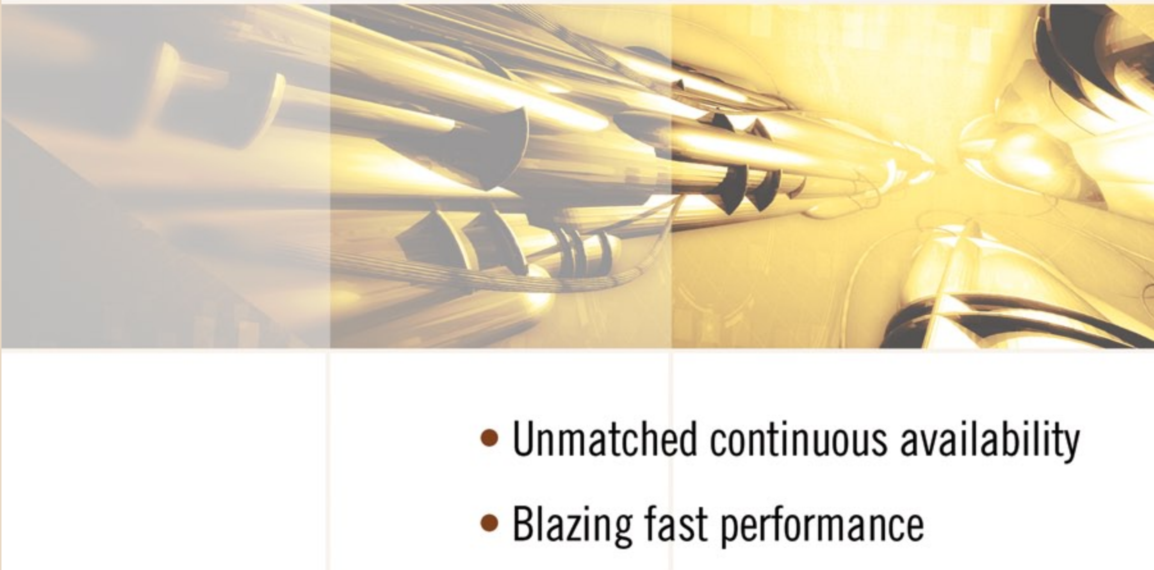
IBM Informix Dynamic Server 11

The Next Generation in OLTP Data Server Technology



- Unmatched continuous availability
- Blazing fast performance
- Legendary reliability
- Virtually zero administration

CARLTON DOE

IBM Informix Dynamic Server 11

The Next Generation in OLTP Data Server Technology

Carlton Doe

MC PRESS

MC Press Online, LP

Lewisville, TX 75077

IBM Informix Dynamic Server 11
The Next Generation in OLTP Data Server Technology
Carlton Doe

First Edition
First Printing—June 2007

Every attempt has been made to provide correct information. However, IBM, the publisher, nor the author guarantee the accuracy of the book and do not assume responsibility for information included in or omitted from it.

MC Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include custom covers and content particular to your business, training goals, marketing focus, and branding interest.

Product feature and technology names mentioned herein are as they existed at the time the book was written. IBM reserves the right to modify and change them prior to the general release of the product.

For information regarding permissions or special orders, please contact:

MC Press
Corporate Offices
125 N. Woodland Trail
Lewisville, TX 75077 USA

For information regarding sales and/or customer service, please contact:

MC Press
P.O. Box 4300
Big Sandy, TX 75755-4300 USA

ISBN: 1-58347-075-1

To Catherine and my children,
Thank you for your continuing support

To the IDS development teams,
I couldn’t be more proud to represent your
work in IDS 11. Well, well done.

Acknowledgments

A book like this doesn't get created out of thin air. Many people had a hand in bringing it to you. At the risk of missing someone, here they are:

- The IDS development teams for creating the technology, letting me crawl through the internal specifications and design documentation, responding to my questions, and making sure I had continuing access to the latest product builds to test and play with.
- Jerry Keesee, Kevin Brown, Jonathan Leffler, Alan Caldera, Keshava Murthy, Mahesh Dashora, David Desautels, Guy Bowerman, Joseph Veleeparambil, Karen Qualley, Robert Uleman, Inge Halilovic, and Hon Cheong for their technical reviews to help keep me out of the technical tar pits.
- Susan Visser, Cathy Elliott, and Pat Moffatt for getting the IBM machinery to publish the book.
- Katie Tipton for catching and correcting all the text warts in what I wrote.
- Merrikay Lee for putting up with all the hassles and working so hard to get this published on time.

To anyone I've missed, I'm sorry but thank you very much for your help and support.

Contents

1

Introduction to Informix Dynamic Server

There's no denying the similarity between the watchwords for businesses today and the Olympic motto, *Citius, Altius, Fortius* — that is, "Faster, Higher, Stronger." Companies today are under pressure to scope, design, and bring products and services to market before their competition. Marketing messages must create an immediate impact to drive demand, and costs and inefficiencies must be reduced to the bare minimum. All-out commitment to the business or product strategy is expected — unless the strategy doesn't appear to be working; in that case, the business (and its employees) must immediately change course, adopt a new strategy, and push forward with the new approach.

IBM Informix® Dynamic Server™ 11 (IDS) — code-named "Cheetah" — continues a longstanding tradition within IBM and Informix of delivering first-in-class data servers. IDS combines the robustness, high performance, availability, and scalability that today's modern businesses need.

Complex, mission-critical data management applications typically require a combination of online transaction processing (OLTP), batch, and decision-support operations, including online analytical processing (OLAP). Meeting these needs demands a data server that can scale in performance as well as in functionality. It must dynamically adjust to changing requirements — the accumulation of larger amounts of data, changes in SQL operations, or increasing numbers of concurrent users. The technology should be designed to efficiently use all the capabilities of the existing hardware and software configuration, including single- and multiprocessor architectures. Last, the data server must satisfy users' demands for more complex application support, which often uses nontraditional or "rich" data types that can't be stored in simple character or numeric form.

Built on the IBM Informix Dynamic Scalable Architecture™ (DSA), IDS provides one of the most effective solutions available: a next-generation parallel database architecture that delivers mainframe-caliber scalability, manageability, and performance along with minimal operating system overhead, automatic distribution of workload, and the capability to extend the server to handle new types of data. With Version 11, IDS increases its lead over the data server landscape with even faster performance, major reductions and improvements to an already practically hands-free self-administrating data server, significant enhancements to its unmatched business-continuity functionality, XML support, expansion of its geospatial capabilities, new application development, integration features, and more.

IDS delivers proven technology that efficiently integrates new and complex data directly into the database so that data can be modeled as the business uses it. With these expanded modeling options, data becomes information.

IDS handles time-series, spatial, geodetic, Extensible Markup Language (XML), video, image, and other user-defined data side by side with traditional legacy data to meet today's most rigorous data and business demands. IDS helps you and your business lower its total cost of ownership (TCO) by leveraging IDS's well-regarded general ease of use and administration as well as its support of existing standards for development tools and systems infrastructure. A development-neutral environment, IDS supports a comprehensive array of application development tools for rapid deployment of applications under Linux, Microsoft® Windows®, and UNIX® operating environments.

The maturity and success of IDS is built on more than 14 years of widespread use in critical business operations, a track record that attests to IDS's stability, performance, and usability. IDS 11 moves this already highly successful enterprise relational database to a new level.

This book briefly introduces the technological architecture that supports all versions of IDS and then describes in greater detail some of the new features available in the latest release. IDS begins with many features and capabilities that are unique in the industry and unavailable in any other data server. With Version 11, IDS continues to maintain and accelerate its lead over other data servers on the market today, enabling customers to use information in new and more efficient ways to create business advantages.

Overview of the Informix Dynamic Scalable Architecture

High system performance is essential for maintaining maximum throughput. IBM Informix Dynamic Server maintains industry-leading performance levels through multiprocessor features, shared memory management, efficient data access, and cost-based query optimization. IDS is available on the broadest set of hardware platforms of all servers in the IBM Information Management family. Because the underlying platform is transparent to applications, the data server can migrate easily to more powerful computing environments as needs change. This transparency enables your developers to take advantage of high-end symmetric multiprocessing (SMP) systems with little or no need to modify application code.

Data server architecture is a significant differentiator and contributor to performance, scalability, and the ability to support new data types and processing requirements. Almost all data servers available today use an older technological design that requires each database operation for an individual user (read, sort, write, communication, and so on) to invoke a separate operating-system process. This architecture worked well when database sizes and user counts were relatively small. Today, these types of servers spawn many hundreds and even thousands of individual processes that the operating system must create, queue, schedule, manage/control, and then terminate when they're no longer needed. Given the fact that, generally speaking, any individual system CPU can work on only one thing at a time — and the operating system works through each of the processes before returning to the top of the queue — this data server architecture creates an environment in which individual database operations must wait for one or more passes through the queue to complete their task. With this type of architecture, scalability has nothing to do with the software; it depends entirely on the speed of the processor — how fast it can work through the queue before it starts over again.

IDS's architecture is based on advanced technology that efficiently uses virtually all of today's hardware and software resources. Called the Dynamic Scalable Architecture, it fully exploits the processing power available in SMP environments by performing similar types of database activities (e.g., I/O, complex queries, index builds, log recovery, inserts, and backups/restores) in parallelized groups rather than as discrete operations. The DSA design architecture includes built-in light-weight multithreading and parallel processing capabilities, dynamic and self-tuning shared memory components, and intelligent logical data storage capabilities, supporting the most efficient use of all available system resources.

In effect, what happens is that the DSA creates a separate virtual database operating system that leverages the resources of the physical server. Because this system was written with a single focus, it is highly optimized and efficient in its execution of data server operations. This optimization translates to reduced hardware and administrative expenses, enabling you to do more with less. It's not uncommon for customers like you to reduce their planned hardware expenditures when deciding to move to IDS. In some cases, customers buy no new hardware at all; they can continue to use existing systems because IDS provides more than acceptable performance even on "older, less powerful" systems. The following sections discuss each component of DSA.

DSA Components: Processor

IDS provides the unique ability to scale the database system by using a dynamically configurable pool of database server processes called virtual processors (VPs). Each database operation, such as a sorted data query, is broken into task-oriented subtasks (data read, join, group, and sort, for example) for rapid processing by virtual processors that specialize in that type of subtask. Virtual processors mimic the functionality of the hardware CPUs in that they schedule and manage user requests using multiple, concurrent threads. Figure 1.1 illustrates this functionality.

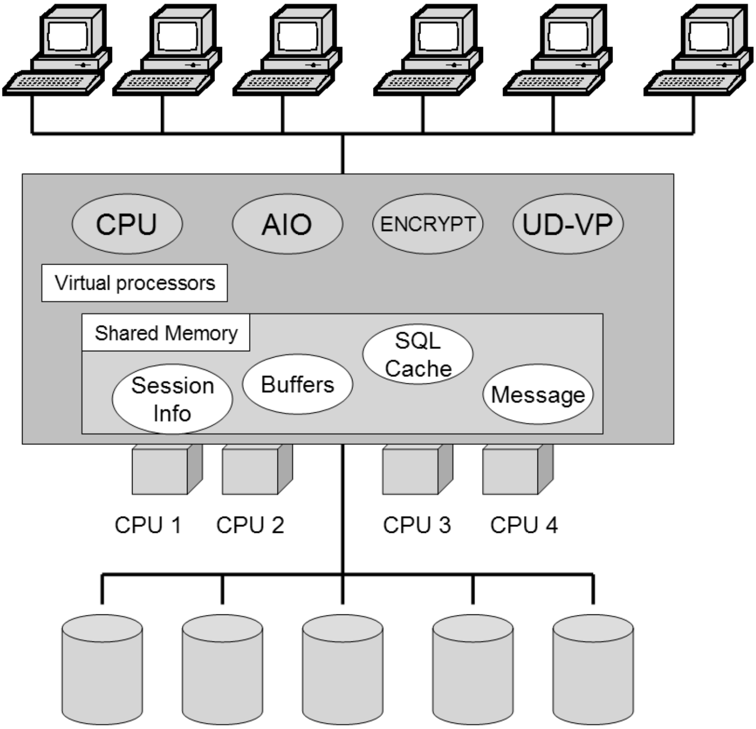


Figure 1.1: IDS has a pool of virtual processors (VPs) that divide operations into discrete tasks and execute them in parallel.

A thread represents a discrete task within a data server process, and many threads may execute simultaneously, and in parallel, across the pool of virtual processors. Unlike a CPU process-based (or single-threaded) engine, which leaves tasks on the system CPU for its given unit of time (even if no work can be done, thus wasting processing time), virtual processors are light-weight and multithreaded. As a consequence, when a thread is either waiting for a resource or has completed its task, a thread switch will occur and the virtual processor will immediately work on another thread. As a result, not only is precious CPU saved but it is also used to satisfy as many user requests as possible in the given amount of time. Figure 1.2 illustrates this type of processing, known as fan-in parallelism.

Not only can one virtual processor respond to multiple user requests in any given unit of time, but one user request can also be distributed across multiple virtual processors. For example, with a processing-intensive request such as a multi-table join, the data server divides the task into multiple subtasks and then spreads these subtasks across all available virtual processors. With the ability to distribute tasks, the request is completed more quickly. Figure 1.3 illustrates this type of processing, referred to as fan-out parallelism.

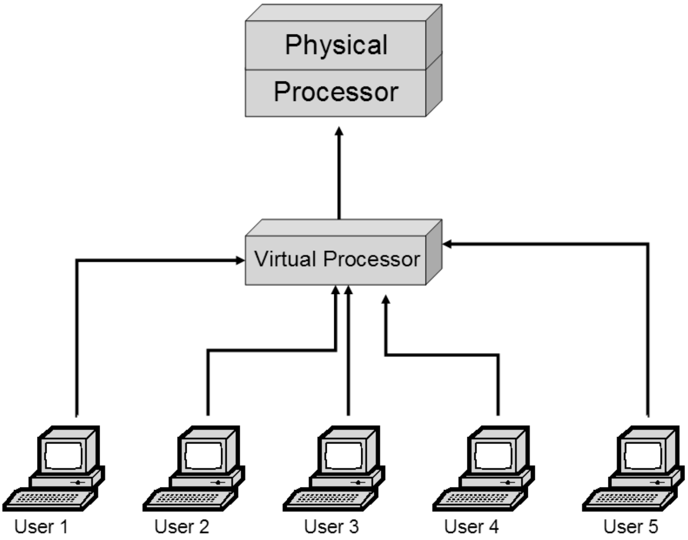


Figure 1.2: IDS VPs employ fan-in parallelism to efficiently use hardware resources to complete more operations in shorter amounts of time.

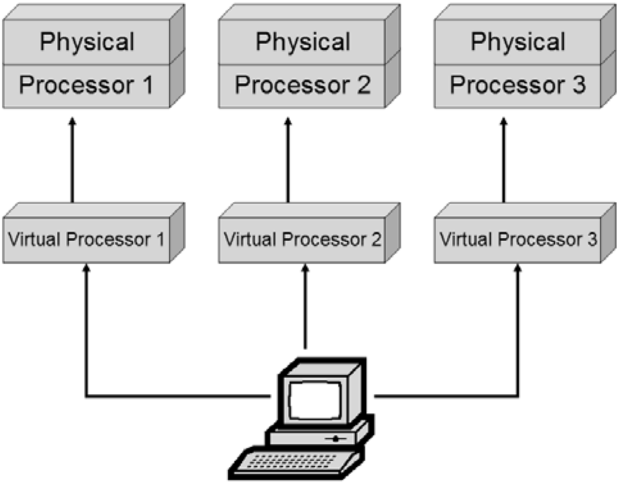


Figure 1.3: Fan-out parallelism uses many VPs to process a single SQL operation.

Together with fan-in parallelism, the net effect is more work being accomplished more quickly than with single-threaded architectures; in other words, the data server is faster.

Dynamic load balancing occurs within IDS because threads aren't statically assigned to virtual processors. The first available VP services outstanding requests, balancing the workload across all available resources. For efficient execution and versatile tuning, VPs can be grouped into classes, each optimized for a particular function, such as CPU operations, disk I/O, communications, and administrative tasks. Figure 1.4 illustrates this type of configuration. You can configure the IDS instance with the appropriate number of virtual processors in each class to handle the workload. You can make adjustments, if necessary, while the instance is online without interrupting database operations to handle occasional periods of heavy activity or different load mixes.

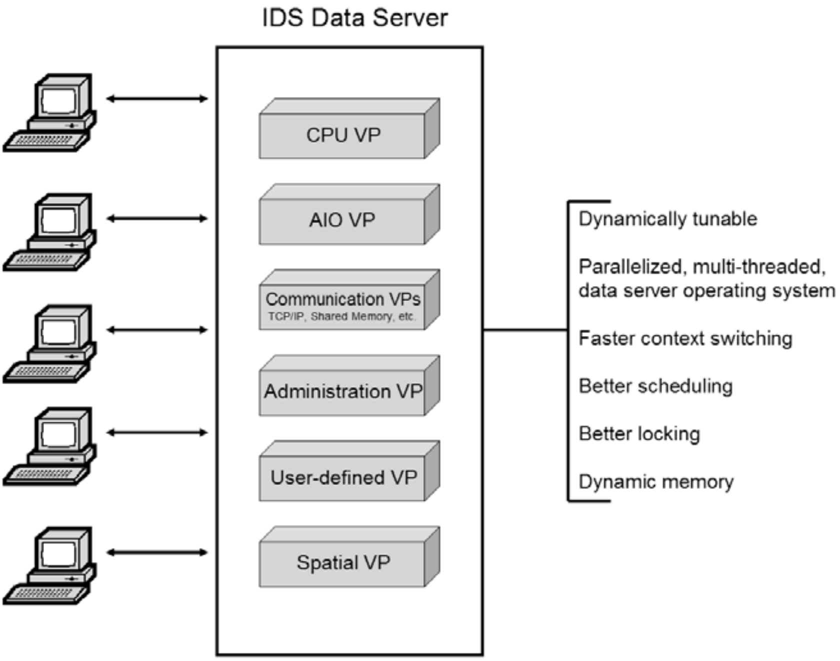


Figure 1.4: VPs are grouped into classes optimized for specific functions. An IDS instance is configured with the number of VPs in each class to handle the expected workload.

