3,4}  
LIST{3,2,1} = LIST{1,2,3}
```

Example 2

Consider the following ESQL:

```
IF InputBody.Places =  
    ROW('Ken' AS first, 'Bob' AS second, 'Kate' AS third) THEN ...
```

With the following XML input message body, the above IF expression evaluates to TRUE:

```
<Places>  
  <first>Ken</first>  
  <second>Bob</second>  
  <third>Kate</third>  
</Places>
```

The presence of an explicitly-constructed ROW as one of the operands to the comparison operator results in the other operand also being treated as a ROW.

Contrast this with a comparison such as:

```
IF InputBody.Lottery.FirstDraw = InputBody.Lottery.SecondDraw THEN ...
```

which compares the value of the FirstDraw and SecondDraw fields, not the names and values of each of FirstDraw and SecondDraw's child fields constructed as a ROW. Thus an XML input message body such as:

```
<Lottery>  
  <FirstDraw>wednesday  
    <ball1>32</ball1>  
    <ball2>12</ball2>  
  </FirstDraw>  
  <SecondDraw>saturday  
    <ball1>32</ball1>  
    <ball2>12</ball2>  
  </SecondDraw>  
</Lottery>
```

would not result in the above IF expression being evaluated as TRUE, because the values wednesday and saturday are being compared, not the names and values of the ball fields.

Example 3

Consider the following ESQL:

```
IF InputBody.Cities.City[] = LIST{'Athens','Sparta','Thebes'} THEN ...
```

With the following XML input message body, the IF expression evaluates to TRUE:

```
<Cities>
  <City>Athens</City>
  <City>Sparta</City>
  <City>Thebes</City>
</Cities>
```

Two message field arrays can be compared together in this way, for example:

```
IF InputBody.Cities.Mediaeval.City[] =
    InputBody.Cities.Modern.City[] THEN ...
```

```
IF InputBody.Cities.Mediaeval.*[] = InputBody.Cities.Modern.*[] THEN ...
```

```
IF InputBody.Cities.Mediaeval.(XML.Element)[] =
    InputBody.Cities.Modern.(XML.Element)[] THEN ...
```

With the following XML input message body, the IF expression of the first and third of the statements above evaluates to TRUE:

```
<Cities>
  <Mediaeval>1350
    <City>London</City>
    <City>Paris</City>
  </Mediaeval>
  <Modern>1990
    <City>London</City>
    <City>Paris</City>
  </Modern>
</Cities>
```

However the IF expression of the second statement evaluates to FALSE, because the *[] indicates that all the children of Mediaeval and Modern are to be compared, not just the (XML.Element)s. In this case the values 1350 and 1990, which form nameless children of Mediaeval and Modern, are compared as well as the values of the City tags.

The IF expression of the third statement above evaluates to TRUE with an XML input message body such as:

```
<Cities>
  <Mediaeval>1350
    <Location>London</Location>
    <Location>Paris</Location>
  </Mediaeval>
  <Modern>1990
    <City>London</City>
    <City>Paris</City>
  </Modern>
</Cities>
```

LISTs are composed of unnamed values. It is the values of the child fields of Mediaeval and Modern that are compared, not their names.

Supported casts

This topic lists the CASTs that are supported between combinations of data-types.

A CAST is not supported between every combination of data-types. Those that are supported are listed below, along with the effect of the CAST.

When casting, there can be a one-to-one or a many-to-one mapping between the source data-type and the target data-type. An example of a one-to-one mapping is where the source data-type is a single integer and the target data-type a single float. An example of a many-to-one mapping is where the source data consists of three integers that are converted to a single date. Table 7 lists the supported one-to-one casts. Table 8 on page 364 lists the supported many-to-one casts.

See “ESQL data types” on page 4 for information about precision, scale, and interval qualifier.

Table 7. Supported casts: one-to-one mappings of source to target data-type

Source data-type	Target data-type	Effect
BIT	BIT	The result is the same as the input.
BIT	BLOB	The bit array is converted to a byte array with a maximum of 2^{63} elements. An error is reported if the source is not of a suitable length to produce a BLOB (that is a multiple of 8).
BIT	CHARACTER	<p>The result is a string conforming to the definition of a bit string literal whose interpreted value is the same as the source value. The resulting string has the form B'bbbbbb' (where b is either 0 or 1).</p> <p>If you specify either a CCSID or ENCODING clause, the bit array is assumed to be characters in the specified CCSID and encoding, and is code-page converted into the character return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown, the data supplied is not an integral number of characters of the code page, or the data contains characters that are not valid in the given code page.</p>
BIT	INTEGER	The bit array has a maximum of 2^{63} elements and is converted to an integer. An error is reported if the source is not of the correct length to match an integer.
BLOB	BIT	The given byte array is converted to a bit array with a maximum of 2^{63} elements.
BLOB	BLOB	The result is the same as the input.

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
BLOB	CHARACTER	<p>The result is a string conforming to the definition of a binary string literal whose interpreted value is the same as the source value. The resulting string has the form X'hhhh' (where h is any hexadecimal character).</p> <p>If you specify either a CCSID or ENCODING clause, the byte array is assumed to be characters in the specified CCSID and encoding, and is code-page converted into the character return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown, the data supplied is not an integral number of characters of the code page, or the data contains characters that are not valid in the given code page.</p>
BLOB	INTEGER	<p>The byte array has a maximum of 2⁶³ elements and is converted to an integer. An error is reported if the source is not of the correct length to match an integer.</p>
BOOLEAN	BOOLEAN	<p>The result is the same as the input.</p>
BOOLEAN	CHARACTER	<p>If the source value is TRUE, the result is the character string TRUE. If the source value is FALSE, the result is the character string FALSE. Because the UNKNOWN Boolean value is the same as the NULL value for Booleans, the result is NULL if the source value is UNKNOWN.</p>
CHARACTER	BIT	<p>The character string must conform to the rules for a bit string literal or for the contents of the bit string literal. That is, the character string can be of the form B'bbbbbb' or bbbbb (where b' can be either 0 or 1).</p> <p>If you specify either a CCSID or ENCODING clause, the character string is converted into the specified CCSID and encoding and placed without further conversion into the bit array return value.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown or the data contains Unicode characters that cannot be converted to the given code page.</p>

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
CHARACTER	BLOB	<p>This cast can work in two ways:</p> <ol style="list-style-type: none"> 1. If you specify either a CCSID or ENCODING clause, the whole string is written out in the code page or encoding that you requested. For example, the string "Cat" in CCSID 850 becomes the three-byte array in hexadecimal, 43,61,74. 2. If you specify neither the CCSID nor ENCODING clause, the string must itself contain two-character hexadecimal digits of the form X'hhhhhh' or hhhhhh (where h can be any hexadecimal characters). In this case, the input string "436174" becomes the same three-byte binary array (43,61,74). <p>Note that an error is generated if the input string is not of the correct format.</p> <p>If you specify only a CCSID, big endian encoding is assumed.</p> <p>If you specify only an encoding, a CCSID of 1208 is assumed.</p> <p>This function reports conversion errors if the code page or encoding are unknown or the data contains Unicode characters that cannot be converted to the given code page.</p>
CHARACTER	BOOLEAN	<p>The character string is interpreted in the same way as a Boolean literal. That is, the character string must be one of the strings TRUE, FALSE, or UNKNOWN (in any case combination).</p>
CHARACTER	CHARACTER	<p>The result is the same as the input.</p>
CHARACTER	DATE	<p>If a FORMAT clause is not specified, the character string must conform to the rules for a date literal or the date string. That is, the character string can be either DATE '2002-10-05' or 2002-10-05.</p> <p>See also "Formatting and parsing dateTimes as strings" on page 339.</p>
CHARACTER	DECIMAL	<p>The character string is interpreted in the same way as an exact numeric literal to form a temporary decimal result with a scale and precision defined by the format of the string. This is converted into a decimal of the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits.</p> <p>If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.</p> <p>The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 336.</p>
CHARACTER	FLOAT	<p>The character string is interpreted in the same way as a floating point literal.</p> <p>The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 336.</p>
CHARACTER	GMTTIME	<p>The character string must conform to the rules for a GMT time literal or the time string. That is, the character string can be either GMTTIME '09:24:15' or 09:24:15.</p> <p>The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.</p>

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
CHARACTER	GMTTIMESTAMP	The character string must conform to the rules for a GMT timestamp literal or the timestamp string. That is, the character string can be either GMTTIMESTAMP '2002-10-05 09:24:15' or 2002-10-05 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.
CHARACTER	INTEGER	The character string is interpreted in the same way as an integer literal. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 336.
CHARACTER	INTERVAL	The character string must conform to the rules for an interval literal with the same interval qualifier as specified in the CAST function, or it must conform to the rules for an interval string that apply for the specified interval qualifier.
CHARACTER	TIME	The character string must conform to the rules for a time literal or for the time string. That is, the character string can be either TIME '09:24:15' or 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.
CHARACTER	TIMESTAMP	The character string must conform to the rules for a timestamp literal or for the timestamp string. That is, the character string can be either TIMESTAMP '2002-10-05 09:24:15' or 2002-10-05 09:24:15. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.
DATE	CHARACTER	The result is a string conforming to the definition of a date literal, whose interpreted value is the same as the source date value. For example: CAST(DATE '2002-10-05' AS CHARACTER) returns DATE '2002-10-05' The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.
DATE	DATE	The result is the same as the input.
DATE	GMTTIMESTAMP	The result is a value whose date fields are taken from the source date value, and whose time fields are taken from the current GMT time.
DATE	TIMESTAMP	The result is a value whose date fields are taken from the source date value, and whose time fields are taken from the current time.
DECIMAL	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the decimal. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 336.
DECIMAL	DECIMAL	The value is converted to the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits. If you do not specify the precision and scale, the value, precision and scale are preserved; it is a NOOP (no operation).

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
DECIMAL	FLOAT	The number is converted, with rounding if necessary.
DECIMAL	INTEGER	The value is rounded and converted into an integer, with a runtime error being generated if the conversion results in loss of significant digits.
DECIMAL	INTERVAL	If the interval qualifier specified has only one field, the result is an interval with that qualifier with the field equal to the value of the exact numeric. Otherwise a runtime error is generated.
FLOAT	CHARACTER	The result is the shortest character string that conforms to the definition of an approximate numeric literal and whose mantissa consists of a single digit that is not 0, followed by a period and an unsigned integer, and whose interpreted value is the value of the float. The behavior changes if the FORMAT clause is specified. See also “Formatting and parsing numbers as strings” on page 336.
FLOAT	FLOAT	The result is the same as the input.
FLOAT	DECIMAL	The value is rounded and converted into a decimal of the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits. If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.
FLOAT	INTEGER	The value is rounded and converted into an integer, with a runtime error being generated if the conversion results in loss of significant digits.
FLOAT	INTERVAL	If the specified interval qualifier has only one field, the result is an interval with that qualifier with the field equal to the value of the numeric. Otherwise a runtime error is generated.
GMTTIME	CHARACTER	The result is a string conforming to the definition of a GMTTIME literal whose interpreted value is the same as the source value. The resulting string has the form GMTTIME 'hh:mm:ss'. The behavior changes if the FORMAT clause is specified. See also “Formatting and parsing dateTimes as strings” on page 339.
GMTTIME	GMTTIME	The result is the same as the input.
GMTTIME	TIME	The resulting value is the source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIME	GMTTIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source GMT time.
GMTTIME	TIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source GMT time, plus the local time zone displacement (as returned by LOCAL_TIMEZONE).
GMTTIMESTAMP	CHARACTER	The result is a string conforming to the definition of a GMTTIMESTAMP literal whose interpreted value is the same as the source value. The resulting string has the form GMTTIMESTAMP 'yyyy-mm-dd hh:mm:ss'. The behavior changes if the FORMAT clause is specified. See also “Formatting and parsing dateTimes as strings” on page 339.
GMTTIMESTAMP	DATE	The result is a value whose fields consist of the date fields of the source GMTTIMESTAMP value.

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
GMTTIMESTAMP	GMTTIME	The result is a value whose fields consist of the time fields of the source GMTTIMESTAMP value.
GMTTIMESTAMP	TIME	The result is a value whose time fields are taken from the source GMTTIMESTAMP value, plus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
GMTTIMESTAMP	GMTTIMESTAMP	The result is the same as the input.
GMTTIMESTAMP	TIMESTAMP	The resulting value is source value plus the local time zone displacement (as returned by LOCAL_TIMEZONE).
INTEGER	BIT	The given integer is converted to a bit array with a maximum of 2^{63} elements.
INTEGER	BLOB	The given integer is converted to a byte array with a maximum of 2^{63} elements.