In Linux and Unix systems, the use of multithreaded VPs significantly reduces the number of Linux/Unix processes, and, as a result, less context switching is required. In Windows systems, VPs are implemented as threads to take advantage of the operating system's inherent multithreading capability. Because IDS includes its own threading capability for servicing client requests, the actual number of Windows threads is decreased — reducing the system thread-scheduling overhead and providing better throughput.

That IDS employs a light-weight threading model is significant. As I mentioned earlier, the DSA creates a type of data server operating system. As the sole owner-operator, the DSA controls all aspects of the data server's operations, including when thread switches occur and how memory is used. This is not the case with heavy-weight threaded applications built using Posix or other threading libraries. Although Posix provides a fairly easy facility to use to thread an application, Posix threads operate under the constraints and direction of the physical server's operating system. As a result, the operating system regularly preempts these threads based on what it thinks is excessive processing time and other conditions. In addition, these threads may be further constrained by having to use O/S interfaces to devices as opposed to the direct device support built into IDS. The net effect is that, although better than a single-threaded environment, a heavy-weight threading model is not as efficient or powerful as the bespoke libraries for IDS.

In fully utilizing the hardware processing cycles, the IDS data server requires less hardware power to achieve performance comparable to or better than other data servers. In fact, real-world tests and customer experiences indicate that IDS needs only 35 percent to 50 percent of the hardware resources to meet or exceed the performance characteristics of single-threaded or process-based database servers. With this kind of efficiency, you can extend the life of your existing systems, defer costly expenditures, avoid migration-oriented outages and work, and, most important, contribute directly to the bottom line of your company's financials.

DSA Components: Dynamic Shared Memory

All memory used by IDS is shared among the pool of virtual processors. Beyond a small initial allocation of memory for instance-level management, usually a single shared memory portion is created and used by the VPs for all data operations. This portion contains the buffers of queried and modified data; sort, join, and group tables; lock pointers; and so on. What is unique to IDS is that should database operations require more (or less) shared memory, additional segments will be dynamically added to (or dropped from) the portion without interrupting user activities. You can also make similar changes manually while the instance is running to prepare for an imminent processing spike. When a user session terminates its connection to the data server, the thread-specific memory for that session is freed within the memory portion and is reused by another session.

The buffer pool is used to hold data from the data server disk supply during processing. When users request data, the data server first tries to locate the data in the buffer pool to avoid unnecessary disk I/O. Depending on the characteristics of the instance workload, increasing the size of the buffer pool can significantly reduce the number of disk accesses, which can substantially improve performance, particularly for OLTP applications.

The buffer pool holds frequently used table or index data using a scorecard system. As each element is used, its score increases. A portion of the buffer system holds these high-score elements, while the remainder holds less frequently used data. This segmentation of high- and low-use data is completely transparent to the application; it receives in-memory response times regardless of which portion of the buffer pool contains the requested element. As data elements are used less often, they are migrated from the high-use to the low-use portion. Data buffers in this area are flushed and reused via a first-in, first-out (FIFO) process.

Included in the memory structures for an instance are cached disk-access plans for the IDS cost-based optimizer. In most OLTP environments, the same SQL operations are executed throughout the processing day, albeit with slightly different variable conditions (such as “customer number,” “invoice number,” and so on). Each time an SQL operation is executed, the data server optimizer must determine the fastest way to access the data. Obviously, if the data is already cached in the memory buffers, it is retrieved from there; otherwise, disk access is required. When this occurs, the optimizer has to decide on the quickest way to get the requested data. It must evaluate whether an index exists pointing directly to the requested data or whether the data has been intelligently fragmented on disk, restricting the possible number of dbspaces to look through (we’ll talk more about dbspaces in a moment). When joining data from several tables, the optimizer evaluates which table will provide the data the others will join to, and so on. Although not really noticeable to end users, these tasks take time to execute and affect response time.

IDS provides a caching mechanism whereby data I/O plans can be stored for reuse by subsequent executions of the same operation. Called, appropriately enough, the SQL Statement Cache, this allocation of instance memory stores the SQL statement and the optimizer’s determination of the fastest way to execute the operation. You can configure the size of this cache as well as when an individual SQL operation is cached. Generally speaking, most administrators choose to cache operations after they’ve been executed three or more times to prevent filling the cache with single-use operations. You can flush the cache so it is refreshed on an as-needed basis without interrupting transaction processing.

With dynamic reconfiguration of memory allocations, intelligent buffer management, caching of SQL access plans, and a number of other technologies, Informix Dynamic Server provides unmatched efficiency and scalability of system memory resources.

DSA Components: Intelligent Data Fragmentation

The parallelism and scalability of the DSA processor and memory components are supported by the ability to perform asynchronous I/O across database tables and indexes that have been logically partitioned. To speed up what is typically the slowest component of database processing, IDS uses its own asynchronous I/O (AIO) feature or the operating system’s kernel AIO, when available. Because I/O requests are serviced asynchronously, virtual processors don't have to wait for one I/O operation to be completed before starting work on another request. To ensure requests are prioritized appropriately, four specific classes of VPs are available to service I/O requests: logical log I/O, physical log I/O, asynchronous I/O, and kernel asynchronous I/O (KAIO). With this separation, you can create additional virtual processors to service specific types of I/O to alleviate any bottlenecks that might occur.

The read-ahead feature enables IDS to asynchronously cache several data pages from disk while the server processes the current set of pages retrieved into memory. This feature significantly improves the throughput of sequential table or index scans; end-user applications spend less time waiting for disk accesses to be completed.

Data Partitioning

You can logically divide table and index data into partitions, or fragments, using one or more *partitioning schemes* to improve the ability to access several data elements within the table or index in parallel as well as to increase and manage data availability and currency. For example, if a sequential read of a partitioned table were required, it would be completed more quickly because the partitions would be scanned simultaneously rather than each disk section being read serially from the top to the bottom. With a partitioned table, you can move, associate, or disassociate partitions to easily migrate old or new data into the table without tying up table access with mass inserts or deletes.

IDS has two major partitioning schemes that define how data is spread across the fragments. Regardless of the partitioning scheme chosen, or even if none is used at all, the effects are transparent to end users and their applications. Table partitions can be set and altered without bringing down the instance and, in some cases, without interrupting user activity within the table. When partitioning a table, you can specify either of two schemes:

- Round robin — Data is evenly distributed across each partition, with each new row going to the next partition sequentially.
- Expression-based — Data is distributed into the partitions based on one or more sets of logical rules applied to values within the data. Rules can be *range-based*, using operators such as =, >, <, <=, **MATCHES**, **IN**, and their inverses, or *hash-based*, where the SQL **mod** operator is used in an algorithm to distribute data.

Depending on the data types used in the table, individual data columns can be stored in a different kind of data storage space, called a *smart BLOBspace*, from the rest of the table's "regular" data (stored in *dbspaces*). These columns, primarily data types referred to as *smart large objects*, or *smart LOBs*, can have their own unique partitioning strategy to distribute the values across one or more smart BLOBspaces in addition to the partitioning scheme applied to the rest of the table. Another type of large object, called *simple LOBs*, can and should be fragmented into simple

BLOBspaces. Be aware, though: these types are black-box objects as far as the instance is concerned, so no further fragmentation options are possible.

You can also partition indexes using an expression-based partitioning scheme. A table's index partitioning scheme need not be the same as the table's data partition scheme. Partitioned indexes can be placed on a different physical disk than the data, resulting in optimum parallel-processing performance. Partitioning tables and indexes improves the performance of data-loading and index-building operations.

With expression-based partitioning, the IDS cost-based SQL optimizer can create more efficient and quicker plans using partition elimination to access only those table/index partitions where the data is known to reside or should be placed. The benefit is that multiple operations can be executing simultaneously on the same table, each in its unique partition, resulting in greater system performance than is available with other data servers.

Depending on the operating system used, you can configure unformatted disk partitions, or "raw" space, for the data server to use when creating spaces to store table or index data. When you employ raw disk space, IDS uses its own data storage system to allocate contiguous disk pages. Contiguous disk pages reduce latency from spindle-arm movement to find the next data element. When using raw disk space, IDS can use direct memory access to manage write operations. This capability is particularly valuable when kernel asynchronous I/O is available in the O/S. With the exception of Windows platforms, where standard file system space should be used, historically Unix/Linux raw-disk-based dbspaces have provided a measurable performance benefit. IDS 11 provides new technology that significantly improves performance for file system-based dbspaces.

Leveraging the Strengths of DSA

With an architecture as robust and efficient as IDS's, the data server provides several performance features that other servers cannot match.

High Performance Loader (HPL)

The High-Performance Loader (HPL) utility loads data into tables very quickly because it can read from multiple data sources (e.g., tapes, disk files, pipes, or other tables) and load the data in parallel. As the HPL reads from the data sources, it can execute data manipulation operations, such as converting from EBCDIC to ASCII, masking or changing data values, or converting data to the local environment based on Global Language Support (GLS) requirements. You can configure an HPL job so that normal load tasks, such as referential integrity checking, logging, and index builds, are performed during the load or afterward, which speeds up the load time. You can also use the HPL to extract data from one or more tables for output to one or more target locations. Data manipulation similar to that performed in a load job can be performed during an unload job.

Parallel Data Query (PDQ) and Memory Grant Manager (MGM)

IDS response to a data operation can vary depending on the amount of data being manipulated and the database's design. While many simple OLTP operations, such as single row inserts/updates/deletes, can be executed without straining the system, a properly designed database can leverage IDS features such as parallel data query, parallel scan, sort, join, group, and data aggregation for larger, more complex operations.

The Parallel Data Query (PDQ) feature takes advantage of the CPU power provided by SMP systems and the IDS virtual processors to execute fan-out parallelism. PDQ is of greatest benefit to more complex SQL operations that are more analytical (or OLAP-oriented) than operational (or OLTP-oriented). With PDQ enabled, not only is a complex SQL operation divided into a number of subtasks, but the subtasks are given higher or lower priority for execution within the data server's resources based on the overall "PDQ-priority" level requested by the operation.

The Memory Grant Manager (MGM) works with PDQ to control the degree of parallelism by balancing the priority of OLAP-oriented user requests with available system resources, such as memory, virtual processor capacity, and disk scan threads. Each OLAP query can be constructed to request a percentage of data server resources for execution by setting the operation's PDQ_PRIORITY value. You can set query type priorities, adjust the number of queries allowed to run concurrently, and change the maximum amount of memory used for PDQ-type queries. The MGM enforces the rules by releasing queries for execution when the proper amounts of system resources are available.

Full Parallelism

The parallel scan feature takes advantage of table partitioning in two ways. First, if the SQL optimizer determines that each partition must be accessed, a scan thread for each partition will be executed in parallel with the other threads to bring the requested data out as quickly as possible. Second, if the access plan calls for only "1" to "N–1" of the partitions to be accessed, another access operation can be executed on the remaining partitions so that two (or more) operations can be active on the table or index at the same time. Because disk I/O is the slowest element of database operations, scanning in parallel or having multiple operations executing simultaneously across the table/index can boost performance significantly.

As data is being retrieved from disk or from memory buffers, the IDS parallel sort and join technology takes the incoming data stream and begins the join and sorting process immediately rather than waiting for the scan to be completed. If several join levels are required, higher-level joins are immediately fed results from lower-level joins as they occur, as Figure 1.5 illustrates.

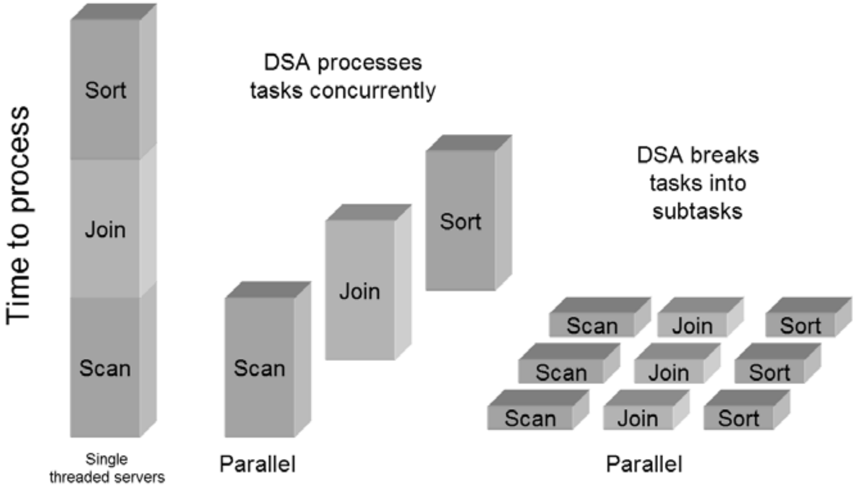


Figure 1.5: With full, integrated parallelism, IDS can simultaneously execute several tasks required to satisfy an SQL operation.

Similarly, if SQL aggregate functions such as **sum**, **avg**, **min**, or **max** need to be executed on the data, or if a **group by** SQL operator is present, these functions execute in real time and in parallel with the disk scan, join, and sort operations. As a consequence, a final result can often be returned to the requester as soon as the disk scan is completed.

Like the parallel scan, a parallel insert takes advantage of table partitioning, letting multiple virtual processors and update threads insert records into the target table (or tables) in parallel. This technique can yield performance gains proportional to the number of disks on which the table was fragmented.

With single-threaded data servers, index building can be a time-consuming process. IDS uses parallelized index-building technology to significantly reduce the time needed to build indexes. During the build process, data is sampled to determine the number of scan threads to allocate. The data is then scanned in parallel (using read-ahead I/O where possible), sorted in parallel, and then merged into the final index as illustrated in Figure 1.6.

In Version 11, IDS takes this efficiency one step further. During the index build, as the key values are being scanned to create the index, the optimizer traps distribution and other statistical information about the leading keys in the index. With this information, the index can be used in optimizer calculations immediately after its creation. You no longer have to execute a separate statistic-gathering operation, which puts an additional load on the system and delays use of the index.

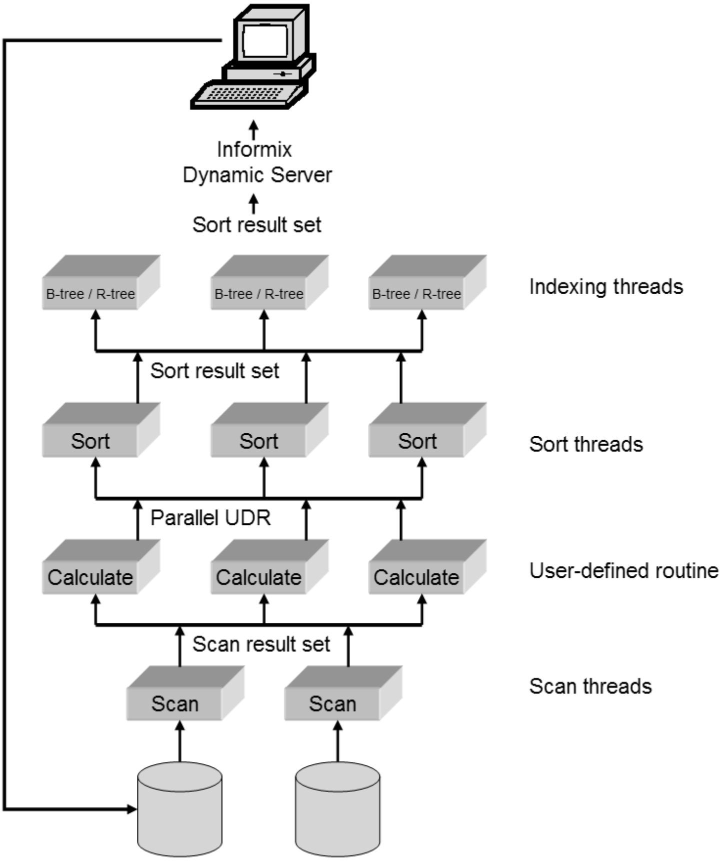


Figure 1.6: An example of how IDS uses parallelism to dramatically reduce index build and maintenance time

As with other I/O operations I've already mentioned, everything occurs in parallel; the sort threads don't need to wait for the scan threads to be completed, and the index builds don't wait for the sorts. This parallelization dramatically increases index-build performance compared with serial index builds.

IDS Cost-Based Optimizer

IDS uses a cost-based optimizer to determine the fastest way to retrieve data from database tables and/or indexes based on detailed statistical information about the data within the database generated by the **update statistics** SQL command. This statistical information includes more than just the number of rows in the table; the maximum and minimum values for selected columns, value granularity and skew, index depth, and more are captured and recorded in overhead structures for the optimizer. The optimizer uses this information to pick the access plan that will provide the quickest access to the data while trying to minimize the impact on system resources. The optimizer's plan is built using estimates of I/O and CPU costs in its calculations.

Access plan information is available for review through several management interfaces so you and your developers can evaluate the effectiveness of application and/or database design. The SQL operation (or operations) under review doesn't need to actually be executed for you to obtain the plan information. When you set an environment variable, execute a separate SQL command, or embed an instruction in the SQL operation, the operation will stop after being "prepared," and the access plan information will be output for review. With this functionality, you can test application logic and database design for efficiency without having to constantly rebuild data back to a known "good" state.

In this release of IDS, IBM has significantly expanded the access plan information available for review. This information now includes estimated as well as actual costs if the operation was allowed to execute, the capture of information for PDQ-enabled operations, and separate iterator-level statistics so you can analyze each component of the operation, not just the overall operation.