INTEGER	CHARACTER	The result is the shortest character string that conforms to the definition of an exact numeric literal and whose interpreted value is the value of the integer. The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing numbers as strings" on page 336.
INTEGER	FLOAT	The number is converted, with rounding if necessary.
INTEGER	INTEGER	The result is the same as the input.
INTEGER	DECIMAL	The value is converted into a decimal of the specified precision and scale, with a runtime error being generated if the conversion results in loss of significant digits. If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.
INTEGER	INTERVAL	If the interval qualifier specified has only one field, the result is an interval with that qualifier with the field equal to the value of the exact numeric. Otherwise a runtime error is generated.
INTERVAL	CHARACTER	The result is a string conforming to the definition of an INTERVAL literal, whose interpreted value is the same as the source interval value. For example: CAST(INTERVAL '4' YEARS AS CHARACTER) returns INTERVAL '4' YEARS
INTERVAL	DECIMAL	If the interval value has a qualifier that has only one field, the result is a decimal of the specified precision and scale with that value, with a runtime error being generated if the conversion results in loss of significant digits. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a runtime error is generated. If you do not specify the precision and scale, the precision and scale of the result are the minimum necessary to hold the given value.
INTERVAL	FLOAT	If the interval value has a qualifier that has only one field, the result is a float with that value. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a runtime error is generated.
INTERVAL	INTEGER	If the interval value has a qualifier that has only one field, the result is an integer with that value. If the interval has a qualifier with more than one field, such as YEAR TO MONTH, a runtime error is generated.

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
INTERVAL	INTERVAL	The result is the same as the input. Year-month intervals can be converted only to year-month intervals, and day-second intervals only to day-second intervals. The source interval is converted into a scalar in units of the least significant field of the target interval qualifier. This value is normalized into an interval with the target interval qualifier. For example, to convert an interval that has the qualifier MINUTE TO SECOND into an interval with the qualifier DAY TO HOUR, the source value is converted into a scalar in units of hours, and this value is normalized into an interval with qualifier DAY TO HOUR.
TIME	CHARACTER	The result is a string conforming to the definition of a TIME literal, whose interpreted value is the same as the source time value. For example: CAST(TIME '09:24:15' AS CHARACTER) returns TIME '09:24:15' The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.
TIME	GMTTIME	The result value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
TIME	GMTTIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source GMT time, minus the local time zone displacement (as returned by LOCAL_TIMEZONE).
TIME	TIME	The result is the same as the input.
TIME	TIMESTAMP	The result is a value whose date fields are taken from the current date, and whose time fields are taken from the source time value.
TIMESTAMP	CHARACTER	The result is a string conforming to the definition of a TIMESTAMP literal, whose interpreted value is the same as the source timestamp value. For example: CAST(TIMESTAMP '2002-10-05 09:24:15' AS CHARACTER) returns TIMESTAMP '2002-10-05 09:24:15' The behavior changes if the FORMAT clause is specified. See also "Formatting and parsing dateTimes as strings" on page 339.
TIMESTAMP	DATE	The result is a value whose fields consist of the date fields of the source timestamp value.
TIMESTAMP	GMTTIME	The result is a value whose time fields are taken from the source TIMESTAMP value, minus the local time zone displacement (as returned by LOCAL_TIMEZONE). The hours field is calculated modulo 24.
TIMESTAMP	GMTTIMESTAMP	The resulting value is the source value minus the local time zone displacement (as returned by LOCAL_TIMEZONE).
TIMESTAMP	TIME	The result is a value whose fields consist of the time fields of the source timestamp value.

Table 7. Supported casts: one-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
TIMESTAMP	TIMESTAMP	The result is the same as the input.

Table 8. Supported casts: many-to-one mappings of source to target data-type

Source data-type	Target data-type	Effect
Numeric, Numeric, Numeric	DATE	Creates a DATE value from the numerics in the order year, month, and day. Non-integer values are rounded.
Numeric, Numeric, Numeric	TIME	Creates a TIME value from the numerics in the order hours, minutes, and seconds. Non-integer values for hours and minutes are rounded.
Numeric, Numeric, Numeric	GMTIME	Creates a GMTIME value from the numerics in the order of hours, minutes, and seconds. Non-integer values for hours and minutes are rounded.
Numeric, Numeric, Numeric, Numeric, Numeric	TIMESTAMP	Creates a TIMESTAMP value from the numerics in the order years, months, days, hours, minutes, and seconds. Non-integer values for years, months, days, hours, and minutes are rounded.
Numeric, Numeric, Numeric, Numeric, Numeric	GMTTIMESTAMP	Creates a GMTTIMESTAMP value from the numerics in the order years, months, days, hours, minutes, and seconds. Non-integer values for years, months, days, hours, and minutes are rounded.
DATE, TIME	TIMESTAMP	The result is a TIMESTAMP value with the given DATE and TIME.
DATE, GMTIME	GMTTIMESTAMP	The result is a GMTTIMESTAMP value with the given DATE and GMTIME.
Numeric, Numeric	INTERVAL YEAR TO MONTH	The result is an INTERVAL with the first source as years and the second as months. Non-integer values are rounded.
Numeric, Numeric	INTERVAL HOUR TO MINUTE	The result is an INTERVAL with the first source as hours and the second as minutes. Non-integer values are rounded.
Numeric, Numeric, Numeric	INTERVAL HOUR TO SECOND	The result is an INTERVAL with the sources as hours, minutes, and seconds, respectively. Non-integer values for hours and minutes are rounded.
Numeric, Numeric	INTERVAL MINUTE TO SECOND	The result is an INTERVAL with the sources as minutes and seconds, respectively. Non-integer values for minutes are rounded.
Numeric, Numeric	INTERVAL DAY TO HOUR	The result is an INTERVAL with the sources as days and hours, respectively. Non-integer values are rounded.
Numeric, Numeric, Numeric	INTERVAL DAY TO MINUTE	The result is an INTERVAL with the sources as days, hours, and minutes, respectively. Non-integer values are rounded.
Numeric, Numeric, Numeric, Numeric	INTERVAL DAY TO SECOND	The result is an INTERVAL with the sources as days, hours, minutes, and seconds, respectively. Non-integer values for days, hours, and minutes are rounded.
Numeric	INTERVAL YEAR	The result is an INTERVAL with the source as years, rounded if necessary.
Numeric	INTERVAL MONTH	The result is an INTERVAL with the source as months, rounded if necessary.
Numeric	INTERVAL DAY	The result is an INTERVAL with the source as days, rounded if necessary.
Numeric	INTERVAL HOUR	The result is an INTERVAL with the source as hours, rounded if necessary.
Numeric	INTERVAL MINUTE	The result is an INTERVAL with the source as minutes, rounded if necessary.

Table 8. Supported casts: many-to-one mappings of source to target data-type (continued)

Source data-type	Target data-type	Effect
Numeric	INTERVAL SECOND	The result is an INTERVAL with the source as seconds.

Implicit casts

This topic discusses implicit casts.

It is not always necessary to cast values between types. Some casts are done implicitly. For example, numbers are implicitly cast between the three numeric types for the purposes of comparison and arithmetic. Character strings are also implicitly cast to other data types for the purposes of comparison.

There are three situations in which a data value of one type is cast to another type implicitly. The behavior and restrictions of the implicit cast are the same as described for the explicit cast function, except where noted in the topics listed below.

Implicit CASTs for comparisons

The standard SQL comparison operators $>$, $<$, $>=$, $<=$, $=$, $<>$ are supported for comparing two values in ESQL.

When the data types of the two values are not the same, one of them can be implicitly cast to the type of the other to allow the comparison to proceed. In the table below, the vertical axis represents the left hand operand, the horizontal axis represents the right hand operand.

L means that the right hand operand is cast to the type of the left hand operand before comparison; R means the opposite; X means that no implicit casting takes place; a blank means that comparison between the values of the two data types is not supported.

	ukn	bln	int	float	dec	chr	time	gtm	date	ts	gts	ivl	blob	bit
ukn														
bln		X				L								
int			X	R	R	L								
float			L	X	L	L								
dec			L	R	X	L								
chr		R	R	R	R	X	R	R	R	R	R	R ¹	R	R
tm						L	X	L						
gtm						L	R	X						
dt						L			X	R ²	R ²			
ts						L			L ²	X	L			
gts						L			L ²	R	X			
ivl						L ¹						X		
blb						L							X	
bit						L								X

	ukn	bln	int	float	dec	char	time	gtm	date	ts	gts	ivl	blob	bit
Notes:														
1. When casting from a character string to an interval, the character string must be of the format INTERVAL '<values>' <qualifier>. The format <values>, which is allowed for an explicit CAST, is not allowed here because no qualifier external to the string is supplied.														
2. When casting from a DATE to a TIMESTAMP or GMTTIMESTAMP, the time portion of the TIMESTAMP is set to all zero values (00:00:00). This is different from the behavior of the explicit cast, which sets the time portion to the current time.														

Numeric types:

The comparison operators operate on all three numeric types.

Character strings:

You cannot define an alternative collation order that, for example, collates upper and lowercase characters equally.

When comparing character strings, trailing blanks are not significant, so the comparison 'hello' = 'hello ' returns true.

Datetime values:

Datetime values are compared in accordance with the natural rules of the Gregorian calendar and clock.

You can compare the time zone you are working in with the GMT time zone. The GMT time zone is converted into a local time zone based on the difference between your local time zone and the GMT time specified. When you compare your local time with the GMT time, the comparison is based on the difference at a given time on a given date.

Conversion is always based on the value of LOCAL_TIMEZONE. This is because GMT timestamps are converted to local timestamps only if it can be done unambiguously. Converting a local timestamp to a GMT timestamp has difficulties around the daylight saving cut-over time, and converting between times and GMT times (without date information) has to be done based on the LOCAL_TIMEZONE value, because you cannot specify which time zone difference to use otherwise.

Booleans:

Boolean values can be compared using all the normal comparison operators. The TRUE value is defined to be greater than the FALSE value. Comparing either value to the UNKNOWN Boolean value (which is equivalent to NULL) returns an UNKNOWN result.

Intervals:

Intervals are compared by converting the two interval values into intermediate representations, so that both intervals have the same interval qualifier. Year-month intervals can be compared only with other year-month intervals, and day-second intervals can be compared only with other day-second intervals.

For example, if an interval in minutes, such as INTERVAL '120' MINUTE is compared with an interval in days to seconds, such as INTERVAL '0 02:01:00', the two

intervals are first converted into values that have consistent interval qualifiers, which can be compared. So, in this example, the first value is converted into an interval in days to seconds, which gives `INTERVAL '0 02:00:00'`, which can be compared with the second value.

Comparing character strings with other types:

If a character string is compared with a value of another type, WebSphere Message Broker attempts to cast the character string into a value of the same data type as the other value.

For example, you can write an expression:

```
'1234' > 4567
```

The character string on the left is converted into an integer before the comparison takes place. This behavior reduces some of the need for explicit `CAST` operators when comparing values derived from a generic XML message with literal values. (For details of explicit casts that are supported, see “Supported casts” on page 357.) It is this facility that allows you to write the following expression:

```
Body.Trade.Quantity > 5000
```

In this example, the field reference on the left evaluates to the character string '1000' and, because this is being compared to an integer, that character string is converted into an integer before the comparison takes place.

You must still check whether the price field that you want interpreted as a decimal is greater than a given threshold. Make sure that the literal you compare it to is a decimal value and not an integer.

For example:

```
Body.Trade.Price > 100
```

does not have the desired effect, because the `Price` field is converted into an integer, and that conversion fails because the character string contains a decimal point. However, the following expression succeeds:

```
Body.Trade.Price > 100.00
```

Implicit CASTs for arithmetic operations

This topic lists the implicit `CAST`s available for arithmetic operations.

Normally the arithmetic operators (+, -, *, and /) operate on operands of the same data type, and return a value of the same data type as the operands. Cases where it is acceptable for the operands to be of different data types, or where the data type of the resulting value is different from the type of the operands, are shown in the following table.

The following table lists the implicit `CAST`s for arithmetic operation.

Left operand data type	Right operand data type	Supported operators	Result data type
INTEGER	FLOAT	+, -, *, /	FLOAT ¹
INTEGER	DECIMAL	+, -, *, /	DECIMAL ¹
INTEGER	INTERVAL	*	INTERVAL ⁴

Left operand data type	Right operand data type	Supported operators	Result data type
FLOAT	INTEGER	+, -, *, /	FLOAT ¹
FLOAT	DECIMAL	+, -, *, /	FLOAT ¹
FLOAT	INTERVAL	*	INTERVAL ⁴
DECIMAL	INTEGER	+, -, *, /	DECIMAL ¹
DECIMAL	FLOAT	+, -, *, /	FLOAT ¹
DECIMAL	INTERVAL	*	INTERVAL ⁴
TIME	TIME	-	INTERVAL ²
TIME	GMTTIME	-	INTERVAL ²
TIME	INTERVAL	+, -	TIME ³
GMTTIME	TIME	-	INTERVAL ²
GMTTIME	GMTTIME	-	INTERVAL ²
GMTTIME	INTERVAL	+, -	GMTTIME ³
DATE	DATE	-	INTERVAL ²
DATE	INTERVAL	+, -	DATE ³
TIMESTAMP	TIMESTAMP	-	INTERVAL ²
TIMESTAMP	GMTTIMESTAMP	-	INTERVAL ²
TIMESTAMP	INTERVAL	+, -	TIMESTAMP ³
GMTTIMESTAMP	TIMESTAMP	-	INTERVAL ²
GMTTIMESTAMP	GMTTIMESTAMP	-	INTERVAL ²
GMTTIMESTAMP	INTERVAL	+, -	GMTTIMESTAMP ³
INTERVAL	INTEGER	*, /	INTERVAL ⁴
INTERVAL	FLOAT	*, /	INTERVAL ⁴
INTERVAL	DECIMAL	*, /	INTERVAL ⁴
INTERVAL	TIME	+	TIME ³
INTERVAL	GMTTIME	+	GMTTIME ³
INTERVAL	DATE	+	DATE ³
INTERVAL	TIMESTAMP	+	TIMESTAMP ³
INTERVAL	GMTTIMESTAMP	+	GMTTIMESTAMP ³

Left operand data type	Right operand data type	Supported operators	Result data type
Notes:			
1. The operand that does not match the data type of the result is cast to the data type of the result before the operation proceeds. For example, if the left operand to an addition operator is an INTEGER, and the right operand is a FLOAT, the left operand is cast to a FLOAT before the addition operation is performed.			
2. Subtracting a (GMT)TIME value from a (GMT)TIME value, a DATE value from a DATE value, or a (GMT)TIMESTAMP value from a (GMT)TIMESTAMP value, results in an INTERVAL value representing the time interval between the two operands.			
3. Adding or subtracting an INTERVAL from a (GMT)TIME, DATE or (GMT)TIMESTAMP value results in a new value of the data type of the non-INTERVAL operand, representing the point in time represented by the original non-INTERVAL, plus or minus the length of time represented by the INTERVAL.			
4. Multiplying or dividing an INTERVAL by an INTEGER, FLOAT, or DECIMAL value results in a new INTERVAL representing the length of time represented by the original, multiplied or divided by the factor represented by the non-INTERVAL operand. For example, an INTERVAL value of 2 hours 16 minutes multiplied by a FLOAT value of 2.5 results in a new INTERVAL value of 5 hours 40 minutes. The intermediate calculations involved in multiplying or dividing the original INTERVAL are carried out in the data type of the non-INTERVAL, but the individual fields of the INTERVAL (such as HOUR, YEAR, and so on) are always integral, so some rounding errors might occur.			

Implicit CASTs for assignment

Values can be assigned to one of three entities.

A message field (or equivalent in an exception or destination list)

Support for implicit conversion between the WebSphere Message Broker data types and the message (in its bitstream form) depends on the appropriate parser. For example, the XML parser casts everything as character strings before inserting them into the WebSphere MQ message.

A field in a database table

WebSphere Message Broker converts each of its data types into a suitable standard SQL C data type, as detailed in the following table. Conversion between this standard SQL C data type, and the data types supported by each DBMS, depends on the DBMS. Consult your DBMS documentation for more details.

The following table lists the available conversions from WebSphere Message Broker to SQL data types

WebSphere Message Broker data type	SQL data type
NULL, or unknown or invalid value	SQL_NULL_DATA
BOOLEAN	SQL_C_BIT
INTEGER	SQL_C_LONG
FLOAT	SQL_C_DOUBLE
DECIMAL	SQL_C_CHAR ¹
CHARACTER	SQL_C_CHAR
TIME	SQL_C_TIME
GMTTIME	SQL_C_TIME
DATE	SQL_C_DATE
TIMESTAMP	SQL_C_TIMESTAMP
GMTTIMESTAMP	SQL_C_DATE

WebSphere Message Broker data type	SQL data type
INTERVAL	Not supported ²
BLOB	SQL_C_BINARY
BIT	Not supported ²
Notes: <ol style="list-style-type: none"> 1. For convenience, DECIMAL values are passed to the DBMS in character form. 2. There is no suitable standard SQL C data type for INTERVAL or BIT. Cast these to another data type, such as CHARACTER, if you need to assign them to a database field. 	

A scalar variable

When assigning to a scalar variable, if the data type of the value being assigned and that of the target variable data type are different, an implicit cast is attempted with the same restrictions and behavior as specified for the explicit CAST function. The only exception is when the data type of the variable is INTERVAL or DECIMAL.

In both these cases, the value being assigned is first cast to a CHARACTER value, and an attempt is made to cast the CHARACTER value to an INTERVAL or DECIMAL. This is because INTERVAL requires a qualifier and DECIMAL requires a precision and scale. These must be specified in the explicit cast, but must be obtained from the character string when implicitly casting. Therefore, a further restriction is that when implicitly casting to an INTERVAL variable, the character string must be of the form INTERVAL '<values>' <qualifier>. The shortened <values> form that is acceptable for the explicit cast is not acceptable here.

Data types of values from external sources

ESQL can extract data from two external sources: message fields and database columns.

The ESQL data type of message fields depends on the type of the message (for example, XML), and the parser that is used to parse it. The ESQL data type of the value returned by a database column reference depends on the data type of the column in the database.

The following table shows which ESQL data types the various built-in database data types are cast to, when they are accessed by message flows that are running in a broker.

The versions that are supported for the database products shown in this table are listed in Supported databases.

Data type	DB2	SQL Server and Sybase	Oracle	Informix®
BOOLEAN		BIT		
INTEGER	SMALLINT, INTEGER, BIGINT	INT, SMALLINT, TINYINT		INT, SMALLINT
FLOAT	REAL, DOUBLE	FLOAT, REAL	NUMBER() ¹	FLOAT, SMALLFLOAT, DOUBLE
DECIMAL	DECIMAL	DECIMAL, NUMERIC, MONEY, SMALLMONEY	NUMBER(P) ¹ , NUMBER(P,S) ¹	DECIMAL, MONEY

Data type	DB2	SQL Server and Sybase	Oracle	Informix®
CHARACTER	CHAR, VARCHAR, CLOB	CHAR, VARCHAR, TEXT	CHAR, VARCHAR2, ROWID, UROWID, LONG, CLOB	CHAR, VARCHAR, CHAR VARYING
TIME	TIME			
GMTTIME				
DATE	DATE			DATE
TIMESTAMP	TIMESTAMP	DATETIME, SMALLDATETIME, TIMESTAMP	DATE	DATETIME
GMTTIMESTAMP				
INTERVAL				INTERVAL
BLOB	BLOB	BINARY, VARBINARY, IMAGE, UNIQUEIDENTIFIER	RAW LONG, RAW BLOB	
BIT				
XML	BLOB			

Note:

1. If an Oracle database column with NUMBER data type is defined with an explicit precision (P) and scale (S), it is cast to an ESQL DECIMAL value; otherwise it is cast to a FLOAT.

For example, an ESQL statement like this:

```
SET OutputRoot.xxx[]
= (SELECT T.department FROM Database.personnel AS T);
```

where Database.personnel resolves to a TINYINT column in an SQL Server database table, results in a list of ESQL INTEGER values being assigned to OutputRoot.xxx.

By contrast, an identical query, where Database.personnel resolves to a NUMBER() column in an Oracle database, results in a list of ESQL FLOAT values being assigned to OutputRoot.xxx.

Miscellaneous ESQL functions

This topic lists the miscellaneous ESQL functions and covers the following:

“COALESCE function”

“EVAL function” on page 372

“NULLIF function” on page 373

“PASSTHRU function” on page 374

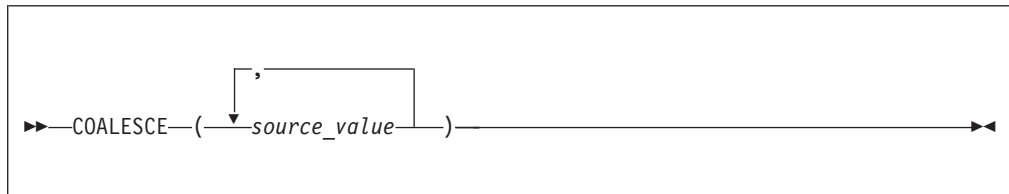
“UIDASBLOB function” on page 376

“UIDASCHAR function” on page 376

COALESCE function

COALESCE is a miscellaneous function that lets you provide default values for fields.

Syntax



The COALESCE function evaluates its parameters in order and returns the first one that is not NULL. The result is NULL if, and only if, all the arguments are NULL. The parameters can be of any scalar type, but they need not all be of the same type.

Use the COALESCE function to provide a default value for a field, which might not exist in a message. For example, the expression:

```
COALESCE(Body.Salary, 0)
```

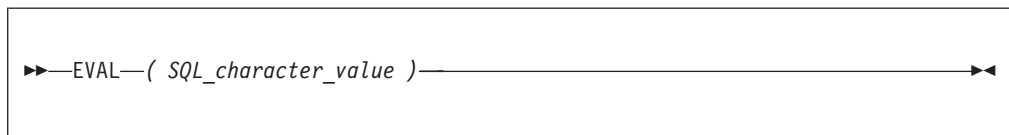
returns the value of the Salary field in the message if it exists, or 0 (zero) if that field does not exist.

EVAL function

The EVAL function takes a character value and interprets that value as an ESQL expression that returns a value.

For details of the EVAL statement, see “EVAL statement” on page 246.

Syntax



EVAL takes one parameter in the form of an expression, evaluates this expression, and casts the resulting value to a character string if it is not one already. The expression that is passed to EVAL must therefore be able to be represented as a character string.

User defined functions cannot be defined within an EVAL function but EVAL can be used to call a user-defined function that is in scope where the EVAL function is used.

If you use the EVAL function to call out to a user-defined function that is not called from anywhere else in the ESQL for a given node, you need to add the following code to your ESQL, to ensure that the user-defined function being called is actually included when the code is compiled:

```
IF (FALSE) THEN CALL function(<parameters>) INTO Environment.temp; END IF;
```

Note, that in the preceding example, you must replace `function()` with the name of the function in question.

In the following examples, A and B are integer scalar variables, and `scalarVar1` and `OperatorAsString` are character string scalar variables.

The following examples are valid uses of EVAL:

- `SET OutputRoot.XMLNS.Data.Result = EVAL(A+B);`
The expression `A+B` is acceptable because, although it returns an integer value, integer values are representable as character strings, and the necessary cast is performed before EVAL continues with its second stage of evaluation.
- `SET OutputRoot.XMLNS.Data.Result = EVAL('A' || operatorAsString || 'B');`
- `EVAL('SET ' || scalarVar1 || ' = 2;');`
The semicolon included at the end of the final string literal is necessary, because if EVAL is being used in place of an ESQL statement, its first stage evaluation must return a string that represents a valid ESQL statement, including the terminating semicolon.

The real power of EVAL is that it allows you to dynamically construct ESQL statements or expressions. In the second and third examples above, the value of `scalarVar1` or `operatorAsString` can be set according to the value of an incoming message field, or other dynamic value, allowing you to effectively control what ESQL is executed without requiring a potentially lengthy IF-THEN ladder.

However, consider the performance implications in using EVAL. Dynamic construction and execution of statements or expressions is necessarily more time-consuming than simply executing pre-constructed ones. If performance is vital, you might prefer to write more specific, but faster, ESQL.

The following are not valid uses of EVAL:

- `SET EVAL(scalarVar1) = 2;`
In this example, EVAL is being used to replace a field reference, not an expression.
- `SET OutputRoot.XMLNS.Data.Result[] = EVAL((SELECT T.x FROM Database.y AS T));`
In this example, the `(SELECT T.x FROM Database.y)` passed to EVAL returns a list, which is not representable as a character string.

The following example is acceptable because `(SELECT T.x FROM Database.y AS T)` is a character string literal, not an expression in itself, and therefore is representable as a character string.

```
SET OutputRoot.XMLNS.Data.Result[]  
= EVAL('(SELECT T.x FROM Database.y AS T)');
```

NULLIF function

NULLIF is a miscellaneous function that returns a NULL value if the arguments are equal.

Syntax

```
►►—NULLIF—(—expression1—,—expression2—)—◄◄
```

The NULLIF function returns a NULL value if the arguments are equal; otherwise, it returns the value of the first argument. The arguments must be comparable. The result of using NULLIF(e1,e2) is the same as using the expression:

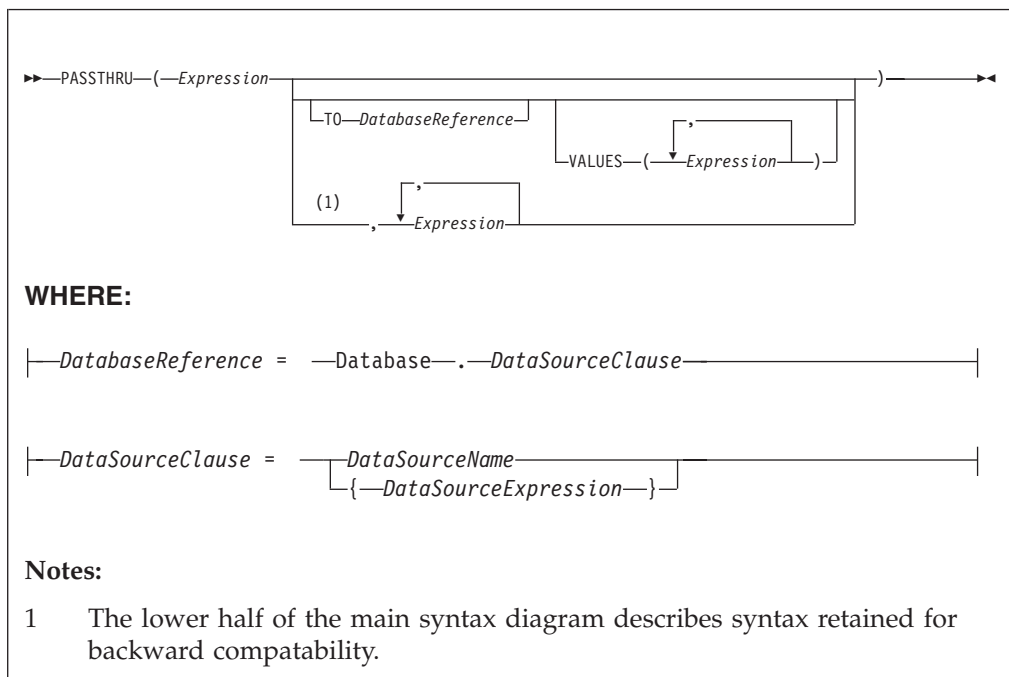
```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

When e1=e2 evaluates to unknown (because one or both of the arguments is NULL), NULLIF returns the value of the first argument.

PASSTHRU function

The PASSTHRU function evaluates an expression and executes the resulting character string as a database statement, returning a result set.

The PASSTHRU function is similar to the PASSTHRU statement, which is described in "PASSTHRU statement" on page 259.



Usage

The main use of the PASSTHRU function is to issue complex SELECTs, not currently supported by the broker, to databases. (Examples of complex SELECTs not currently supported by the broker are those containing GROUP BY or HAVING clauses.)

The first expression is evaluated and the resulting character string is passed to the database pointed to by DatabaseReference (in the TO clause) for execution. If the TO clause is not specified, the database pointed to by the node's data source attribute is used.

Use question marks (?) in the database string to denote parameters. The parameter values are supplied by the VALUES clause.

If the VALUES clause is specified, its expressions are evaluated and passed to the database as parameters; (that is, the expressions' values are substituted for the question marks in the database statement).

If only one VALUE expression exists, the result might or might not be a list. If it is a list, the list's scalar values are substituted sequentially for the question marks. If it is not a list, the single scalar value is substituted for the (single) question mark in the database statement. If more than one VALUE expression exists, none of the expressions evaluate to a list; their scalar values are substituted sequentially for the question marks instead.

Because the database statement is constructed by the user program, it is not essential to use parameter markers (that is, the question marks) or the VALUES clause, because the whole of the database statement could be supplied, as a literal string, by the program. However, use parameter markers whenever possible because this reduces the number of different statements that need to be prepared and stored in the database and the broker.

Database reference

A database reference is a special instance of the field references that is used to refer to message trees. It consists of the word **Database** followed by the name of a data source (that is, the name of a database instance).

You can specify the data source name directly or by an expression enclosed in braces ({...}). A directly-specified data source name is subject to name substitution. That is, if the name used has been declared to be a known name, the value of the declared name is used rather than the name itself (see "DECLARE statement" on page 235).

If you have created a message flow that contains one of the following nodes, and the ESQL that is associated with this node includes a PASSTHRU statement and a database reference, you must specify a value for the Data source property of the relevant node:

- Compute
- Database
- Filter

Handling errors

It is possible for errors to occur during PASSTHRU operations. For example, the database might not be operational or the statement might be invalid. In these cases, an exception is thrown (unless the node has its Throw exception on database error property cleared). These exceptions set appropriate SQL code, state, native error, and error text values and can be dealt with by error handlers (see the DECLARE HANDLER statement).

For further information about handling database errors, see "Capturing database state" on page 84.

Example

The following example performs a SELECT on table Table1 in schema Schema1 in database DSN1, passing two parameters to the WHERE clause and asking for the result set to be ordered in ascending name order. The result set is assigned to the SelectResult folder:

```

SET OutputRoot.XML.Data.SelectResult.Row[] =
  PASSTHRU('SELECT R.* FROM Schema1.Table1 AS R WHERE R.Name = ? OR R.Name =
           ? ORDER BY Name'
  TO Database.DSN1
  VALUES ('Name1', 'Name4'));

```

The above example assigns the result set to the OutputRoot message body tree that is owned by the Generic XML parser, which allows self-defining messages.