In some rare cases, the optimizer may not choose the best plan for accessing data. This result can happen when, for example, the operation is extremely complex or when insufficient statistical information is available about the table's data. In these situations, after careful review and consideration, you or your developer can influence the plan by including *optimizer directives* (also known as optimizer hints) in the SQL statement. You can set optimizer directives to use or exclude specific indexes, specify the join order of tables, or indicate the join type to be used when the operation is executed. You can also set an optimizer directive to optimize a query to retrieve only "N" rows of the possible result set. IBM has enhanced this functionality to include *registering* of optimizer directives for SQL operations whose source code is unavailable. This feature lets you enhance or modify data access operations that are executed by third-party applications that you've purchased and are delivered as compiled binaries.

An Introduction to IDS Extensibility

IDS provides a complete set of object-oriented technology to extend the data server and the databases it contains, including support for new data types, routines, aggregates, and access methods. With this technology, in addition to recognizing and storing standard character and numeric-based information, the data server can, with the appropriate access and manipulation routines, manage nontraditional data structures that are either modeled more like the business environment or contain new types of information never before available for business application processing. Although the data may be considered "nonstandard," and some types can be table-like in and of themselves, it is stored in a relational manner using tables, columns, and rows. In addition, all data, data structures created through Data Definition Language (DDL) commands, and access routines recognize objected-oriented behaviors such as overloading, inheritance, and polymorphism. This combination of object-oriented and relational capabilities, referred to as object-relational extensibility, supports transactional consistency and data integrity while simplifying database optimization and administration.

Other data servers rely on middleware to link multiple add-on servers, each managing different data types, to make it look as if there is a single processing environment. This approach compromises not only performance but also transactional consistency and integrity because problems with the network can corrupt the data. Still other data servers simply create a view that looks like an object, but in fact the "object" is a standard relational table, and "object indexes" are themselves tables that are sequentially scanned. This is not the case with IDS. Its object-relational technology is built into the DSA core, and you can use it, or not, at will within the context of a single database environment.

IDS Extensibility: Data Types

IDS uses a wide range of data types to store and retrieve data, as Figure 1.7 illustrates. The breadth and depth of the data types available to your database administrators and application developers is significant, letting them truly define data structures and rules that accurately mirror the business environment rather than trying to approximate it through normalized database design and access constraints.

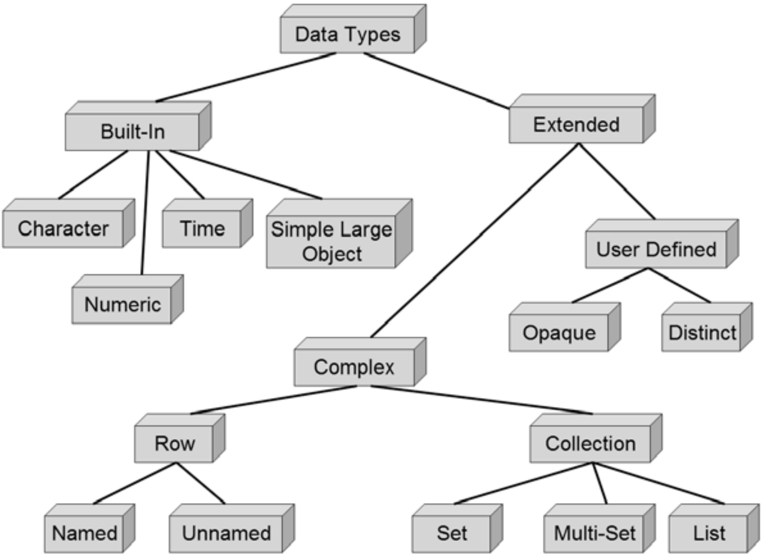


Figure 1.7: The IDS data type tree

Some types, referred to as built-in types, include standard data representations such as **character(n)**, **decimal**, **integer**, **serial**, **varchar(n)**, **date**, and **datetime**; alias types such as **money**; and simple large objects (**LOB**s). Recent releases of IDS have provided additional built-in types, including **boolean**, **int8**, **serial8**, and an even longer variable-length character string, the **lvarchar**.

Extended data types themselves are of two classes, including

- supersets of built-in data types with enhanced functionality
- types that weren't originally built into the IDS data server but that, once defined, can be used to intelligently model data objects to meet business needs

You use the **collection** type to store repeating sets of values within one row of one column that normally would require multiple rows or redundant columns in one or more tables in a traditional relational database design. The three collection types enforce rules about whether duplicate values or data order is significant. Collection data types can be nested and can contain almost any type, built-in or extended.

With **row** data types, you can build a new data type composed of other data types. The format of a row type is similar to that used when defining columns to build a table — an attribute name and a data type. Once defined, row types can be used as columns within a table or as a table in and of themselves. With certain restrictions, a row type can be dynamically defined on the fly as a table is being created or can be inherited into other tables, as the examples in Figure 1.8 illustrate.

A **distinct** data type is an alias for an existing data type. A newly defined distinct data type inherits all the properties of its parent type (for example, a type defined using a **float** parent will inherit the elements of precision before and after the decimal point), but because it is a unique type, you can't combine its values with any other data type but its own without either "casting" the value or using a user-defined routine.

Last, **opaque** data types are those created by developers in C or Java™; they can be used to represent any data structure that needs to be stored in the database. When you use opaque data types, as opposed to the other types already mentioned, the data server is completely dependent on the type's creator to define all access methods that might be required for the type, including insert, query, modify, and delete operations.

Extended data types can be used in queries or function calls, passed as arguments to database functions, and indexed and optimized in the same way as the core built-in data types. Because any data that can be represented in C or Java can be natively stored and processed by the data server, IDS can encapsulate applications that have already implemented data types as C or Java structures. Because the definition and use of extended data types is built into the DSA architecture, specialized access routines support high performance. The access routines are fully and automatically recoverable, and they benefit from the proven manageability and integrity of the IDS data server architecture.

```
Named row type:
create row type name_t
     (fname char(20),
      lname char(20));

create row type address_t
     (street_1 char(20),
      street_2 char(20),
      city char(20),
      state char(2),
      zip char(9));

create table student
   (student_id serial,
    name name_t,
    address address_t,
    company char(30));

Unnamed row type:
ROW (a int, b char (10))

-which is equal to-

ROW(x int, y char(10))

create table part
   (part_id serial,
    cost decimal,
    part_dimensions row
      (length decimal,
       width decimal,
       height decimal,
       weight decimal));
```

Figure 1.8: Examples of "named" and "unnamed" row types and their use

IDS Extensibility: Data Type Casting

With the enormous flexibility and capability that both built-in and extended data types provide to create a database environment that accurately matches the business environment, you'll need to use the types together fairly often. Doing so requires functionality to convert values between types. You generally accomplish this through the use of *casts*, and quite often the casting process will use *user-defined functions (UDFs)*.

Casts enable you to manipulate values of different data types together or to substitute the value of one type in the place of another. While casts, as an identifiable function, have only recently been added to the SQL syntax, IDS administrators and developers have been using casts for some time; however, casts have been hidden in the data server's functionality. For example, to store the value of the integer "12" in a table's character field requires casting the integer value to its character equivalent, and the data server performs this action on behalf of the user. The inverse cannot be done because no appropriate cast is available to represent a character (such as an "a") in a numeric field.

When you use *user-defined types (UDTs)*, you must create casts to change values between the source type and each of the expected target data types. For some types, such as collections, LOBs, and unnamed row types, casts cannot be created due to the unique nature of these types. You can define casts as either *explicit* or *implicit*. For example, with an implicit cast, you can create a routine that adds values of type "a" to the value of type "b" by first converting the value of one type to the other type and then adding the values together. The result can either remain in that type or be converted back into the other type before being returned. Any time an SQL operation requires this operation to occur, this cast is automatically invoked behind the scenes and a result returned. An explicit cast, while it may perform the exact same task as an implicit cast, is executed only when it is specifically called to manipulate the values of the two data types. Although using explicit casts requires a little more developer effort, more program options are available with their use based on the desired output type.

IDS Extensibility: User-Defined Routines, Aggregates, and Access Methods

In earlier versions of IDS, if you wanted to capture application logic that manipulated data and have it execute within the data server, you had only stored procedures to work with. Although stored procedures have an adequate amount of functionality, they may not be the best solution from a performance perspective. IDS now provides the ability to create significantly more robust and better-performing application or data manipulation logic in the data server, where it can benefit from the processing power of the physical server and the DSA.

A *user-defined routine (UDR)* is a collection of program statements that — when invoked from an SQL statement, a trigger, or another UDR — perform new domain-specific operations, such as searching geographic data or collecting data from Web site visitors. UDRs are most commonly used to execute logic in the data server — either generally useful algorithms or business-specific rules — reducing the time it takes to develop applications and increasing application speed. UDRs can be *functions* that return values or *procedures* that do not. They can be written in IBM Informix Stored Procedure Language (SPL), C, or Java. SPL routines contain SQL statements that are parsed, optimized, and stored in the system catalog tables in executable format — making SPL ideal for SQL-intensive tasks. Because C and Java are powerful, full-function development languages, routines you write in these languages can carry out much more complicated tasks than SPL routines. C routines are stored outside the data server with the path name to the shared library file registered as the UDR. Java routines are first collected into "jar" files, which are stored inside the database server as *smart large objects (SLOs)*. Regardless of their storage location, your C and Java routines are executed as if they were a built-in component of IDS.

A *user-defined aggregate (UDA)* is a UDR that can either extend the functionality of an existing built-in aggregate (e.g., **SUM** or **AVG**) or provide new functionality that wasn't previously available. In general, aggregates return summarized results from one or more queries. For example, the built-in **SUM** aggregate adds values of certain built-in data types from a query result set and returns their total. You can, for example, create an extension of the **SUM** aggregate to include user-defined data types, enabling the reuse of existing client application code without requiring new SQL syntax to handle the functionality of new data types within the application. To do so, using the example of the **SUM** aggregate, would require you to create (and register) a user-defined function that would overload the **PLUS** function and take the user-defined data types that needed to be added together as input parameters.

To create a completely new user-defined aggregate, you need to create and register two to four functions to perform the following tasks:

1. Initialize the data working space.
2. Merge a partial existing result set with the result of the current iteration.
3. Merge all the partial result sets.
4. Return the final result set with the associated closure and release of system resources to generate the aggregate.

In defining the ability to work with partial result sets, UDAs can, like built-in aggregates, execute in parallel. Functions created and registered for UDAs can be written in SPL, C, or Java. As with built-in aggregates, the data server wholly manages a UDA once it's registered (as either an extended or a user-defined aggregate).

IDS provides primary and secondary *access methods* to access and manipulate data stored in tables and indexes. Primary access methods, used with built-in data types, provide functionality for table use. Secondary access methods are specifically targeted to indexes and include B-tree and R-tree indexing technologies as well as the CopperEye Indexing DataBlade module, which significantly reduces the creation and maintenance of indexes on extremely large data sets. You can create additional user-defined access methods to access other data sources. IDS has methods that provide SQL access to data in a heterogeneous data server table, in an external sequential file, or to other nonstandard data stored anywhere on the network. Secondary access methods can be defined to index any data, as can alternative strategies to access SLOs. You can create these access methods using the Virtual Table Interface (VTI) and the Virtual Index Interface (VII) server APIs.

IDS Extensibility: DataBlades

IBM Informix DataBlade™ modules bring additional business functionality to the data server through specialized user-defined data types, routines, and access methods. Your developers can use these new data types and routines to more easily create and deploy richer applications that better address your company's business needs. IDS provides the same level of support to DataBlade functionality that is accorded to built-in or other user-defined types/routines. With IBM Informix DataBlade modules, almost any kind of information you need to work with can be easily managed as a data type within the data server.

With this release of IDS, several Blades have been bundled into the data server for your use. The *IBM Informix Basic Text Search DataBlade* gives you the ability to perform word or phrase searching through the use of a text search index. This Blade works against text stored in **CHAR**, **VARCHAR**, **LVARCHAR**, **BLOB**, or **CLOB** data types and searches based on a predicate containing a single word, a phrase, single or multiple wildcards, or Boolean operators. It can execute exact, fuzzy, or proximity searches.

The *IBM Informix Node DataBlade* enables you to model data that is hierarchical in nature. It's impossible, for example, to accurately model an organization chart or your genealogy in a relational data server. The closest you can come is a table with keys that refer back to the same table. Processing of operations on this type of table is difficult and complex to write and manage. It involves either recursive programming or set-processing routines, neither of which is very fast. In simple tests using less than 100 MB of data, there was a 40X performance decrease when compared with data modeled using this Blade's technology. I'm sure the performance delta between the two would increase significantly as the amount of data grew.

A growing portfolio of "for purchase" third-party DataBlade modules is also available. Or your developers can use the IBM Informix DataBlade Developer's Kit (DBDK) to create specialized Blades for your particular business needs. The following is a partial list of available IBM Informix DataBlade technologies (for a current list, visit *http://www.ibm.com/informix*):

- *IBM Informix TimeSeries DataBlade* — This DataBlade gives you a better way to organize and manipulate any form of realtime, timestamped data. Applications that use large amounts of timestamped data, such as network analysis, manufacturing throughput monitoring, scientific or medical instrument monitoring, or financial ticker data analysis can provide measurably better performance and reduced data storage requirements with this DataBlade than you can achieve using traditional relational database design, storage, and manipulation technologies.
- *IBM Informix TimeSeries Real-Time Loader®* — A companion piece to the IBM Informix TimeSeries DataBlade, the TimeSeries Real-Time Loader is specifically designed to load timestamped data and make it available to queries in real time. Many customers use this Blade for streaming data loads in situations where the data must be available for use in a table less than one second after being presented to the server.
- *IBM Informix Geodetic DataBlade* and *IBM Informix Spatial DataBlade* — These two DataBlades give you the functionality to intelligently manage

complex geospatial information within the efficiency of an object-relational database model. The Geodetic DataBlade stores and manipulates objects from a "whole-earth" perspective using four dimensions — latitude, longitude, altitude, and time. It's designed to manage spatio-temporal data in a global context, such as satellite imagery and related metadata, or through trajectory tracking in an airline, cruise, or military environment. The Spatial DataBlade is a set of routines written to the open-GIS (Geographic Information System) Simple Features for SQL specification. It takes a "flat-earth" perspective to mapping geospatial data points. Based on ESRI technology (*http://www.esri.com*), routines, and utilities, this DataBlade is better-suited to answering questions such as "How many grocery stores are within *n* miles of point X?" or "What's the most efficient route from point A to point B?" All IBM Informix geo-oriented DataBlades take advantage of the built-in IBM Informix R-tree multidimensional index technology, resulting in industry-leading spatial query performance. While the Geodetic DataBlade is a for-charge item, the Spatial DataBlade is available at no charge to all licensees beginning with this release; previously, only Enterprise Edition licensees could use its functionality without an additional license fee.

Note: As an interesting side note, with IDS's object-oriented capabilities, a friend of mine was able to "inherit" the properties of the geo-oriented DataBlades along with another custom user-defined type and set of functions to create a fabric classification and management system for a large manufacturer. He used the dimensional attributes of the geo blades to describe color using the Cielab classification system (magenta, cyan, and yellow) and a hierarchical data type to structure the fabric by "kind" (natural, artificial; then wool, cotton, silk; then woven, knit; and so on). This example illustrates the power of IDS: the ability to model and use data as you need to in your business.

- *IBM Informix Excalibur Text DataBlade* — Using this DataBlade, you can perform full-featured text searches of documents stored in database tables. It supports any language, word, or phrase that can be expressed in an eight-bit, single-byte character set. This Blade provides significantly greater functionality than the Basic Text Search DataBlade in that the Excalibur Blade understands document formats such as Adobe PDF or Microsoft PowerPoint and can search inside documents of these types (and many more) stored in databases in the data server.

- *IBM Informix Video Foundation DataBlade* — This Blade lets strategic third-party development partners incorporate specific video technologies, such as video servers, external control devices, codecs, or cataloging tools, into your data server. It also provides the ability to manage video content and video metadata regardless of the content's location.
- *IBM Informix Image Foundation DataBlade* — The Image Foundation DataBlade provides functionality for you to store, retrieve, transform, and convert the format of image-based data and metadata. While this DataBlade supplies basic imaging functionality, third-party development partners can also use it as a base for new DataBlade modules to give you new functionality, such as support for new image formats, new image-processing functions, and content-driven searches.
- *IBM Informix C-ISAM DataBlade* — The C-ISAM DataBlade provides two separate pieces of functionality to the storage and use of Indexed Sequential Access Method (ISAM)–based data. If you have an environment with native ISAM flat-file data, the DataBlade provides data server–based SQL access to the data. From a user or application developer perspective, it's as if the data resided in one of IDS's database tables. The second element of functionality enables you to store/retrieve ISAM data in/from IDS database tables while preserving the native C-ISAM application access interface. From your C-ISAM developer's perspective, it's as if the data continues to reside in its native flat-file format; however, with the data stored in a full-functioned data server environment, transactional integrity can be added to C-ISAM applications. Another benefit to storing C-ISAM data in the data server is the more comprehensive backup and recovery routines provided by IDS.

The IBM Informix DataBlade Developer's Kit is a single development kit for C-, Java-, and SPL-based DataBlades and the DataBlade API. The API is a server-side C API for adding functionality to the data server as well as for managing database connections, server events, errors, and memory and processing query results. Additional support for your DataBlade module developers includes the IBM Informix Developer Zone, available at *http://www7b.boulder.ibm.com/dmdd/zones/informix*. There, developers can interact with peers, exchange information and expertise, and discuss new development trends, strategies, and products. Examples of DataBlades and Bladelets, indexes, and access methods are available for downloading and use. Online documentation for the DBDK and other IBM Informix products is available at *http://www.ibm.com/informix/pubs/library*.