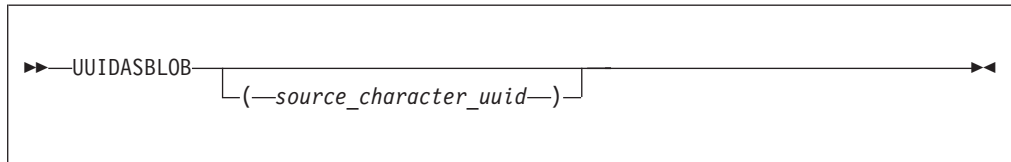
If assigning the result set into a message tree owned by one of the MRM parsers, and the result set structure exactly matches the MRM message definition, the result set can be assigned directly into the OutputRoot message body tree.

If the result set structure does not exactly match the MRM message definition, you must first assign the result set into a ROW data type, or an Environment tree that does not have any parsers associated with it. The required data can then be assigned to OutputRoot to build a message tree that conforms to the message definition.

UIDASBLOB function

UIDASBLOB is a miscellaneous function that returns universally unique identifiers (UUIDs) as BLOBs.

Syntax



If (*source_character_uuid*) is not specified, UIDASBLOB creates a new UUID and returns it as a BLOB.

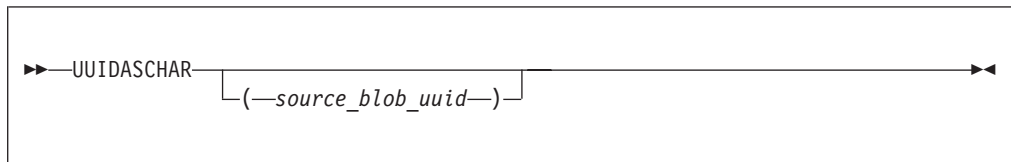
If (*source_character_uuid*) is specified, UIDASBLOB converts an existing character UUID in the form dddddddd_dddd_dddd_dddd_ddddddddddd to the BLOB form. An exception is thrown if the parameter is not of the expected form.

The result is NULL if a NULL parameter is supplied.

UIDASCHAR function

UIDASCHAR is a miscellaneous function that returns universally unique identifiers (UUIDs) as CHARACTER values.

Syntax



If (*source_blob_uuid*) is not specified, UIDASCHAR creates a new UUID and returns it as a CHARACTER value.

If (*source_blob_uuid*) is specified, UUIDASCHAR converts an existing BLOB UUID to the character form.

The result is NULL if a NULL parameter is supplied.

ESQL constants

Use these constants to make or parse a bitstream.

ESQL constants in the Mapping node	Corresponding ESQL constants
\$esql:RootBitStream	RootBitStream
\$esql:FolderBitStream	FolderBitStream
\$esql:ValidateContentAndValue	ValidateContentAndValue
\$esql:ValidateValue	ValidateValue
\$esql:ValidateContent	ValidateContent
\$esql:ValidateNone	ValidateNone
\$esql:ValidateException	ValidateException
\$esql:ValidateExceptionList	ValidateExceptionList
\$esql:ValidateLocalError	ValidateLocalError
\$esql:ValidateUserTrace	ValidateUserTrace
\$esql:ParseComplete	ParseComplete
\$esql:ParseImmediate	ParseImmediate
\$esql:ParseOnDemand	ParseOnDemand

Broker properties that are accessible from ESQL and Java

You can access broker, message flow, and node properties from ESQL and Java.

The following table shows the properties that are available to ESQL and Java code. The fourth column indicates whether the properties are also accessible from Java nodes. If a property is listed as being accessible from Java nodes, it is accessible from Java nodes *only*, *not* from Java routines that are called as ESQL functions or procedures.

For a complete overview of broker properties, see “Broker properties” on page 8.

Property type	Property name	Return type	From Java nodes?	What is it?
General broker properties ¹	BrokerName	Character	Yes ²	The name of the broker.
	BrokerDataSource	Character	No	The ODBC Data Source Name (DSN) of the database that contains the broker's tables.
	BrokerUserId	Character	No	The user ID under which the broker is running (that is, the user ID specified by the -i flag on the <code>mqsicreatebroker</code> command).
	BrokerVersion	Character	No	The 4-character version number of the broker (see "BrokerVersion" on page 380).
	ExecutionGroupLabel	Character	Yes ³	The label of the execution group (a human-readable name).
	ExecutionGroupName	Character	No	The name of the execution group (typically a UUID identifier).
	Family	Character	No	The generic name of the software platform that the broker is running on ('WINDOWS', 'UNIX', or 'ZOS').
	ProcessId	Integer	No	The process identifier (PID) of the execution group.
	QueueManagerName	Character	Yes ⁴	The name of the WebSphere MQ queue manager to which the broker is connected.
	WorkPath	Character	No	(Optional) The directory in which working files for this broker are stored.
Flow properties	AdditionalInstances	Integer	No	The number of additional threads that the broker can use to service the message flow.
	CommitCount	Integer	No	The number of input messages that are processed by the message flow before a syncpoint is taken.
	CommitInterval	Integer	No	The time interval at which a commit is taken when the CommitCount property is greater than 1 (that is, where the message flow is batching messages), but the number of messages processed has not reached the value of the CommitCount property.
	CoordinatedTransaction	Boolean	Yes ⁵	Whether the message flow is processed as a global transaction, coordinated by WebSphere MQ.
	MessageFlowLabel	Character	Yes ⁶	The name of the flow.

Property type	Property name	Return type	From Java nodes?	What is it?
Node properties	DataSource	Character	No	The ODBC Data Source Name (DSN) of the database in which the user tables are created.
	DataSourceUserId	Character	No	The user ID that the broker uses to access the database user tables.
	MessageOptions	Integer (64-bit)	No	The bit stream and validation options in force.
	NodeLabel	Character	Yes ⁷	The name of the node.
	NodeType	Character	No	The type of node (Compute, Database, or Filter).
	ThrowExceptionOnDatabaseError	Boolean	Yes ⁸	Whether the broker generates an exception when a database error is detected.
	TransactionType	Character	Yes ⁹	The type of transaction (Automatic or Commit) used to access a database from this node.
	TreatWarningsAsErrors	Boolean	Yes ¹⁰	Whether database warning messages are treated as errors, and cause the output message to be propagated to the failure terminal.

Notes:

1. The only broker-defined properties that can be used in a Trace node are those in the “General broker properties” group. For example, you could specify the Pattern setting of a Trace node as:

```
#### Start Trace Input Message
Time: ${CURRENT_TIMESTAMP}
Broker: ${BrokerName} Version: ${BrokerVersion} Platform: ${Family}
ProcessID: ${ProcessId} BrokerUserId: ${BrokerUserId}
ExecutionGroupLabel: ${ExecutionGroupLabel}
Transaction: ${Transaction}
Root Tree: ${Root}
#### End Trace Input Message
```

2. Accessible through:
 - a. `MbNode.getBroker()`
 - b. `MbBroker.getName()`
3. Accessible through:
 - a. `MbNode.getExecutionGroup()`
 - b. `MbExecutionGroup.getName()`
4. Accessible through:
 - a. `MbNode.getBroker()`
 - b. `MbBroker.getQueueManagerName()`
5. Accessible through:
 - a. `MbNode.getMessageFlow()`
 - b. `MbMessageFlow.isCoordinatedTransaction()`
6. Accessible through:
 - a. `MbNode.getMessageFlow()`
 - b. `MbMessageFlow.getName()`
7. Accessible through `MbNode.getName()`
8. Accessible through `MbSQL.getThrowExceptionOnDatabaseError()`

9. Accessible through `MbSQL.getTransactionType()`
10. Accessible through `MbSQL.getTreatWarningsAsErrors()`

BrokerVersion

The `BrokerVersion` property contains a 4-character code that indicates the version of the broker. The code is based on the IBM® Version/Release/Modification/Fix pack (VRMF) product-numbering system. The VRMF code works like this:

- V** The Version number. A Version is a separate IBM licensed program that usually has significant new code or new function. Each version has its own license, terms, and conditions.
- R** The Release number. A Release is a distribution of new function and authorized program analysis report (APAR) fixes for an existing product.
- M** The Modification number. A Modification is new function added to an existing product, and is delivered separately from an announced Version or Release.
- F** The Fix pack number. Fix packs contain defect and APAR fixes. They do not contain new function.

A fix pack is cumulative: that is, it contains all the fixes shipped in previous maintenance to the release, including previous fix packs. It can be applied on top of any previously-shipped maintenance to bring the system up to the current fix pack level.

Special characters, case sensitivity, and comments in ESQL

When you work with ESQL, it is useful to know about the special characters that are available, whether ESQL syntax is case sensitive, and how to handle comments.