Informix Dynamic Server Editions and Functionality

With data server technology as dynamic and flexible as the DSA, it's only natural to assume that you can buy just the level of IDS functionality you need. IBM has packaged Informix Dynamic Server into three "editions," each tailored from a price and functionality perspective to a specific market segment. Regardless of the edition purchased, IDS comes with the full implementation of DSA and its unmatched performance, reliability, ease of use, and availability. Pricing varies based on licensing and/or connection and scalability restrictions. Below is a brief comparison of the three editions and their feature sets.

- *IBM Informix Dynamic Server Express Edition* — Targeted to small- to medium-sized businesses requiring enterprise-class OLTP performance without all the extra features and price tag. This edition is licensed on either a per-processor or a "server plus authorized user" basis. It is limited to using no more than two system processors and 4 GB of RAM and is available only for Linux and Windows. You can upgrade IDS-Express directly to any other edition simply by installing the new data server binaries.
- *IBM Informix Dynamic Server Workgroup Edition* — Stakes out the middle ground for midsized companies or departmental servers in a much larger enterprise deployment. Available on all supported operating systems, this edition has higher hardware limits than IDS-Express: four system CPUs and 8 GB of RAM. IDS Workgroup's licensing model is slightly different, though. While it can be licensed per CPU for applications with large user counts, IDS Workgroup also has a "server plus concurrent session" model. With this approach, you buy a license for the data server plus the anticipated *concurrent sessions*. This scheme is a departure from earlier IDS licensing, which was user-based. Under the new model, pricing is determined by the number of concurrent sessions the data server is supporting, regardless of whether they come from just you or from 500 other people. You can't hide behind transaction concentrators, either; you're responsible for the full concurrent session count as seen by the concentrator.

  IDS Workgroup Edition gives you additional data server functionality, with the ability to configure and use Parallel Data Query metrics to reserve data server resources for complex SQL operations. You can also use the High Performance Loader and the parallel backup and restore options in the **ON-Bar** utility suite. An IDS Workgroup instance can be a leaf node (or target) in an Enterprise Replication (ER) cluster receiving data updates from other nodes in the clusters. Last, you can buy an optional license and use High-Availability Data Replication (HDR) to instantiate a disaster-recovery, hot-site failover node.
- *IBM Informix Dynamic Server Enterprise Edition* — Includes all the features of IDS Workgroup with unlimited scalability required for the highest OLTP performance. This edition can be purchased on a system CPU or concurrent session basis. With this edition, full HDR and ER functionality is included, along with all the bundled DataBlades, such as the geospatial, node, and basic text search. New to Version 11 is the ability to buy optional packages to add

the discrete units of functionality you need. See your authorized IDS reseller for a list of these options.

Conclusion

In these few short pages, I've tried to give you a basic understanding of what the Informix Dynamic Server data server is, how it compares with other data servers, and some of the unique and market-leading features and functionality it provides. There's no way I can explain everything or describe how to set up and administer an IDS environment here; you'll need to buy my other books to get that information! ☺ By now, though, you should realize that, at a high level, IDS is a very powerful and easy-to-use data server — one I hope you'll consider using for your next project if you're not already.

From this point forward, we'll discuss in greater detail the new features in Version 11. We'll begin in Chapter 2 by looking at technologies that can help you achieve security and regulatory compliance objectives, be they internal or externally mandated.

2

Security and Regulatory Compliance

If there's one area in computing infrastructure or requirements that has experienced great activity and pressure in the past few years, it has been all aspects of data security and historical auditability. Driven by new and ever-changing regulatory compliance requirements to trace actions and patterns in order to discourage illegal activity, today's businesses are creating an avalanche of new data along with the requirement to protect that data from anyone inside or outside the company who shouldn't see it.

In Informix Dynamic Server Version 10, IBM added on-disk column-level encryption, giving customers the option of using several ciphers to encrypt data as it flowed in or out of the data server to or from rest on-disk. With IDS 11, additional functionality helps you and your business comply with new security and audit compliance rules. This chapter discusses the highlights of these features.

Label-Based Access Control

IDS 11's *Label-Based Access Control (LBAC)* facility uses a set of security *policies* and *labels* to control the ability of any user to read, write, delete, or update data in a database. You use labels to enforce security policies defined in the database that govern data access to the selected tables. The facility enforces these policies regardless of the method used to access the data, be it application-based SQL operations, utility-based operations such as load or unload, or jobs using **dbexport/dbimport** or **dbschema** operations. Operations executed through the High-Performance Loader (HPL) utility or the **onload/unload** utilities are also possible, although the user ID executing them must hold the credentials to bypass LBAC control.

Table 2.1 illustrates how you can control access using LBAC. In this example, the table policy specifies three levels of security, with **level_1** having the broadest privilege level and **level_3** being the most restrictive, or limited, in terms of access. If a user has **level_3** security, he or she can see only rows 1, 3, 7, and 9. Someone with **level_2** privileges can see those rows plus rows 2, 5, and 8, and so on.

Table 2.1: Controlling access through LBAC

Row #	Security label	Columns 1, 2, 3, ...
1	level_3	Data . . .
2	level_2	Data . . .
3	level_3	Data . . .
4	level_1	Data . . .
5	level_2	Data . . .
6	level_1	Data . . .
7	level_3	Data . . .
8	level_2	Data . . .
9	level_3	Data . . .

Using LBAC requires the creation and use of a new data server-based role, **DBSECADM**. Membership in this role can be granted only by a member of the **DBSA** group. **DBSECADM** is a data server–level role and is the only role that can create, drop, or rename security policies, security labels, or the label elements that make up the security label itself. A member of the **DBSECADM** role can attach or detach policies to or from tables as well as grant label privileges to the other user IDs accessing the instance. This new role helps with aspects of government-oriented security requirements by removing the need to use the **root** and/or **Informix** user IDs to manage certain aspects of the data server. Because only a member of the **DBSA** group can grant this role to a user ID, you must carefully control who is a member of the **DBSA** group. User **Informix** does not have the **DBSECADM** role by default and cannot grant the role to user **Informix**; however, user **Informix** can grant the role to another user.

The basic rules of label-based access control are patterned after the Bell-LaPadula security model. (For details about this model, including the 1976 paper that defines it, see http://en.wikipedia.org/wiki/Bell-LaPadula_model.) A security administrator assigns security labels to users, and the system assigns labels to data objects (in a database, rows of data) on behalf of the users that create those objects. The system enforces rules so that no user sees data that he or she isn't allowed to see and so that no user creates data with a label that permits others to see data they shouldn't see. If a user has a high security clearance (label) for read access, that user can read data protected with the same label or with a less secure label. The user's read label must "dominate" the data's label. Similarly, any data written on behalf of that user will have an appropriate high security clearance so that other people with lower-security clearances won't be able to see the data generated by that user. This security approach is called "no read up, no write down"; a user can't read material classified at a higher level than his or her access permits and can't write to a lower classification level than his or her access permits. There is also a system of exemptions that can be granted (temporarily) to let authorized users bypass the controls. The LBAC system of protections is more complex than the plain Bell-LaPadula model because it permits compartmentalization of data within security levels (or compartmentalization instead of security levels).

A security policy is defined as a set of security components from which the label definitions are drawn. A simple policy might use just one security label component, but LBAC permits you to use as many as 16 components if you really want or need to assert that much access control. As you might expect, systems with that many components are difficult to understand and manage. Comprehensible systems will use at most three components. Although from a design perspective you would take a different approach, construction of the security policy pieces begins with the security components, followed by policy and then labels.

A simple policy will consist of one security component and will have very simple labels, too:

```
create security label component secrecy array [‘secret’,
‘confidential’, ‘public’];
create security policy example_com component secrecy;
create security label example_com.secret component secrecy
‘secret’;
create security label example_com.confidential component secrecy
‘confidential’;
create security label example_com.public component secrecy
‘public’;
```

Now you can apply the policy to tables and grant labels to users:

```
alter table product_plans add (rowsec idssecuritylabel) add
security policy example_com;

grant label example_com.public to public;
grant label example_com.secret to insider;
grant label example_com.confidential to minion;
```

The special built-in type **IDSSECURITYLABEL** means that each row in the **product_plans** table must have a security label. You can also apply labels to columns. If you do so, a user’s label must dominate the column’s label if the user is to read the column at all. Separately, the user’s read label must dominate the row label if both row and column labels exist on the table.

You can build more complex policies from more components or from different types of components. You can define components using one of three models: **array**, **set**, or **tree**. An **array** component (the type you saw in the example above) is the simplest and, as the name implies, consists of an ordered list of elements, highest priority/privilege first. You can define up to 64 “elements” in an **array** component.

```
create security label component personnel_level array [‘Board’,
‘Executive’, ‘Director’, ‘Manager’];
```

You can read only data that is equal to or less than your level in the array. Writing data is a little different; you can write only at your own level.

A **set** component is an unordered group. You can define one or more of these components for each policy, up to a limit of 64:

```
create security label component department set {‘Marketing’,
‘Sales’, ‘Support’, ‘Distribution’, ‘Development’, ‘Personnel’};
```

When a policy uses **set** components, your label must include all the components defined for a given row in order to read from or write to the row.

The last component model, the **tree**, is a hierarchical model. Like the others, it is limited to 64 elements, but this model understands levels, relationships between levels, and level memberships.

```
create security label component region tree (‘Corp_wide’ root,
‘East’ under ‘Corp_wide’, ‘West’ under ‘Corp_wide’, ‘Central’
under ‘Corp_wide’, ‘Southern_California’ under ‘West’ . . . );
```

You can access data under a tree component model if your label contains any of these components or the ancestor of a component.

The **policy** is the name of the security policy protecting the table; it consists of one or more *security label components*. For example, the statement

```
create security policy my_co components personnel_level,
department, region;
```

creates a security policy with three areas of finer segregation. You can define up to 16 security label components within each policy.

To actually create a label and implement the policy, you can combine several pieces together. For example:

```
create security label my_co.top_executive
component level ‘Board’,
component department ‘Marketing’, ‘Sales’, ‘Support’,
‘Distribution’, ‘Development’,
component region ‘Corp_wide’;

create security label my_co.IT_Development
component department ‘Development’;
```

Once you’ve created a label, you can “grant” it to one or more users as well as use it to protect either rows or columns within database tables. Protecting rows in a table requires you to define a column with the new **IDSSECURITYLABEL** data type as shown in Figure 2.1.

```
create table my_tab
   ( col1 idssecuritylabel,
       col2 serial,
       col3 name_t,
       col4 address_t
       .
       .
       );

create table my_tab_2
   ( col1 int,
       col2 char(10) column secured with  my_co.top_executive,
       col3 smallint,
       .
       .
       );

alter table my_tab_3 add (col_14 idssecuritylabel) add security
policy DB_Development;

alter table my_tab_4 modify (col_3 char(15) column secured with
SoCal_Sales;
```

Figure 2.1: Syntax examples of LBAC policy implementations

Not all tables and columns can be protected with LBAC labels. Tables created through inheriting a **named row** type, explicitly created temp tables, tables that are hierarchical in nature, tables built using the Virtual Table Interface (VTI), or those that use the Virtual Index Interface (VII) cannot support LBAC security policies. Columns that are used for keys (primary or foreign) or that have a check constraint or a unique constraint defined on them cannot support column-level policies.

A set of read functions and a set of write functions enforce the LBAC “no read up” and “no write down” policies. In normal operation, you don’t need to know the names of these functions or invoke them explicitly; the system simply enforces the rules. However, a user with **DBSECADM** authority might temporarily grant another user an exemption from the normal rules to let that user do something that otherwise would be prohibited. The three read functions are **IDSLBACREADARRAY**, **IDSLBACREADSET**, and **IDSLBACREADTREE**; they control what happens when a user tries to execute **select**, **update**, or **delete** operations to ensure the user’s individual security label contains the required elements to access the data. The **IDSLBACWRITEARRAY**, **IDSLBACWRITESET**, and **IDSLBACWRITETREE** functions execute on **insert**, **update**, and **delete** operations to enforce the “write down” rules. Users will have a read and a write label for each security policy of which they are a member. The read permissions must always be greater than the write permissions.

Three other functions are available for use in **select**, **update**, and **insert** operations to either populate or convert label permissions on data secured with a label. The **SECLABEL_BY_COMP** function provides a row’s security label during an **insert** or **update** operation based on the security components passed into the function as part of the operation. The **SECLABEL_BY_NAME** function provides labels to inserted or updated rows based on the component name. The **SECLABEL_TO_CHAR** function retrieves the security label of a queried row. Figure 2.2 shows syntax examples for these functions.

```
insert into my_tab_1 values (seclabel_by_comp(‘my_co’,
‘Marketing:West’), 2345, “abc”);

update my_table_89 set col_12 = seclabel_by_name(‘my_co’,
‘DB_Development’);

select seclabel_to_char(‘my_co’, col_1), col_2, col_4
from my_table_89;
```

Figure 2.2: Using the LABC built-in functions to select, insert, or update data

Users with **DBA** permissions in the database who are required to execute certain database maintenance operations can, with direct SQL access, change their identity and access level by using the **set session authorization** SQL command. Although they may be able to make changes to the database, they shouldn’t be able to see data outside their “real” policy level. A new privilege, **SETSESSIONAUTH**, has been created to limit those who can successfully execute the **set session authorization** command.

Common Criteria Certification

Common Criteria certification isn’t a “feature” in the sense of new functionality; rather, it is the vetting, or verification, of IDS’s compliance with an internationally recognized security standardization structure. According to the Common Criteria organization (*http://www.commoncriteriaportal.org*), Common Criteria grew out of the U.S. Department of Defense Orange Book and other standards to define what a secure computing environment was from hardware, operating system, software, and other perspectives. As other countries began developing their own criteria — particularly Canada and the European ITSEC — it became clear that a single international standard (ISO *xxxxx*-*x*:200*x*) would help companies more efficiently design, certify, and sell solutions needed around the world.

Version 2.0 of the Common Criteria is now in use and is used to define *Protection Profiles (PPs)*, or a set of standards and practices that must be met. The PPs are evaluated for compliance against a *Target of Evaluation (TOE)* through various methods, depending on the target. If discrepancies are found, the system developer must make changes to comply with the PP.

Eleven areas of functional components can be tested under the Common Criteria. Although not all the areas apply to data servers, it's interesting to note how comprehensive they are:

- FAU (Audit) — recognizing, recording, storing, and providing a way to analyze data related to security and security actions
- FCO (Communications) — identity validation of both the sender and the receiver of traffic
- FDP (User Data Protection) — protection of user information at all times, including at rest and in transit via import and export
- FIA (Identification and Authentication) — the unambiguous identification of all parties and the correct association of privileges based on their identity
- FCS (Cryptographic Support) — the implementation of cryptographic functionality in the system
- FRU (Resource Utilization) — fault tolerance, resource allocation, and prioritization of tasks in the system's processing and storage capacities
- FPR (Privacy) — protection of the user and the user identify against discovery and misuse
- FPT (Protection of Target of Evaluation Functions) — protection of data in the TOE security systems (as opposed to user data)
- FMT (Security Management) — management of the TOE's security functions, including attributes, functions, and data
- FTP (Path/Channels) — building trusted communications channels between all entities in the secured TOE, including users and TOEs
- FTA (TOE Access) — requirements in addition to those of the FIA for the beginning of a user session on a TOE, including access histories, changes to access parameters, permissions, scope, and other limits

A series of assurance classes in the evaluation process of these components look at everything from development of the specification to documentation, object creation, asset protection, system tests, threat assessments, and ongoing support and enhancements.

Validation against the standards proceeds on a seven-step gradient of increasing severity, as described in Table 2.2.

Table 2.2: Common Criteria security levels

Common criteria	Description
EAL 1	Functionally tested
EAL 2	Structurally tested; controls user access to data
EAL 3	Methodically tested and checked; validates user accountability features such as logins
EAL 4	Methodically designed, tested. And reviewed; validates security policies and labeled data
EAL 5	Semi-formally designed and tested; validates more explicit and formalized security policies
EAL 6	Semi-formally verified design and tested; stringent engineering and monitoring controls; highly secured systems
EAL 7	Formally verified design and tested; tests no additional functions over EAL 6, but subjects the functions to a formalized functional analysis to ensure compliance and security

With Version 11, IDS development is pursuing the highest level of certification for a data server, the EAL 4 rating. Testing for certification cannot proceed until IDS 11 is publicly available; IBM is highly confident that IDS will achieve certification shortly thereafter.

Encryption in an HA Environment

For more than 10 years, IDS has provided customers like you with unmatched replication functionality to create a highly available data server environment that can not only survive a server outage but also distribute data through your environment based on rules you set without having to change your applications. Recently, other IBM data servers have adopted some of this technology to offer the same level of protection to a larger population of IBM customers.

One of the key technologies in this area is called *High-Availability Data Replication (HDR)*. HDR is designed to provide hot-site failover in the event of a data server outage for any reason. IDS 11 includes three major enhancements to this technology, one of them in the area of security.

The underlying architecture of HDR involves transferring logical log records of data, index, and schema change events between the primary server and its secondary. Until IDS 11, these records were transmitted in the clear; now, you have the ability to encrypt the communications between the servers. HDR encryption uses the same facilities as Enterprise Replication encryption, which was available in IDS 10. With the new release, the encryption “engine” has been changed to IBM Crypto for C, a Federal Information Processing Standards (FIPS) 140-2 certified technology.

Enabling HDR encryption occurs in the **$ONCONFIG** file with a new **ENCRYPT_HDR** parameter. By default, HDR encryption is turned off; if it is enabled, the instance (or instances) must be restarted because the parameter is read only during instance startup. Once enabled, four other **$ONCONFIG** parameters — **ENCRYPT_CIPHERS**, **ENCRYPT_MAC**, **ENCRYPT_MACFILE**, and **ENCRYPT_SWITCH** — are used to define the cipher and message authentication code (MAC) level. These parameters, too, are read only at instance startup. It should go without saying that all servers involved in encrypted HDR communications should have the same values for these parameters. For example:

```
ENCRYPT_HDR 1
ENCRYPT_CIPHERS aes:cbc
ENCRYPT_MAC high
ENCRYPT_MACFILE $INFORMIXDIR/etc/mac.myserver
ENCRYPT_SWITCH  10,10
```

These are the only parameters and/or changes required for HDR encryption. Encrypted HDR communication occurs over the same standard network instance alias defined for unencrypted HDR communication. With that in mind, if you plan to encrypt client/server traffic, you’ll need to define an additional network instance alias in **$ONCONFIG**, **$INFORMIXSQLHOSTS**, and **/etc/services** for this communication to occur through an encryption Communication Support Module.

sysdbopen and sysdbclose Functions

Although the description of **sysdbopen** and **sysdbclose** could fit here as a security enhancement, these two functions also are administrative aids. For the purposes of this book, I’ve included them in the next chapter, which covers administrative enhancements.

Backup and Restore Filters

Similarly, the description of IDS’s backup and restore filters could be covered here as a security enhancement, but these functions also constitute a significant enhancement to IDS’s backup and recovery facilities. For the purposes of this book, I’ve included them in the next chapter as well.

This page intentionally left blank

3

Relieving the Load on the IDS Administrator

In this chapter, we look at some enhancements that make an Informix Dynamic Server administrator's job easier. This assignment is a tough one because, unlike other data servers, IDS requires little to no ongoing care and feeding. With its extensible and dynamic architecture, IDS is smart enough to handle nearly all situations on its own. That said, IDS 11 does provide new functionality in this area that will be of benefit to you.

New Administration API

One of the few complaints new administrators make about IDS is that it's "difficult" to understand the IDS command-line utilities. As a data server deeply rooted in the Unix world, IDS's primary administrative interface has been a series of command-line interface (CLI) tools. Only recently have graphical utilities been created for IDS — because many people felt they weren't needed! As a long-time IDS administrator myself (before "crossing over to the dark side" and becoming the vendor), I (and my peers) used to laugh at graphical utilities: "We don't need no stinkin' GUI to run IDS! We can do everything with a couple of parameters from the command line."

Well, your mileage may vary on this issue, but if you really can't live without graphical tools, you'll be glad to know they exist. But IDS 11 provides a new way to administer IDS that requires neither the GUI nor learning the intricacies of the **onspaces** or **onparams** CLI: the SQL Administration API.

Built on top of the **sysadmin** database, a new instance-level database, and about 16 new tables in the **sysmaster** database, the SQL Administration API gives members of the **DBSA** group access to a broad range of functionality, including alerts, performance monitoring, and baseline generation as well as instance maintenance. Because the API is SQL-based, you can execute commands from any machine with connectivity to the network — there's no need to have the overhead of a Web server or a heavy-weight client application.

Focusing on the administration part for the moment, the API provides a set of functions to manage storage spaces, instance configuration, and validation as well as perform routine operations. Executed commands and their results are captured for forensic purposes should the need arise to review what's changed in the instance.

The API's **task** function accepts a series of string fields that indicate the task to perform and any parameters for the task. The function returns plain text indicating whether the command succeeded. For example:

```
execute function task('add log', 'logs_space', '10 MB');
(expression) created logical log number 15 in logs_space

execute function task('archive fake');
(expression) backup complete
```

The API's **admin** function accepts the same set of parameters but returns an integer based on the success or failure of the command. If the returned value is positive, the command succeeded and the returned value represents the serial number value of the command in the history table. If the function returns **0** (zero), the command succeeded, but the data server wasn't able to insert a new row into the history table (time to check your free space!). If the returned value is a negative number, the command failed. Here are a couple **admin** examples:

```
execute function admin ('create dbspace',
'/opt/IBM/Informix/devices/tagus/data_1', 'tagus_data_1',
'300 MB', '0');
(expression) 234

execute function admin ('shutdown');
(expression) 400
```

To minimize the amount of typing, path names can start with an environment variable provided the variable exists in the target server's environment. You can also use "real" units for sizes (KB, MB, GB, and so on) as opposed to trying to remember whether the IDS CLI utilities want units in pages or kilobytes.

All told, you can execute about 80 operations or tasks through the SQL Administration API, making it a serious contender for my preferred IDS administration tool! I discuss other parts of the API in other sections of this chapter.

Database Scheduler

In highly segregated IT departments, a series of Maginot Lines surround each area of specialization. Many Unix administrators don't work with data server administrators or give them "super" privileges to execute O/S operations such as creating directories or changing permissions. Storage array administrators tell both Unix and data server administrators to mind their own business; the array they configure will eliminate all contention yet save the company money on disk space used. And heaven forbid you cross a network administrator; you'll never get the port upgrade or replication subnet you're looking for. Although each person is just trying to do his or her job, nothing in system administration is ever clear-cut enough to fall into just one domain.

An issue IDS administrators have faced in the past is getting **crontab** command entries created for tasks that must be executed on a regular basis. Super-user–level permissions or a super-user–entitled user was required to create the entries, and most IDS administrators in larger enterprises have never had that access level. With the SQL Administration API comes a new, built-in database scheduler that can execute a broad range of functionality from the data server without requiring super-user access to operating system schedulers.

With the database scheduler, you can create "jobs" in four categories:

- Tasks — work that needs to occur when triggered
- Sensors — information gathering
- Startup tasks — work that must be performed on instance start
- Startup sensors — tasks that collect and store information at instance start

Tasks can be simple or complex SQL statements or a user-defined routine (UDR) written in C or Java or as a stored procedure. A sensor is a task whose function is to gather and save information automatically. For example, you can build a sensor to capture virtual processor (VP) and I/O activity at regular intervals so you can do a historical trend analysis; then after tuning, the sensor can capture more results to measure the impact of the changes.

All jobs to be executed through the database scheduler are created using SQL to insert a string and variables into the appropriate **sysadmin** table.

Granular Installation and the Silent Installer

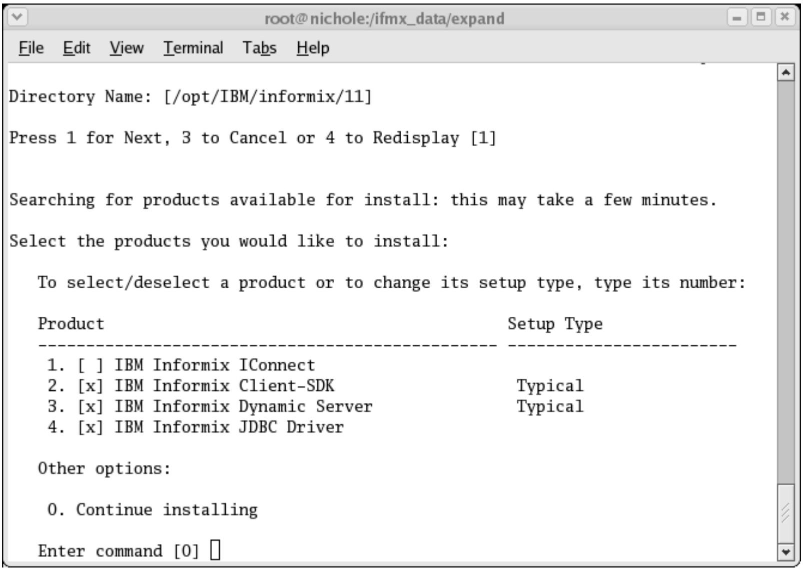
One of the great things about IDS customers is the variety of ways in which they use the technology. The uses range from what some might consider "basic" online transaction processing (OLTP) applications except that they run against very, very large data sets to applications that push the performance characteristics to the utmost. Some customers use replication to protect against failures; others just need to get data from Server A to Server B on a timely basis. Still others need and use sophisticated data models with complex relationships between attributes in the database. The point is that no two customers are alike, and because each has different requirements, they should be able to tailor the functionality and installed footprint of the data server to match those needs.

With IDS's new deployment wizard and its associated silent installer, you now have the ability to tailor which components of IDS are installed to suit your situation.

With this capability, you can install just the base server functionality with its 63 to 112 MB footprint (operating system and hardware dependent) or all options, requiring close to 290 MB. When you select components to install, the wizard displays the storage space required for each option. In addition, the look and feel of the installation process has been standardized for all ports of IDS.

You can invoke the Java-based installer in one of three modes:

- Silent — This mode runs completely unattended, without any administrative intervention to manage. There is no ability to configure the installation as it occurs. You can use a control **.ini** file to "replay" a desired installation footprint.
- Console — This mode provides a command-line–driven interface with multiple interactions, letting you configure the installation as it occurs. Figure 3.1 shows the console interface.
- Graphical — This mode uses a graphical interface to manage administrative interaction to configure the installation.


root@nichole:/ifmx_data/expand
File Edit View Terminal Tabs Help

```
Directory Name: [/opt/IBM/informix/11]

Press 1 for Next, 3 to Cancel or 4 to Redisplay [1]


Searching for products available for install: this may take a few minutes.

Select the products you would like to install:

   To select/deselect a product or to change its setup type, type its number:

   Product                                          Setup Type
   ------------------------------------------------ ------------------------
    1. [ ] IBM Informix IConnect
    2. [x] IBM Informix Client-SDK                   Typical
    3. [x] IBM Informix Dynamic Server               Typical
    4. [x] IBM Informix JDBC Driver

   Other options:

    0. Continue installing

   Enter command [0]
```

Figure 3.1: Console interface to the IDS deployment wizard

You can install IDS components using any of these three modes at two levels: bundle and product. A bundle install is one in which, with one command (**ids_install** for Unix/Linux or **setup.exe** for Windows), all products are installed (IDS data server, Client Software Development Kit, ICONNECT, and so on). At this time, you can't "record" this installation to replay via the silent install mode. On a product level, you install each product individually; so you would execute **installserver** to install IDS, execute **installclientsdk** for the CSDK, and so on.

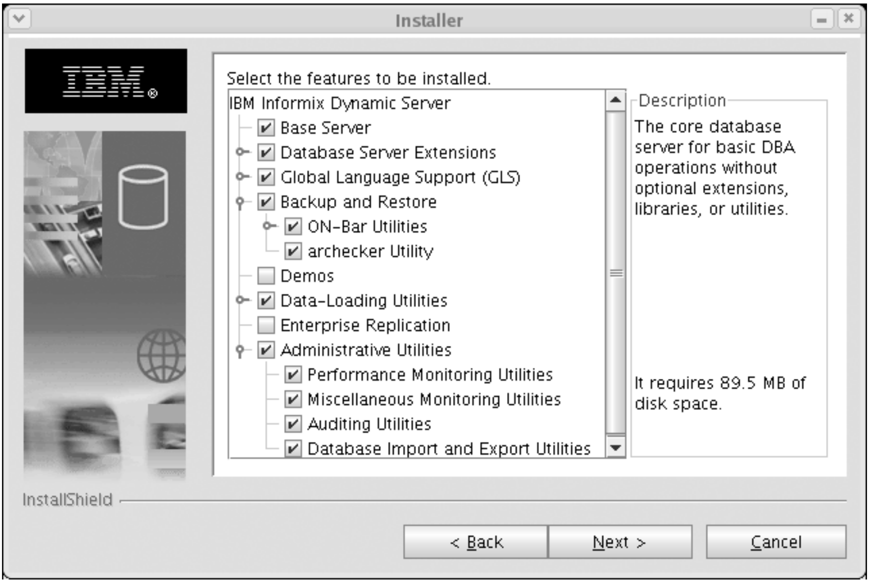
In the graphical or console mode of a product-level installation, an optional parameter can "record" all actions taken into a separate configuration file for each product. Once you have these files, you can include them with the distribution of the data server and "replay" the install on any number of servers to faithfully re-create the installation wherever necessary without any local interaction. This method requires you to execute separate installs for each product, however.

It's possible to create a bundle-level configuration file, but the process isn't automated at this time. You'd need to copy the **bundle.ini** file found in the top level of the expanded IDS distribution and then modify the copied file, turning on or off options as needed. For the most part, each option is controlled with a binary response. For example, the commands

```
-P csdk.active=false
-SP SERVER/IIF.jar IDS-EASTEURO.active=true
```

turn off the installation of the CSDK but do install the Eastern European language libraries for Global Language Support (GLS). Each option in the **bundle.ini** file is briefly explained so you know what you're turning on or off from an installation perspective. Once all the products are installed (whether via a bundle- or product-level installation), additional unattended configuration can occur using scripts provided as part of the application installation.

As I mentioned previously, the wizard provides the ability to customize which components are installed, as shown in Figure 3.2.



Select the features to be installed.

IBM Informix Dynamic Server
- ☑ Base Server
- ☑ Database Server Extensions
- ☑ Global Language Support (GLS)
- ☑ Backup and Restore
  - ☑ ON-Bar Utilities
  - ☑ archecker Utility
- □ Demos
- ☑ Data-Loading Utilities
- □ Enterprise Replication
- ☑ Administrative Utilities
  - ☑ Performance Monitoring Utilities
  - ☑ Miscellaneous Monitoring Utilities
  - ☑ Auditing Utilities
  - ☑ Database Import and Export Utilities

Description

The core database server for basic DBA operations without optional extensions, libraries, or utilities.

It requires 89.5 MB of disk space.

Figure 3.2: Graphical interface to the deployment wizard

If you don't require functionality such as Enterprise Replication or the ability to create a demonstration database, you can use the deployment wizard to remove these features from the installation, as shown in the figure. Many options permit even finer levels of granularity, as evidenced by the "twisties" on the left side of the selection list.

As the installation is executed, the install process creates a manifest file of all installed modules. You can add or subtract modules at any time, and the wizard will use the manifest to determine which modules need to be installed or removed without affecting the rest of the installation. The manifest is also used to validate package dependencies and verify that uninstallation occurs as it should.

Backup and Restore Enhancements

IDS provides two backup and recovery utilities. The first, **ontape**, is the workhorse of IDS environments around the world. An administrative-directed, tape-oriented utility, **ontape** is simple to configure and use, with very few options. The second utility suite, **ON-Bar**, includes the **ON-Bar** API (IDS's implementation of the X/Open Backup Services API, **XBSA**) and a limited-functionality tape management system. You're not required to use the bundled storage library with the **ON-Bar** API; you can schedule and manage regular and logical log backups from an enterprise storage management package such as Tivoli StorageManager. This section discusses enhancements made to both backup and restore utilities to improve their functionality for today's needs.

Full Backup Parallelism

Full backup parallelism is an enhancement to backups created through the **ON-Bar** API to either the bundled storage manager or a third-party manager.

Before starting an **ontape** or **ON-Bar** backup, a checkpoint flushes all dirty pages to disk so the databases are logically consistent. Using comparisons between the backup timestamp and page timestamps, the data server streams the appropriate pages from disk to the backup utility. **Ontape** backups are serial in nature, starting with chunk 0 (the root dbspace) and proceeding through the rest of the chunks in chunk creation order.

With **ON-Bar**, you can create backups either serially or "in parallel." When executed serially, the backups behave the same way as an **ontape** backup. When they're created in parallel, the utility creates a backup timestamp for each chunk as it's streamed to disk. This feature permits multiple chunks to be backed up simultaneously across several backup devices without excessive interference from checkpoints. A side effect of this process is the requirement to have all applicable logical logs available during recovery operations. The logs are necessary to roll forward all chunks to a single point of logical consistency because each chunk will have an earlier or later backup timestamp, reflecting different moments of consistency. This logical roll-forward operation could take some time, depending on the number of logs that needed to be replayed.

In IDS 11, IBM has improved the backup algorithm for parallel **ON-Bar** backups so that parallel backups are executed with a single timestamp for all chunks. With this single timestamp, full recovery doesn't require logical log roll-forward operations. As a result, you can use as many backup devices as you'd like to back up your IDS environments and be able to get online more quickly should you ever require a full instance recovery.

dbspace Ordering

In another enhancement to the **ON-Bar** utility, if enabled the data server will intelligently order the dbspaces sent to the backup devices in an effort to reduce the total time required to back up (and conversely restore) an entire instance.

IDS has always supported "hot" backups while its instances have been online processing user operations, but there was some cost to this process in terms of I/O. Granted, the impact was minor, but it was there. In some customer operations, the volume and response-time requirements are such that even the little impact of a backup operation could affect the operational requirements. As a result, such customers have conducted their backups during lower-use periods to minimize the impact of a backup. With today's business environment operating all day, every day, finding an "off-hours" period is getting harder. Backups need to be executed more quickly to further reduce any impact, even the most minimal, on instance operations.

As with **ontape** operations, **ON-Bar** used to start parallel backups at chunk 0 (zero) and move through the chunks in creation order. If three backup devices were configured, three chunks at a time were backed up at the same time. If you had a mixture of small and large chunks, it was entirely possible for the operation to back up two or more larger chunks at once, causing all the others to wait.

With the new version of the server, you can set the **BAR_MAX_BACKUP** parameter to a value greater than **0** (zero), and the instance will reorder how chunks are sent to the backup devices to reduce the total time of the backup operation. For the ordering to be as effective as possible, the value of this parameter should correspond to the number of available backup devices. If you neglect to specify a value here, the system default of **4** is used.

Figure 3.3 illustrates how this new feature will be of value.