- “Special characters”
- “Case sensitivity of ESQL syntax” on page 381
- “Comments” on page 381

Special characters

Symbol	Name	Usage
;	semicolon	End of ESQL statement
.	period	Field reference separator or decimal point
=	equals	Comparison or assignment
>	greater than	Comparison
<	less than	Comparison
[]	square brackets	Array subscript
'	single quotation mark	Delimit string, date-time, and decimal literals Note, that to escape a single quotation mark inside a string literal, you must use two single quotation marks.
	double vertical bar	Concatenation
()	parentheses	Expression delimiter
"	quotation mark	Identifier delimiter
*	asterisk	Any name or multiply

Symbol	Name	Usage
+	plus	Arithmetic add
-	minus	Arithmetic subtract, date separator, or negation
/	forward slash	Arithmetic divide
_	underscore	LIKE single wildcard
%	percent	LIKE multiple wildcard
\	backslash	LIKE escape character
:	colon	Name space and Time literal separator
,	comma	List separator
<>	less than greater than	Not equals
--	double minus	ESQL single line comment
/* */	slash asterisk asterisk slash	ESQL multiline comment
?	question mark	Substitution variable in PASSTHRU
<=	less than or equal	Comparison
>=	greater than or equal	Comparison
/*!{ }!*/	executable comment	Bypass tools check

Case sensitivity of ESQL syntax

The case of ESQL statements is:

- Case sensitive in field reference literals
- Not case sensitive in ESQL language words

Comments

ESQL has two types of comment: single line and multiple line. A single line comment starts with the characters -- and ends at the end of the line.

In arithmetic expressions you must take care not to initiate a line comment accidentally. For example, consider the expression:

```
1 - -2
```

Removing all white space from the expression results in:

```
1--2
```

which is interpreted as the number 1, followed by a line comment.

A multiple line comment starts with /* anywhere in ESQL and ends with */.

ESQL reserved keywords

The following keywords are reserved in uppercase, lowercase, or mixed case. You cannot use these keywords for variable names. However, you *can* use reserved keywords as names in a field reference.

ALL	ASYMMETRIC	BOTH
CASE	DISTINCT	FROM

ITEM	LEADING	NOT
SYMMETRIC	TRAILING	WHEN

ESQL non-reserved keywords

Some keywords are used in the ESQL language, but are not reserved. Do not use them for variable, function, or procedure names (in any combination of uppercase and lowercase) because your code can become difficult to understand.

The following keywords are used in the ESQL language, but are not reserved.

- AND
- ANY
- AS
- ATOMIC
- ATTACH
- BEGIN
- BETWEEN
- BIT
- BLOB
- BOOLEAN
- BY
- CALL
- CATALOG
- CCSID
- CHAR
- CHARACTER
- COMPUTE
- CONDITION
- CONSTANT
- CONTINUE
- COORDINATED
- COUNT
- CREATE
- CURRENT_DATE
- CURRENT_GMTDATE
- CURRENT_GMTTIME
- CURRENT_GMTTIMESTAMP
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATA
- DATABASE
- DATE
- DAY
- DAYOFWEEK
- DAYOFYEAR
- DAYS
- DECIMAL
- DECLARE
- DEFAULT
- DELETE
- DETACH
- DO
- DOMAIN
- DYNAMIC

- ELSE
- ELSEIF
- ENCODING
- END
- ENVIRONMENT
- ESCAPE
- ESQL
- EVAL
- EVENT
- EXCEPTION
- EXISTS
- EXIT
- EXTERNAL
- FALSE
- FIELD
- FILTER
- FINALIZE
- FIRSTCHILD
- FLOAT
- FOR
- FORMAT
- FOUND
- FULL
- FUNCTION
- GMTTIME
- GMTTIMESTAMP
- GROUP
- HANDLER
- HAVING
- HOUR
- IDENTITY
- IF
- IN
- INF
- INFINITY
- INOUT
- INSERT
- INT
- INTEGER
- INTERVAL
- INTO
- IS
- ISLEAPYEAR
- ITERATE
- JAVA
- LABEL
- LANGUAGE
- LAST
- LASTCHILD
- LEAVE
- LIKE
- LIST
- LOCALTIMEZONE
- LOG
- LOOP
- MAX

- MESSAGE
- MIN
- MINUTE
- MODIFIES
- MODULE
- MONTH
- MONTHS
- MOVE
- NAME
- NAMESPACE
- NAN
- NEXTSIBLING
- NONE
- NULL
- NUM
- NUMBER
- OF
- OPTIONS
- OR
- ORDER
- OUT
- PARSE
- PASSTHRU
- PATH
- PLACING
- PREVIOUSIBLING
- PROCEDURE
- PROPAGATE
- QUARTEROFYEAR
- QUARTERS
- READS
- REFERENCE
- REPEAT
- RESIGNAL
- RESULT
- RETURN
- RETURNS
- ROW
- SAMEFIELD
- SCHEMA
- SECOND
- SELECT
- SET
- SETS
- SEVERITY
- SHARED
- SHORT
- SOME
- SQL
- SQLCODE
- SQLERRORTXT
- SQLEXCEPTION
- SQLNATIVEERROR
- SQLSTATE
- SQLWARNING
- SUM

- TERMINAL
- THE
- THEN
- THROW
- TIME
- TIMESTAMP
- TO
- TRACE
- TRUE
- TYPE
- UNCOORDINATED
- UNKNOWN
- UNTIL
- UPDATE
- USER
- UUIDASBLOB
- UUIDASCHAR
- VALUE
- VALUES
- WEEKOFMONTH
- WEEKOFYEAR
- WEEKS
- WHERE
- WHILE
- YEAR

Example message

This topic defines the example message that is used in many of the examples throughout the information center.

The example message is:

```

<Invoice>
<InvoiceNo>300524</InvoiceNo>
<InvoiceDate>2000-12-07</InvoiceDate>
<InvoiceTime>12:40:00</InvoiceTime>
<TillNumber>3</TillNumber>
<Cashier StaffNo="089">Mary</Cashier>
<Customer>
  <FirstName>Andrew</FirstName>
  <LastName>Smith</LastName>
  <Title>Mr</Title>
  <DOB>20-01-70</DOB>
  <PhoneHome>01962818000</PhoneHome>
  <PhoneWork/>
  <Billing>
    <Address>14 High Street</Address>
    <Address>Hursley Village</Address>
    <Address>Hampshire</Address>
    <PostCode>S0213JR</PostCode>
  </Billing>
</Customer>
<Payment>
  <CardType>Visa</CardType>
  <CardNo>4921682832258418</CardNo>
  <CardName>Mr Andrew J. Smith</CardName>
  <Valid>1200</Valid>
  <Expires>1101</Expires>
</Payment>
<Purchases>

```

```

    <Item>
      <Title Category="Computer" Form="Paperback" Edition="2">The XML Companion
</Title>
      <ISBN>0201674866</ISBN>
      <Author>Neil Bradley</Author>
      <Publisher>Addison-Wesley</Publisher>
      <PublishDate>October 1999</PublishDate>
      <UnitPrice>27.95</UnitPrice>
      <Quantity>2</Quantity>
    </Item>
    <Item>
      <Title Category="Computer" Form="Paperback" Edition="2">A Complete Guide
to DB2 Universal Database</Title>
      <ISBN>1558604820</ISBN>
      <Author>Don Chamberlin</Author>
      <Publisher>Morgan Kaufmann Publishers</Publisher>
      <PublishDate>April 1998</PublishDate>
      <UnitPrice>42.95</UnitPrice>
      <Quantity>1</Quantity>
    </Item>
    <Item>
      <Title Category="Computer" Form="Hardcover" Edition="0">JAVA 2 Developers
Handbook</Title>
      <ISBN>0782121799</ISBN>
      <Author>Philip Heller, Simon Roberts </Author>
      <Publisher>Sybex, Inc.</Publisher>
      <PublishDate>September 1998</PublishDate>
      <UnitPrice>59.99</UnitPrice>
      <Quantity>1</Quantity>
    </Item>
  </Purchases>
<StoreRecords/>
<DirectMail/>
<Error/>
</Invoice>

```

For a diagrammatic representation of this message, and for examples of how this message can be manipulated with ESQL statements and functions, refer to “Writing ESQL” on page 25.