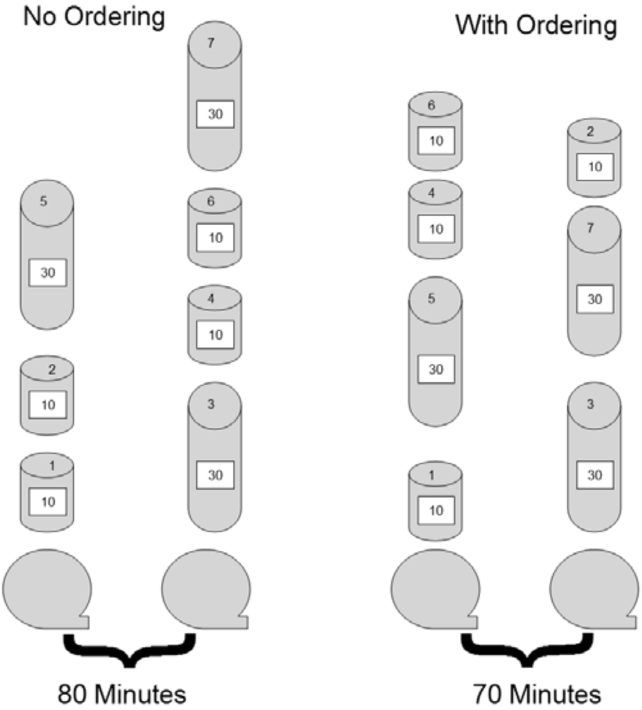


Figure 3.3: An example of dbspace ordering during backup

In this rather simple example, there are seven spaces to back up. Without ordering, the spaces are spooled in creation order with a large space at the end. With ordering, the largest spaces are backed up first (after the rootdbs, which is always backed up first) followed by the smaller spaces. In this case, the backup operation is completed 10 minutes faster. As the number of spaces and the size of the spaces increases, reordering the dbspaces will yield greater and greater dividends.

On-Bar Performance and Monitoring Tool

ON-Bar, as I mentioned, includes an API and the bundled storage manager. With it, you have the flexibility to use almost any major third-party backup management system to back up your instances and logical logs. Although the utility has provided some limited debugging functionality, it was assumed you'd use tools that accompanied your backup management software to resolve any issues that arose. Although those tools worked, they couldn't give you any information about what was happening inside the data server when a backup or restore operation was occurring.

In IDS 11, the new **BAR_PERFORMANCE $ONCONFIG** parameter enables capturing the following information:

- total time spent in **XBSA** calls
- total time spent in archive API calls
- time spent transferring data from/to the **XBSA** API
- time spent transferring from/to **ON-Bar** to IDS
- amount of data transferred through the **XBSA** API
- amount of data transferred to/from **ON-Bar** and IDS

With this information, you'll be able to monitor and isolate any perceived performance problems when using **ON-Bar** and your storage manager.

There's more to this feature, though, because early in IDS 7, when **On-Bar** was first released, the utility suite included an undocumented "null" XBSA storage facility. It was used as a debugging tool for IDS development during the creation of IDS's implementation of the XBSA specifications. I found out about this facility back in the dark ages when I first tried to implement **ON-Bar** at the company I was working for. I couldn't get IDS and the third-party backup management system to connect and work together. IDS advanced support showed me how to use the functionality; with it, I was able to isolate a bug in the backup software that the vendor then fixed.

In IDS 11, the null facility is being documented and made available for your use. With it, you can create a "fake" backup to test the speed at which data flows through the XBSA interface to the backup management system. This technique differs from the **onbar -b -F** option, which sets some flags in the instance's reserved pages. When you use the null facility, a backup will be executed with whatever conditions you set — backup level, serialized, parallel, all spaces, or a subset thereof. Data will be read from disk and passed through the entire **ON-Bar** path until it gets passed out the **XBSA** API. At this point, the data will be routed to a **/dev/null** equivalent. With this functionality, you can determine how long the data server portion and the backup management system take to execute a backup.

The last part of this enhancement is the ability to transfer objects to and from the backup management system and the data server using the new **onbar -S** (capital "s") command option. For more information about this flag and the objects you can transfer between the backup management system and the data server, consult the IBM Informix Backup and Restore Guide.

Backup to Directories

As I mentioned in the introduction to this section, the **ontape** utility has been around since the beginning of the IDS data server. It was designed to write directly to tape and assumed full control over the tape device. Administrator interaction was required to regularly change tape cartridges and respond to **ontape**'s prompts.

In today's environment, where disk drives are so inexpensive, many customers are now directing their **ontape** backups to directory rather than tape. This approach requires managing file names and ensuring previous backups aren't overwritten.

Enhancements in IDS 10 — specifically, the ability to output to standard I/O (**STDIO**) — made backing up to disk easier to manage. You could, for example, output the backup to **STDIO** and then use O/S "pipes" to send the backup to a compression utility and then through another pipe to an archive utility such as **tar** or **cpio** with intelligently generated file names. Even with this functionality, more could be done to make the process of handling **ontape** backups easier.

With the latest version of IDS, **ontape** now explicitly supports backups to a directory and will dynamically manage file naming for you. Figure 3.4 shows the default behavior for this type of backup.

```
nichole_1_L0
nichole_1_20070315_173223_L0
nichole_1_20070316_173244_L0
nichole_1_20070317_173413_L0

nichole_1_L1
nichole_1_20070318_173502_L1

nichole_4_L0
```

Figure 3.4: Default file-name convention for ontape instance backups to directory

In the figure, you can see a series of Level 0 and Level 1 backup images created on a physical server named "nichole." The first numeric value after the physical server name is the **SERVERNUM** of the instance. In this case, two instances are backed up to this directory, one with a **SERVERNUM** of 1 and the other with a **SERVERNUM** of 4.

This functionality intelligently handles renaming existing backup file images as new images are created. The latest backup for each level contains the server name, **SERVERNUM**, and the level number (e.g., **nichole_1_L0**). As a new backup image for that level is created, the previous image is renamed to include a date and timestamp indicating when that image was created. With this support, you can select any previous image, rename it, and use it to restore from.

You can also back up logical logs to a directory, as shown in Figure 3.5.

```
nichole_4_Log0000000006
nichole_4_Log0000000007
nichole_4_Log0000000008
nichole_4_Log0000000009
```

Figure 3.5: Default file-name convention for ontape logical log backups to directory

You can override the stem of the file name with the **IFX_ONTAPE_FILE_PREFIX** environment parameter to meet your company's naming conventions. With this support, your backup directory could contain files named as shown in Figure 3.6.

```
tagus_L0
tagus_20070315_173223_L0
tagus_20070316_173244_L0
tagus_20070317_173413_L0

tagus_L1
tagus_20070318_173502_L1

odra_L0
```

Figure 3.6: Impact of the IFX_ONTAPE_FILE_PREFIX parameter on instance backups to directory

Best practices demand a different prefix for each instance, so the **SERVERNUM** isn't included in the file names.

Unattended ontape Backup and Restore

If you use **ontape** to back up or restore to or from a directory, you can now eliminate the "Press return to continue" administrative prompts and have the utility execute in hands-free, near-silent mode. I say "near-silent" because it will still output the usual status messages as the backup is executed.

During a restore operation, **ontape** isn't silent, nor is it completely hands-free. Although many of the default prompts are answered automatically, you'll need to interact with the utility to indicate whether you want to salvage the logical logs, specify which logs to salvage, and so on. As with backups, the utility displays the full set of status messages during a restore. Because of the need to respond to the prompts, it wouldn't be prudent to redirect the output of a restore operation to **/dev/null**.

Backup and Restore Filters

This feature applies to both **ontape** and **ON-Bar** and is part of the groundwork being laid for more extensive on-disk encryption functionality to follow in the future.

Most encryption systems work as close to the data as possible. As data flows in and out of the data server, functions are executed to make the necessary transformations. One side effect of this architecture is that data passed to utilities such as **ontape** and **ON-Bar** has been decrypted. As a result, backups are created and stored "in the clear," which for many industries does not comply with security or other regulations.

With this release, you can use encryption or compression "filter plugins," which execute inline between the data server and your backup utility of choice. You can think of this feature as the Communication Support Module (CSM) equivalent for backups. Several releases ago, you could use CSMs to encrypt communications between the data server and client applications to prevent network sniffers from capturing sensitive data as it moved back and forth. The backup plugins perform the same functionality for **ontape** and the **ON-Bar** API.

Four new **$ONCONFIG** parameters have been added to support this functionality: **BACKUP_FILTER** and **RESTORE_FILTER** point to the executable to be used during backup or restore operations. **BACKUP_FILTER_OPTION** and **RESTORE_FILTER_OPTION** are used to pass any program options to the executables. Possible options could include encryption key(s), block size, and other values.

As a side note, although the primary function of this feature is to provide encryption, you can use a compression filter instead to reduce the amount of media required to create your backups.

For more information about the new parameters and how to implement this feature, see the IBM Informix Backup and Restore Guide.

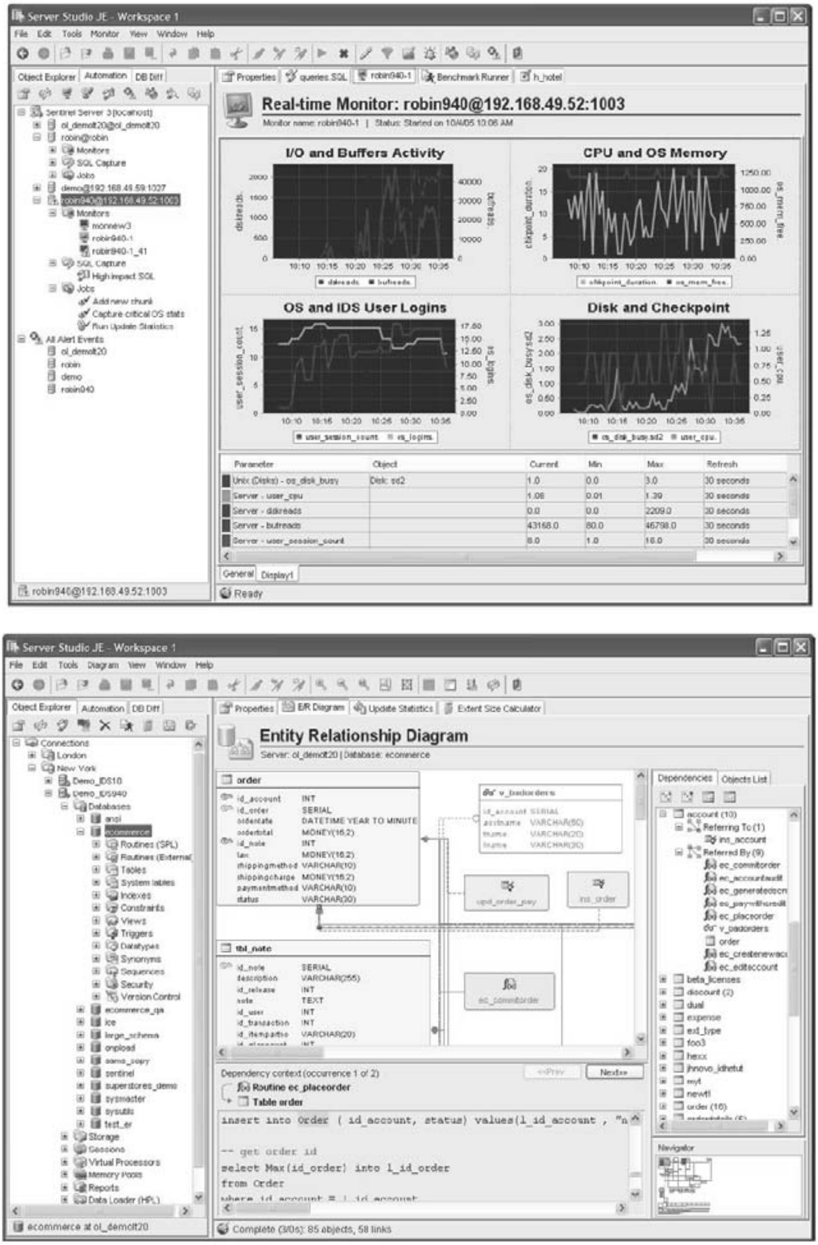
OpenAdmin Tool for IDS (previously known as IDSAdmin)

Building on functionality provided by the SQL-based administration interface, a new graphical administration console has been developed and is included with this port of IDS. Called *OpenAdmin Tool for IDS*, the console was written in PHP (PHP: Hypertext Preprocessor), and plans are to provide hooks so you can add modules or functionality as your environment requires. Before covering this new tool, let's take a look at the administration utilities currently available with IDS.

As I noted at the beginning of this chapter, the heritage of the IDS data server is distinctly Unix-oriented, and, as a result, most administration utilities have been and continue to be command-line oriented. They include

- **onparams** — Used to add or remove logical logs, resize and/or relocate the physical log, and create or change buffer pool settings.
- **onspaces** — Focuses (as the name implies) on data storage space administration, including creation and deletion of chunks, dbspaces, and BLOBspaces (both smart and simple); starting and stopping IDS-based mirroring; and so on.
- **oninit** — Used to initialize and/or start the instance from an offline mode or to switch operating modes while the instance is operational.
- **onmode** — Used to shut down an operating instance or to change the operating mode if the instance is already partially shut down; also used to instantiate and manage High-Availability Data Replication (HDR)–based replication and related functionality and to manage shared memory changes.
- **oncheck** — Used to check and verify the consistency of table and index data pages.
- **onstat** — The queen of all IDS utilities, **onstat** can be used to interrogate almost anything happening in the instance at any moment in time. It feels as if there are 500 command flags for this utility, which enables you to debug and troubleshoot any problem in the instance.

Unfortunately, some people can't manage an environment without some sort of graphical interface, so several utilities have been created. AGS, Ltd., an IBM IDS partner, created ServerStudio Java Edition (SSJE) and continues to enhance and add functionality to this utility on a regular basis. Figure 3.7 shows two sample screens presented by this tool.



Real-time Monitor: robin940@192.168.49.52:1003

Monitor name: robin940-1 | Status: Started on 10/4/05 10:06 AM

	Parameter	Object	Current	Min	Max	Refresh
■	Unix (Disks) - os_disk_busy	Disk: sd2	1.0	0.0	3.0	30 seconds
■	Server - user_cpu		1.06	0.01	1.39	30 seconds
■	Server - dskreads		0.0	0.0	2209.0	30 seconds
■	Server - bufreads		43168.0	80.0	46798.0	30 seconds
■	Server - user_session_count		8.0	1.0	16.0	30 seconds

General Display1

robin940@192.168.49.52:1003

Ready

Entity Relationship Diagram

Server: ol_demo20 | Database: ecommerce

order

id_account	INT
id_order	SERIAL
orderdate	DATETIME YEAR TO MINUTE
ordertotal	MONEY(16,2)
id_note	INT
tax	MONEY(16,2)
shippingmethod	VARCHAR(10)
shippingcharge	MONEY(16,2)
paymentmethod	VARCHAR(10)
status	VARCHAR(30)

tbl_note

id_note	SERIAL
description	VARCHAR(255)
id_release	INT
note	TEXT
id_user	INT
id_transaction	INT
id_itempartno	VARCHAR(20)

v_badorders

id_account	SERIAL
acctname	VARCHAR(50)
fname	VARCHAR(20)
lname	VARCHAR(30)

upd_order_pay

ins_order

ec_commitorder

Dependency context (occurrence 1 of 2)

Routine ec_placeorder

Table order

```
insert into Order ( id_account, status) values(l_id_account , "n

-- get order id
select Max(id_order) into l_id_order
from Order
```

ecommerce at ol_demo20

Complete (3/0s): 85 objects, 58 links

Figure 3.7: Two screen shots from the ServerStudio Java Edition utility

Originally designed as a database administration tool, SSJE now contains functionality to manage the instance as well as generate performance and other historical data. While the base functionality of SSJE is bundled with the data server, access to more advanced functionality — such as schema comparisons and debugging, schema version control, dependency analysis, entity relationship diagraming, Stored Procedure Language (SPL) debugging, performance statistics gathering, and histogram generation — requires purchasing a license for that module from an authorized reseller. It would serve you well to look at the additional functionality available for purchase. It is very good.

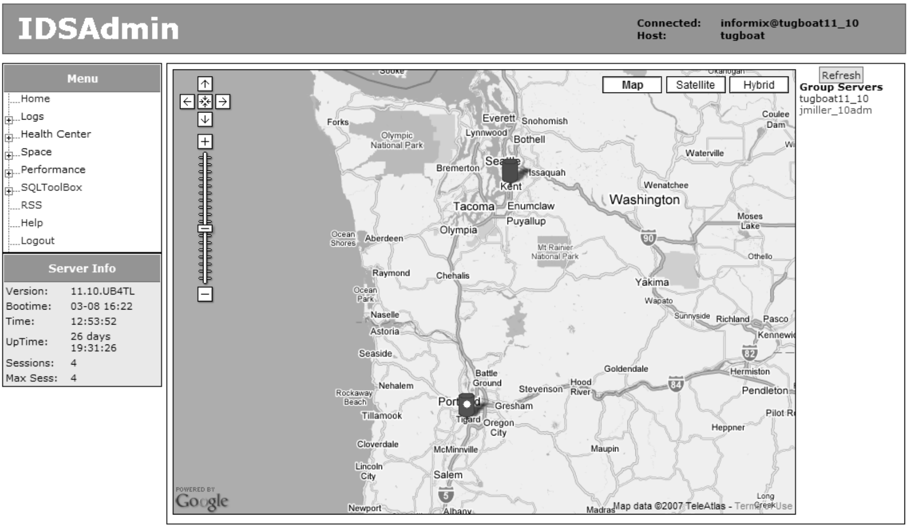
Another relatively new tool, called DBSonar, is produced by Cobrasonic, another IBM IDS partner. At present, DBSonar focuses on performance management and provides many powerful features, enabling you to isolate potential areas of concern within your system. Plans call for a broader range of functionality, so you'd be wise to examine this product as well. Unlike SSJE, no part of DBSonar is included as part of the IDS distribution, but you can buy the tool from an authorized reseller.

With that background, let's look at the OpenAdmin Tool for IDS utility. Originally developed as a skunk-works project, OpenAdmin Tool for IDS was designed to help database administrators (DBAs) answer the most commonly asked questions and perform the most commonly executed tasks. Written in PHP and working in conjunction with an Apache (or similar) Web server, a single install of OpenAdmin Tool for IDS can administer as many instances as you'd like. No client-side (instance level) installation is required. It supports full globalization, and the default modules can easily be converted to your or another native language through the use of translation tables. Finally, in a first for an IDS utility, OpenAdmin Tool for IDS has an RSS plug-in that can feed you realtime updates based on rules you set in the utility.

Given the flexibility of the PHP-based architecture, you can add quite a bit of additional functionality. As Figure 3.8 illustrates, you can even create objects such as map mashups to identify all your instances and their locations. The status of each instance can be identified with a color code that indicates the overall health and welfare of the instance, letting you tell at a glance whether you need to examine an instance. As a side note, both AGS and Cobrasonic plan to leverage this flexibility and the administration API mentioned earlier in this chapter to quickly and easily add more functionality to their products.

The OpenAdmin Tool for IDS utility contains tasks and sensors to gather and display information as well as execute jobs. Figure 3.9 shows part of the OpenAdmin Tool for IDS tasks and sensors functionality.

As the name implies, a task is a specific job executed at a specific time. A task can be a simple or compound SQL statement or a UDR written in C, Java, or SPL. Some tasks, called sensors, can collect information. Sensors assist the DBA because he or she simply needs to set up the sensor once; it will continue to gather information at the requested intervals without further intervention.



IDSAdmin

Connected: informix@tugboat11_10
Host: tugboat

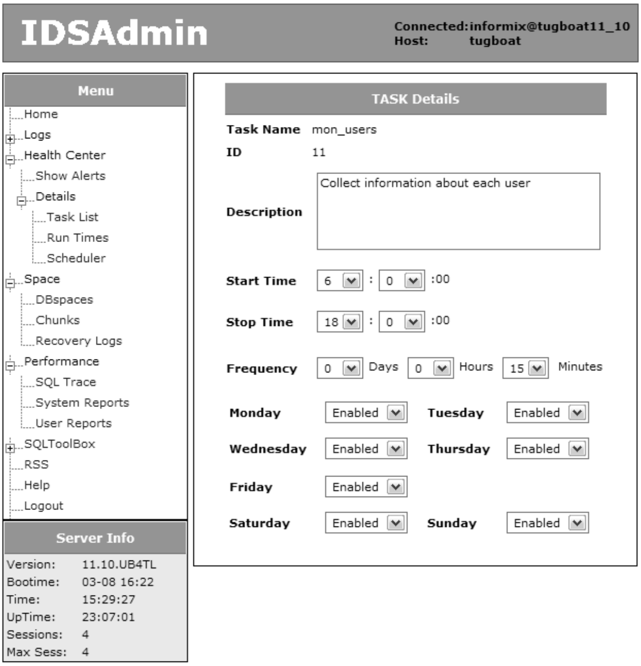
Menu
- Home
- Logs
- Health Center
- Space
- Performance
- SQLToolBox
- RSS
- Help
- Logout

Server Info

Version:	11.10.UB4TL
Bootime:	03-08 16:22
Time:	12:53:52
UpTime:	26 days 19:31:26
Sessions:	4
Max Sess:	4

Refresh
Group Servers
tugboat11_10
jmiller_10adm

Figure 3.8: Opening screen of the IDSAdmin utility



IDSAdmin

Connected: informix@tugboat11_10
Host: tugboat

Menu
- Home
- Logs
- Health Center
  - Show Alerts
  - Details
    - Task List
    - Run Times
    - Scheduler
- Space
  - DBspaces
  - Chunks
  - Recovery Logs
- Performance
  - SQL Trace
  - System Reports
  - User Reports
- SQLToolBox
- RSS
- Help
- Logout

Server Info

Version:	11.10.UB4TL
Bootime:	03-08 16:22
Time:	15:29:27
UpTime:	23:07:01
Sessions:	4
Max Sess:	4

TASK Details

Task Name mon_users
ID 11
Description Collect information about each user
Start Time 6 : 0 :00
Stop Time 18 : 0 :00
Frequency 0 Days 0 Hours 15 Minutes

Monday	Enabled	**Tuesday**	Enabled
Wednesday	Enabled	**Thursday**	Enabled
Friday	Enabled		
Saturday	Enabled	**Sunday**	Enabled

Figure 3.9: Part of the IDSAdmin tasks and sensors functionality

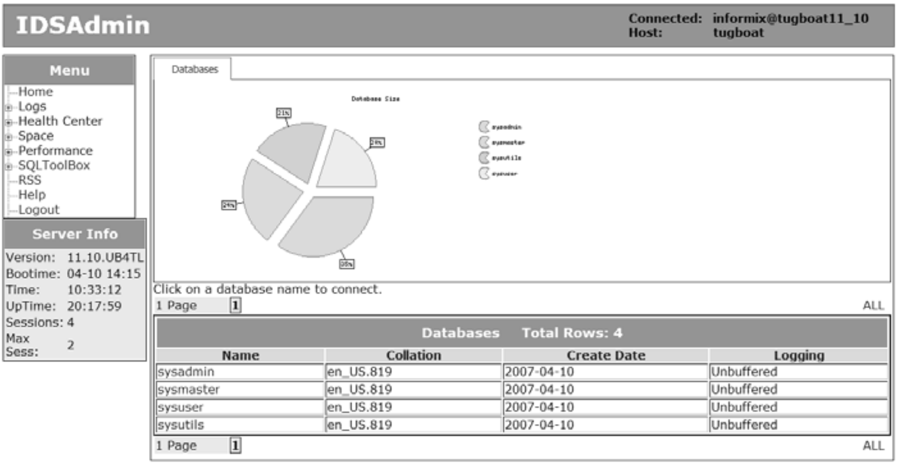
You can set up some tasks and sensors to execute on instance events, such as startup or shutdown. Other tasks can be configured to execute in response to an instance condition, such as when all logical logs are full or a dbspace runs out of room.

Given this capability, you can use OpenAdmin Tool for IDS as a partial replacement for **ALARMPROGRAM** functionality.

A variety of tasks and sensors are bundled into OpenAdmin Tool for IDS, giving you a broad range of functionality. You can also use the prebuilt tasks and sensors as templates to create your own tasks and sensors. These functions

- manage the command history table
- capture and save any changes made to the instance's **$ONCONFIG** file
- can save all $**ONCONFIG** files for all managed instances so you can restore any instance in the event of corruption or loss
- collect virtual processor timings and usage statistics
- collect checkpoint information
- collect information about all users who connect to the instance
- check to ensure backups are executed on a regular basis
- collect table name and profile information

Additional functionality includes a Health Center with alert information color-coded to indicate the severity of the alarm or alert, along with recommendations on resolving the issue. The Logs area lets you view instance and **ON-Bar** logs to check for error reports. The Space section of the utility provides dbspace and table usage information at a much more granular level than provided by the previous IDS-developed graphical utility. Figure 3.10 shows one of the many OpenAdmin Tool for IDS snapshot screens.



Connected: informix@tugboat11_10
Host: tugboat

Server Info

Version: 11.10.UB4TL
Bootime: 04-10 14:15
Time: 10:33:12
UpTime: 20:17:59
Sessions: 4
Max Sess: 2

Click on a database name to connect.

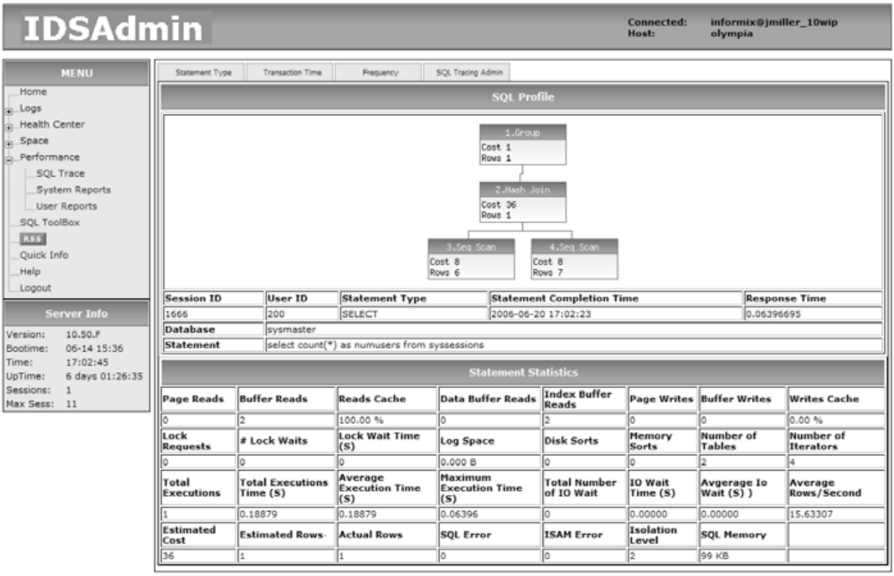
Databases Total Rows: 4

Name	Collation	Create Date	Logging
sysadmin	en_US.819	2007-04-10	Unbuffered
sysmaster	en_US.819	2007-04-10	Unbuffered
sysuser	en_US.819	2007-04-10	Unbuffered
sysutils	en_US.819	2007-04-10	Unbuffered

Figure 3.10: One of the many IDSAdmin snapshot screens

The Performance section is quite rich; you can set up sensors to build comprehensive performance histograms to evaluate the need for additional instance tuning.

After making changes, you can monitor their impact on instance operations. You can track memory and disk statistics as well as lock waits and other conditions. In conjunction with the SQL Toolbox section, you can capture information about the most commonly executed or most expensive SQL operations. You can use the SQL drill-down functionality, illustrated in Figure 3.11, to look at each part of an SQL operation and determine where and why it may be performing poorly.



Connected: informix@jmiller_10wip
Host: olympia

Server Info

Version:	10.50.F
Bootime:	06-14 15:36
Time:	17:02:45
UpTime:	6 days 01:26:35
Sessions:	1
Max Sess:	11

Session ID	User ID	Statement Type	Statement Completion Time	Response Time
1666	200	SELECT	2006-06-20 17:02:23	0.06396695

Database	sysmaster
Statement	select count(*) as numusers from syssessions

Page Reads	Buffer Reads	Reads Cache	Data Buffer Reads	Index Buffer Reads	Page Writes	Buffer Writes	Writes Cache
0	2	100.00 %	0	2	0	0	0.00 %
Lock Requests	# Lock Waits	Lock Wait Time (S)	Log Space	Disk Sorts	Memory Sorts	Number of Tables	Number of Iterators
0	0	0	0.000 B	0	0	2	4
Total Executions	Total Executions Time (S)	Average Execution Time (S)	Maximum Execution Time (S)	Total Number of IO Wait	IO Wait Time (S)	Avgerage Io Wait (S))	Average Rows/Second
1	0.18879	0.18879	0.06396	0	0.00000	0.00000	15.63307
Estimated Cost	Estimated Rows·	Actual Rows	SQL Error	ISAM Error	Isolation Level	SQL Memory	
36	1	1	0	0	2	99 KB	

Figure 3.11: The SQL drill-down functionality of the IDSAdmin utility

Automated and Improved Statistics Gathering

From the beginning, IDS has used a cost-based SQL optimizer to determine data access plans. One disadvantage of this type of optimizer is that its plans can vary from perfect to slightly off to horribly wrong if statistical information about the data in the database (or databases) isn't gathered on a regular basis. To determine how requested data should be queried, joined, or otherwise manipulated to return the desired result, the optimizer needs to know the number of rows in each table, what indexes exist and on which columns, what the data distribution cardinality is, and the number of rows for each set of unique values.

Gathering this statistical information has always been a manual task. Any IDS administrator or DBA knows that the first question asked when trying to help an IDS administrator with a performance problem is always, "When was the last time you ran the **update statistics** command and what options did you use when you ran it?" Recommendations about when to run this command and which options to use have circulated and been updated in the IDS support groups since the data server's beginning in an attempt to help administrators keep the optimizer "properly nourished."

To assist you and other administrators with this task, this release of IDS provides several improvements as to when and how the statistical gathering process works.

First, when an index is created on a table, rather than require you to execute a statistics-gathering operation on the new index, branch and leaf node information is fed to the optimizer during the index build as data is scanned. As a result, when the index build is completed, the optimizer will already have accurate information about the index that it can immediately and efficiently use in creating access plans. The gathered statistics reflect what normally would have been created by executing the **update statistics medium** and **update statistics high** commands on the lead column of the index and executing an **update statistics low** command on the rest of the index. IDS 11 extends this functionality to indexes created on explicitly created temporary tables as well.

As with almost anything, there is the exception that proves the rule, and so there are a couple of cases where automated statistics gathering does not occur. There is an undocumented environment variable, which I won't give you, that forces a top-down index build. With this type of build, it's not possible to properly gather the statistical information as the index is built. Other cases where full automated statistic gathering during an index build is turned off are

- if the lead column in the index is a user-defined type (UDT), because this also forces a top-down index build
- if a table contains less than two rows — because it's a little hard to generate statistics on an empty table!
- if the index is created using the data server's Virtual Index Interface (VII) to access non-relational or non-IDS–managed data such as that stored in smart BLOBspaces or in flat files outside the instance
- if a functional index is created using the results returned from a function

Index statistical information is updated automatically when an index is created on a table with fragmentation and the fragmentation scheme is changed or when a unique index is "promoted" to a primary key or unique constraint. See my other publications for information about the behavioral differences between a unique index and a unique constraint or a primary key and why it's much better to build indexes that you then promote to the final desired state.

The next improvement to the **update statistics** process is the ability to more precisely define the sampling size when executing an **update statistics medium** command. When you use the medium level, not only does the command capture the number of rows in the table (or index); it also gathers information about the distribution of data, including the range of values and the approximate row count within each range. In IDS 10 and earlier, not every row in a table was read to generate this distribution information; instead, a statistically random number of rows was used. As a result, the returned values could vary from one execution to another depending on which rows were sampled.

You could change two parameters, **resolution** and/or **confidence**, to modify the number of sampled rows to increase or decrease the accuracy of the distribution set. Although this step helped, you still didn't have full control over the process. That notwithstanding, the defaults were pretty good; they generated an approximate statistical accuracy of 80 percent compared with that generated when each row is read, as happens when an **update statistics high** is executed.

As you might expect, generating statistics with the **medium** or **high** option requires more time than the **low** option. Although constant improvements have been made to the **update statistics** command over the years, many customers wanted more control over the sample size and the corresponding creation time required to generate optimizer statistics. In response, a new "sample size value" option has been added to the **update statistics medium** syntax, as shown in Figure 3.12.

```
update statistics medium for table my_tab (col1, col2)
sampling size .75 distributions only;

update statistics medium for table my_tab_2 (col4)
sampling size 5000;

update statistics medium for table my_tab_3 (col1)
sampling size 10000 resolution 1 0.95;
```

Figure 3.12: Examples of the new update statistics medium syntax

The figure shows several examples of how to use the sampling size syntax. The value immediately following the new syntax determines the minimum number of rows to sample. The number of rows sampled is either a percentage of total rows or an actual row count as follows:

- If the value is greater than **1** (one), it is interpreted as the number of rows to be sampled. In the second and third commands shown in the figure, that means 5,000 and 10,000 rows, respectively.
- If the value is less than or equal to **1** (one), it is interpreted as the percentage of rows (up to 100 percent) to be read to generate the statistical information. In the first command shown, 75 percent of the rows are to be sampled.

Following the sampling size option, you can still include **resolution** and **confidence** parameters to further refine the result set if you desire. Without these parameters, the default statistical accuracy is still 80 percent. As a result, if the 5,000 rows to be sampled aren't enough to meet the minimum accuracy measurement, more will automatically be read. If 5,000 is more than enough to meet the objective, 5,000 rows will still be read because the optimizer assumes you want a higher degree of accuracy based on your explicit setting of the rows to be read.

With the resolution and confidence values in the third command string of the figure, the administrator is specifying that at least 10,000 rows must be read to achieve an accuracy percentage of 95 percent. At least 100 ranges (or statistical "buckets") are to be created in which row counts are stored.

Information about the sampling size, as well as when and how **update statistics** commands have been run, is now stored in new columns and tables in the **sysmaster** database. Consult the data server documentation for more information about this data.

The next enhancement to the statistics-gathering process focuses on temporary tables. From an optimizer perspective, these tables have always been treated like regular tables. Once a temporary table is created and populated, you need to execute an **update statistics** operation on the table for the optimizer have the information it needs to build access plans using the temporary table. This fact required application developers to include these commands whenever they explicitly created temporary tables. Because developers are, for the most part, not as versed in data server operations as a DBA or an IDS administrator, they commonly forgot this step. Performance suffered when these tables were used. Invariably, the data server was blamed and, unless the DBA or IDS administrator performed a code walkthrough, the problem was rarely found or fixed.

The IDS 11 release remedies this problem. Now, the number of rows and pages (the equivalent of an **update statistics low** command) are updated whenever the temp table is accessed. As I already mentioned in this section, when an index is created on a temp table, statistical information will be gathered as the index is built, similarly to indexes built on "regular" tables, provided the indexed columns don't fall into the exclusion list given earlier.

The last improvement in statistics management occurs when you execute the **dbschema** or **dbexport** utility. Previously, when you used the **-hd** flag in invoking **dbschema**, only the resolution and confidence values were returned. Now, the results include the sampling size parameter, as you can see near the top of Figure 3.13.

```
DBSCHEMA Schema Utility       INFORMIX-SQL Version 11.UC1
Distribution for informix.customer.customer_num
.
.
Constructed on 2007-03-22 18:03:39.00000
Medium Mode, 0.250000 Sampling Size, 2.500000 Resolution,
0.800000 Confidence


{ DATABASE stores  delimiter | }
grant dba to "Informix";
grant connect to "public";
.
.
update statistics medium for table stock (description,
manu_code, stock_num, unit, unit_descr, unit_price)
sampling size .25 resolution   2.50000   0.80000 ;
```

Figure 3.13: New information in the dbexport and dbschema utilities

In the second example in the figure, you can see that when exporting a database using the **-ss** flag to the **dbexport** utility, full statistical information is published, including the sampling size.