Part 3. Appendixes

Appendix. Notices for WebSphere Message Broker

Read the legal notices for WebSphere Message Broker.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.*

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan*

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information includes examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) *(your company name)* (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks in the WebSphere Message Broker Information Center

Review the trademark information for WebSphere Message Broker.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks of Intel Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

C

- code pages
 - converting with ESQL 61

D

- data types
 - support for 370
- databases
 - data type support 370
 - stored procedures in ESQL 77
- Destination (LocalEnvironment),
populating 56

E

- Environment tree
 - accessing with ESQL 58
- ESQL
 - accessible from Java 8
 - adding keywords 67
 - BLOB messages 152
 - Broker attributes 8
 - constants 377
 - converting EBCDIC NL to ASCII
CRLF 63
 - data
 - casting 60
 - converting 61
 - transforming 60
 - data types 4
 - database columns
 - referencing 68
 - selecting data from 70
 - database content, changing 75
 - database state 84
 - database updates, committing 77
 - databases, interacting with 68
 - datetime representation 164
 - Destination, populating 56
 - developing 3
 - elements
 - accessing 29
 - setting or querying null 29
 - elements, multiple occurrences
 - accessing known 33
 - accessing unknown 34
 - Environment tree, accessing 58
 - errors 79
 - example message 385
 - ExceptionList tree, accessing 58
 - exceptions 83
 - explicit null handling 29
 - field references 9
 - anonymous 35
 - creating 36
 - syntax 175
 - field types, referencing 28
 - fields
 - copying those that repeat 43

- ESQL (*continued*)
 - fields (*continued*)
 - creating new 37
 - manipulating those that repeat in a
message tree 47
 - files
 - copying 21
 - creating 14
 - deleting 25
 - moving 22
 - opening 15
 - renaming 22
 - saving 20
 - functions 11
 - headers, accessing 48
 - IDoc messages 149
 - implicit null handling 29
 - JMS messages 148
 - keywords
 - non-reserved 382
 - reserved 381
 - like-parser-copy 65
 - list type elements, working with 45
 - LocalEnvironment tree, accessing 54
 - mapping between a list and a
repeating element 46
 - message body data, manipulating 28
 - message format, changing 65
 - message tree parts, manipulating 48
 - MIME messages 150
 - modules 13
 - MQCFH header, accessing 50
 - MQMD header, accessing 49
 - MQPCF header, accessing 50
 - MQRFH2 header, accessing 49
 - MRM domain messages
 - handling large 145
 - working with 142
 - MRM domain messages, accessing
 - attributes 135
 - elements 133
 - elements in groups 136
 - embedded messages 139
 - mixed content 138
 - multiple occurrences 134
 - namespace-enabled messages 140
 - MRM domain messages, null values
 - querying 141
 - setting 142
 - multiple database tables,
accessing 74
 - nested statements 10
 - node
 - creating 16
 - deleting 24
 - modifying 19
 - numeric operators with datetime 40
 - operators 9
 - complex comparison 183
 - logical 185
 - numeric 186

- ESQL (*continued*)
 - operators (*continued*)
 - rules for operator precedence 187
 - simple comparison 181
 - string 187
 - output messages, generating 39
 - overview of 159
 - preferences, changing 23
 - procedures 12
 - Properties tree, accessing 52
 - returns to SELECT, checking 76
 - SELECT function 85
 - settings
 - editor 23
 - validation 24
 - special characters 380
 - statements 10
 - stored procedures, invoking 77
 - subfield, selecting 42
 - syntax preference 160
 - tailoring for different nodes 27
 - time interval, calculating 41
 - unlike-parser-copy 65
 - variables 4
 - XML domain, manipulating messages
in the 132
 - XML messages
 - complex message,
transforming 89
 - data, translating 95
 - message and table data,
joining 96
 - message data, joining 93
 - scalar value, returning 91
 - simple message, transforming 86
 - XMLNS domain, manipulating
messages in the 122
 - XMLNSC domain, manipulating
messages in the 109
- ESQL data types
 - BOOLEAN 160
 - database, ROW 169
 - Datetime 160
 - DATE 161
 - GMTTIME 161
 - GMTTIMESTAMP 162
 - INTERVAL 162
 - TIME 161
 - TIMESTAMP 161
 - ESQL to Java, mapping of 171
 - ESQL to XPath, mapping of 172
 - list of 160
 - NULL 166
 - numeric 166
 - DECIMAL 167
 - FLOAT 168
 - INTEGER 168
 - REFERENCE 169
 - string 170
 - BIT 170
 - BLOB 170

- ESQL data types (*continued*)
 - string (*continued*)
 - CHARACTER 171
- ESQL functions 275
 - CAST
 - formatting and parsing dates as strings 339
 - formatting and parsing numbers as strings 336
 - formatting and parsing times as strings 339
 - complex 329
 - CASE 330
 - CAST 331
 - data types from external sources 370
 - LIST constructor 353
 - ROW and LIST combined 354
 - ROW and LIST comparisons 354
 - ROW constructor 352
 - SELECT 346
 - Supported casts 357
 - database state 279
 - SQLCODE 279
 - SQLERRORTXT 279
 - SQLNATIVEERROR 280
 - SQLSTATE 281
 - datetime 283
 - CURRENT_DATE 286
 - CURRENT_GMTDATE 287
 - CURRENT_GMTTIME 287
 - CURRENT_GMTTIMESTAMP 287
 - CURRENT_TIME 286
 - CURRENT_TIMESTAMP 286
 - EXTRACT 284
 - LOCAL_TIMEZONE 288
 - field 316
 - ASBITSTREAM 316
 - BITSTREAM 319
 - FIELDNAME 320
 - FIELDNAMESPACE 321
 - FIELDTYPE 321
 - FIELDVALUE 323
 - FOR 324
 - LASTMOVE 326
 - SAMEFIELD 326
 - implicit casts 365
 - arithmetic operations 367
 - assignment 369
 - comparisons 365
 - list 326
 - CARDINALITY 327
 - EXISTS 327
 - SINGULAR 328
 - THE 329
 - miscellaneous 371
 - COALESCE 371
 - EVAL 372
 - NULLIF 373
 - PASSTHRU 374
 - UUIDASBLOB 376
 - UUIDASCHAR 376
 - numeric 288
 - ABS and ABSVAL 289
 - ACOS 290
 - ASIN 290
 - ATAN 290
- ESQL functions (*continued*)
 - numeric (*continued*)
 - ATAN2 290
 - BITAND 291
 - BITNOT 291
 - BITOR 292
 - BITXOR 292
 - CEIL and CEILING 293
 - COS 293
 - COSH 294
 - COT 294
 - DEGREES 294
 - EXP 294
 - FLOOR 295
 - LN and LOG 295
 - LOG10 296
 - MOD 296
 - POWER 296
 - RADIANS 297
 - RAND 297
 - ROUND 297
 - SIGN 301
 - SIN 301
 - SINH 301
 - SQRT 301
 - TAN 302
 - TANH 302
 - TRUNCATE 303
 - string manipulation 303
 - CONTAINS 304
 - ENDSWITH 305
 - LEFT 305
 - LENGTH 305
 - LOWER and LCASE 306
 - LTRIM 306
 - OVERLAY 307
 - POSITION 308
 - REPLACE 309
 - REPLICATE 309
 - RIGHT 310
 - RTRIM 310
 - SPACE 311
 - STARTSWITH 311
 - SUBSTRING 312
 - TRANSLATE 313
 - TRIM 314
 - UPPER and UCASE 315
- ESQL statements (*continued*)
 - ESQL statements 188
 - ATTACH 189
 - BEGIN ... END 191
 - BROKER SCHEMA 194
 - PATH clause 194
 - CALL 197
 - CASE 199
 - CREATE 202
 - CREATE FUNCTION 209
 - CREATE MODULE 218
 - CREATE PROCEDURE 220
 - DECLARE 235
 - DECLARE HANDLER 240
 - DELETE 245
 - DELETE FROM 243
 - DETACH 246
 - EVAL 246
 - FOR 248
 - IF 248
 - INSERT 250
 - ESQL statements (*continued*)
 - ITERATE 253
 - LEAVE 254
 - list of available 188
 - Local error handler 240
 - LOG 255
 - LOOP 256
 - MOVE 257
 - PASSTHRU 259
 - PROPAGATE 261
 - REPEAT 265
 - RESIGNAL 265
 - RETURN 266
 - SET 268
 - THROW 271
 - UPDATE 272
 - WHILE 275
 - ExceptionList tree
 - accessing with ESQL 58
- H**
 - headers
 - accessing 48
- J**
 - Java, broker attributes accessible from 8
- K**
 - keywords
 - ESQL 67
- L**
 - list of available 275
 - LocalEnvironment tree
 - accessing with ESQL 54
 - populating Destination 56
 - using as scratchpad 55
- M**
 - message body
 - ESQL, accessing with 28
 - message flows
 - creating ESQL code 16
 - customizing nodes with ESQL 25
 - ESQL 3
 - managing ESQL files 13
 - MQCFH header
 - accessing with ESQL 50
 - MQMD (message descriptor)
 - accessing with ESQL 49
 - MQPCF header
 - accessing with ESQL 50
 - MQRFH2 header
 - accessing with ESQL 49
- P**
 - Properties tree, accessing with ESQL 52

T

trademarks 391

X

XPath property editors 173



Printed in USA