Recovery Time Objective and Interval Checkpoints

Although they are two unique new features, the subjects of this section work together to provide some very powerful functionality and help extend IDS's lead in the marketplace where autonomic management is concerned.

IDS administrators have traditionally had to work around two issues they couldn't control well: how long it takes an instance to restart if shut down abnormally and the length of time the instance suspends operations during a checkpoint. During the course of normal instance operations, IDS persists a lot of "state" information to disk about ongoing transactions. When an IDS instance isn't shut down properly — for example, if the physical server experiences a failure or an administrator kills the wrong Unix process — this transaction state information is used to recover the instance to logical consistency as close to the failure time as possible. This process, called "fast recovery," involves replacing data pages on disk with "before" images kept in the physical log. Once this step takes place, transactions are replayed based on logical log entries from the last checkpoint to the failure point. Transactions that were started but never completed are reversed so that by the end of the fast recovery operation, all committed transactions are properly written to disk and the database (or databases) are in a consistent state.

The amount of time required for the fast recovery to be completed depends on how long the instance operated after the last checkpoint before the failure occurred and on the volume of transactions recorded in that period of time. The greater the volume of transactions, the longer the time to recover — potentially lasting *well* into the minutes. If the system didn't have HDR or some other failover mechanism instantiated, user operations were affected even if the failure was transient and the physical server and instance could be immediately restarted.

Looking at checkpoints and their duration for a moment, checkpointing is the process wherein all committed data is persisted to disk to clear dirty memory buffers and to increase recoverability in the event of uncontrolled shutdown. Currently, during a checkpoint, **insert**, **update**, and **delete** operations are temporarily suspended while this flush occurs. Although this is the most efficient of IDS's mechanisms to write to disk, user operations could be affected for up to a minute or more depending on the volume of data to be written. Depending on the types of operations being executed when a checkpoint occurs, some users might notice more than others. Those executing read-intensive operations wouldn't see much, if any, impact, but those changing data would have to wait for the checkpoint to be completed before receiving a confirmation of their transaction and beginning the next unit of work. It would appear as though the system "was hung."

From an administrative perspective, IDS administrators have been trying to mitigate the impact of transactions in several ways. Some decrease the time interval between mandated checkpoints so that instead of the default five minutes, the checkpoints occur every three minutes or so. Others modify the **LRU_MIN_DIRTY** and **LRU_MAX_DIRTY** configuration parameters, which start and stop the trickle write of data from memory to disk based on the percentage of clean and dirty buffers in the Least Recently Used (LRU) queues.

In IDS 9.4, the **LRU_MIN/MAX_DIRTY** parameters could include a fractional value, so it was possible for an administrator to set **LRU_MAX_DIRTY** to 4 and **LRU_MIN_DIRTY** to 2.5. The instance would begin flushing buffers when 4 percent were dirty and would stop when only 2.5 percent were dirty. Due to playing with the amount of clean and dirty buffers, when a checkpoint occurred there were fewer buffers to flush, and the checkpoint was completed more quickly. Unfortunately, writes based on LRU dirty values aren't as efficiently organized as checkpoint writes and, as a result, have a performance overhead that had to be accounted for in severely constrained systems.

Other options for mitigating the impact of checkpoints included changing the size of the physical and logical log buffers, increasing the number or size of the LRU queues, and so on. For all of these options, a number of conflicting results had to be balanced.

With IDS 11, an improved checkpoint algorithm has been implemented. You can also set an instance recovery time objective, or the maximum amount of time the instance will require to start or restart after a controlled or uncontrolled shutdown. The implementation of both features will remove the last of the "real" IDS tunable parameters from your plate, letting you focus on more important tasks.

Looking first at the recovery time objective, you can set a time value in seconds (1 to 30 minutes) either in **$ONCONFIG** or dynamically using the **onmode** utility. When you set this value, the data server will automatically manage several previously hand-tuned parameters, including the checkpoint interval, **LRU_MIN/MAX_DIRTY**, the number of asynchronous I/O (AIO) VPs, and so on. The data server will constantly monitor activity in the instance and, as activity fluctuates, will make sure the proper amount of dirty pages are flushed to disk so that in the event of a shutdown and restart, the logical roll-forward operation is small enough to be completed in the desired recovery time objective.

As part of this process, two other parameters, **RAS_PLOG_SPEED** and **RAS_LLOG_SPEED**, have been added to **$ONCONFIG**. These values control the volume of transactions that can be processed through the logical and physical logs. Default values are set by the server and are constantly being updated based on real-life instance operations. They are *not* to be manually changed under any circumstances.

A key component to achieving the recovery time objective is the use of an improved checkpoint algorithm, the "interval" checkpoint. Automatically activated when the recovery time objective has been set, the new algorithm combines some of the features of the now deprecated "fuzzy" checkpoint with the original "sync" checkpoint.

When an interval checkpoint is triggered, a checkpoint record is generated. Part of this record is the buffer addresses of all dirty buffers containing committed data to be flushed to disk. The gathering of these addresses is the only "blocking" action in the new checkpoint algorithm. Because it occurs in memory, the block and list generation should not be noticeable at all to end-user operations; the instance is already tracking these buffers. Once the list is generated, **insert**, **update**, and **delete** operations continue processing while the dirty buffers are flushed to disk. Once all the buffers in the list are written to disk, a formal checkpoint record is entered into the instance's reserved pages, and the physical log is purged of the associated "before" images.

One impact of this checkpoint algorithm is that there will be more activity than before in the physical log, and the log will hold more records for a longer period of time. As a result, you'd be wise to increase the size of the physical log. For instances where the buffer pool is relatively small, you should size the physical log to almost 110 percent of the buffer pool. Remember, the actual amount used will be affected by the time set for the recovery objective. As instance operations ebb and flow, the instance will constantly monitor and adjust checkpoint operations, which will clear the buffer pool (or pools) and the physical log. But, as a "belt and suspenders" DBA, I recommend you size the physical log as large as is practical so that the increased activity doesn't trigger a checkpoint due to the fact that the physical log is 75 percent full.

Three other new **$ONCONFIG** parameters affect the new checkpoint algorithm and the recovery time objective: **AUTO_LRU_TUNING**, **AUTO_AIOVPS**, and **AUTO_CKPTS**. You can set or change each of these after instance startup if you desire with **onmode –wm** or **onmode –wf**. For more information about these options and the parameter ranges, check the data server documentation.

Modifying the Physical Log

With the increased activity in the physical log, it's important to manage the log size and location as quickly and as easily as possible. Previously, any changes to the physical log required an instance outage. The instance had to be brought to quiescent mode to change the storage dbspace for, or the size of, the physical log. That is no longer necessary.

You can now make changes to the physical log with the instance online and processing end-user operations using either the **onparams** or **onmonitor** utility or the SQL API. Once the "new" physical log is built, it is immediately activated for use. I *strongly* suggest you immediately create a full instance backup after making any change to the physical log.

sysdbopen/sysdbclose Functions

As I mentioned at the end of the previous chapter, the **sysdbopen** and **sysdbclose** functions can serve several purposes. The primary intent of the feature, as I understand it, was as more of an administrative aid than for security purposes, but you can certainly use the functions for security-oriented operations.

When a session connects to an instance, unless the DBA has used new functionality introduced in IDS 10, all sessions have identical access permissions. The setting of database "roles" must be executed as part of the application, which is problematic to enforce for sessions that connect directly through an SQL reporting tool (e.g., Crystal Reports, Microsoft Excel). With IDS 10, the DBA could create a default role for each user, which helped, but many customers required more functionality for their applications as well as for security reasons.

You can now write a series of UDRs that are executed when a user session connects and disconnects from the instance. These routines can perform any functionality, including setting roles, specifying session operating parameters such as **PDQPRIORITY**, setting the isolation level or the output location of optimizer reports (a.k.a. sqexplain.out files), turning on (or off) monitoring functionality, and sending an alert — in short, whatever you need to do. Unlike other UDRs, you can't call another function, so all required functionality must be completely contained in the open and/or close function.

You can create customized open and close functions for each user and/or the **PUBLIC** group. The default behavior is that they'll be executed when the user connects to or disconnects from the instance. If the **IFX_NODBPROC** environment variable is set to any value (including **0** (zero)) before connecting to the instance, these functions won't be executed. You can use this feature to write and test the function (or functions). Simply write one or more UDRs, set the environment variable, and then connect to the instance to register the function(s). Test the function by calling it via a standard SQL function call. If the function works as needed, unset the environment variable, and the UDR(s) will execute for all future sessions.

Direct I/O with Cooked Spaces

If you've been around the IDS data server for any amount of time, you've undoubtedly participated in or followed one of the endless discussions about which disk format is "better" for an IDS environment: raw (unformatted disk partitions) or cooked (O/S-formatted partitions with flat files). On one side of the discussion are those who (correctly) state that there is somewhere around a 10 percent to 15 percent performance increase (or better) with raw space, possibly more if kernel asynchronous I/O (KAIO) is used. The fact that you must be O/S-knowledgeable to create the partitions (or be able to arm-wrestle into submission the storage administrators to create them for you) is a minor inconvenience.

On the other side of the discussion, flat files to support chunks and dbspaces are easy to create, require no special dispensation from the storage team, and can be backed up with the rest of the system files using a single backup routine (provided the instance is shut down). Add to this the fact that operating systems are getting better at speeding up flat-file throughput, and the arguments on this side of the discussion are becoming compelling as well.

IDS 11 adds a new **$ONCONFIG** parameter, **DIRECT_IO**, that enables the data server to bypass O/S buffers and write directly to flat files used to create chunks and dbspaces. With this feature enabled, you can approach the same performance metrics using cooked space as you can with raw space. The feature is O/S-dependent, so consult the release notes that accompany your distribution of the data server to see whether this enhancement is available for your port.

Instance Administration Mode

When IBM released IDS 10, a huge cheer erupted from IDS administrators around the world — finally, a single user instance operating mode! The feature prevented general end-user access to the instance but provided full connectivity to the **Informix** user ID and members of the **DBSA** group for executing instance or database maintenance tasks. Administrators no longer had to put the instance in online mode, open all the sessions they needed, execute a shutdown to quiescent mode, and then go through killing user sessions that managed to connect. The **-j** flag was added to the **oninit** and **onmode** utilities, which transitioned the instance to this operating mode.

With ever-increasing pressures from the security and regulatory environment, companies are now restricting the access and use of the **Informix** user ID. As a result, a new administration and mechanism is needed. IDS 11 introduces the administrative operating mode, which replaces single-user mode and gives you the ability to define a limited number of user IDs that can access the instance in the new mode.

You have two ways to define user access while in administrative mode. The new **ADMIN_MODE_USERS $ONCONFIG** parameter permits continual access. Flags have been added to the **oninit** and **onmode** utilities, but their definitions are limited to a single instance startup/shutdown cycle.

The **-j** flag for **oninit** and **onmode** will transition the instance to this new operating mode. A **-U** (capital "u") flag has been added to the **oninit** utility to define the user IDs that can continue to access the instance in administrative mode. For example, the command

```
oninit -U jerry,fred,cindy
```

gives the three named user IDs access to the instance if it should be transitioned to administrative mode. The scope of this command operation is just for this startup of the instance, though; once the instance has been completely shut down, the definition is lost.

The **-U** flag to define user IDs has been added to the **onmode** utility as well. When you use it with the **-j** flag, all user sessions except those defined with the **-U** flag are disconnected from the instance and are prevented from reconnecting until the instance is brought back into online mode.

You can execute the **onmode** utility as many times as you need to with these flags, varying the user IDs that are permitted to connect to the instance. The actions are subtractive rather than cumulative, though. For example, if you execute the command four times with different user IDs over a period of time, at the end of each execution only those IDs defined as a part of the command string will be allowed access to the instance. Any other previously permitted user IDs will be disconnected.

Private VP Memory Cache

This feature doesn't really fall into the category of "easing an administrator's burden," but it doesn't really fit in the other chapters either, so I've included it here.

In Chapter 1, I discussed the unique aspects of the Dynamic Scalable Architecture (DSA), including a self-managing shared-memory component. A major element in this memory allocation are the buffers used to hold data that is queried from disk or is in-flight to disk as a result of an **insert** or **update** operation.

Depending on the individual operation being executed, virtual processors must find available space in the buffers to host the data. This process is partially assisted by the FIFO and LRU mechanisms, which are responsible for ensuring data that's not being used is flushed from the buffers so space is available to support ongoing tasks. When a VP tries to allocate buffer space, it must first execute a bitwise search to locate one or more memory locations large (or small enough) to support the operation's data. Although this action occurs so rapidly that end-user operations aren't affected, in large physical servers with many physical CPUs being used to support IDS instances, the overhead of supporting many concurrent bitwise searches can begin to affect total instance performance as the number of VPs increases.

IDS 11 provides the ability to create "private" memory allocations from within the instance's shared memory structures for each VP. These allocations are managed by the VP and, as a result, greatly reduce the need to execute bitwise searches. Using the new **VP_MEMORY_CACHE_KB $ONCONFIG** parameter, you can set a cache value in KB to be allocated to each VP.

You need to exercise care and caution when using this functionality, however. The total VP cache allocated is bounded by the **SHMTOTAL $ONCONFIG** parameter. Depending on the number of VPs and the cache allocation value, you could easily use all your memory for VP caching, leaving nothing for instance operations. You could also starve other VPs of memory if you have several VPs that are, for the most part, idle. Each VP receives the full cache allocation whether it needs it or not for operations. VPs that are frequently idle consume memory that might be better used in the general instance pool. Generally speaking, IDS development recommends no more than 40 percent of **SHMTOTAL** be used for VP private caches. Their guidance is that much less should be used for this feature.

You're not required to establish a private VP memory cache. Setting the **$ONCONFIG** parameter to **0** (zero) disables the feature. If you choose to use the feature, you can monitor the cache's effectiveness with the **onstat -g vpcache** command.

4

Better Business Continuity Using IDS

In a release packed with a number of fairly new and significant features, none stand taller, in my opinion, than the changes and enhancements to IDS's business continuity features. IDS has led the market in this area for years, and *no other data server* comes close in terms of functionality and performance. The changes made in Version 11 increase this lead and should further solidify IDS's position as the data server on which to bet your business.

IDS Replication Technologies

IDS has had two replication technologies for years. Each is designed to meet a different objective, and, as a result, each has different configuration and management requirements. *High Availability Data Replication (HDR)* should be used for disaster tolerance and high availability requirements. Until this release, this technology was fairly simple to understand: a "primary" instance is mirrored, usually in realtime, to a "secondary" instance so that if the primary fails, the secondary can immediately take over the processing requirements. Figure 4.1 depicts this replication scenario.

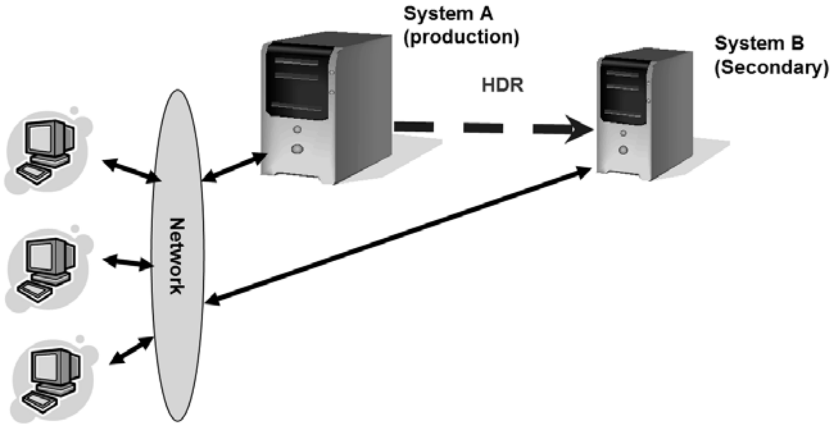


Figure 4.1: High Availability Data Replication (HDR) in IDS

One of the significant differentiators of the IDS HDR secondary instance, when compared with all other technologies, is that it can be used for query operations while still participating in replication activities. Other data servers either can't support this functionality or require breaking replication to the secondary, effectively creating a data silo with out-of-date information.

Setting up HDR is simple and easy to execute. I usually tell customers that if their administrators can't get HDR instantiated in less than a half hour, they ought to consider a different line of work. It's that simple. There are only three decisions to make:

- Which physical server will host the secondary instance?
- Do you want synchronous (strongly recommended) or asynchronous transfers to the secondary?
- Do you want to automate failover to the secondary (my opinion: not recommended)?

Note that the recommendation against automated failover is a personal one on my part and doesn't reflect a weakness in IDS technology. Rather, it is based on real-life experience in which the combination of network conditions and automated failover caused data to be lost. What happened is that the network link between the instances failed. Because the instances couldn't communicate with each other, each (correctly) assumed the other was off-line. The secondary was configured to automatically convert and process transactions, which happened. When the network link was re-established, each instance thought the other should accept its transactions. I had to shut one down and lose its changes.

If you can positively, absolutely, without-a-doubt guarantee that the network will never go down, then you can use automated HDR failover and it will work perfectly.

Enterprise Replication (ER), illustrated in Figure 4.2, is the other IDS replication technology. Its design is focused on data distribution and consolidation throughout the enterprise.

With this technology, using basic SQL statements, you can define rules about which data objects are replicated, when replication occurs, and under what conditions. The figure depicts a topology in which users anywhere on the network can update their local copy of the data, which is, in turn, replicated to all other nodes, including several standby nodes.

Replication topologies can vary depending on your needs, as illustrated in Figure 4.3.

Note: IBM has published a Redbook titled Informix Dynamic Server V10: Superior Data Replication for Availability and Distribution (SG24-7319). This book covers the differences between the two replication technologies and describes how to instantiate and manage their use on an ongoing basis. Therefore, I don't include that information here. To download your copy of this publication, visit http://www.redbooks.ibm.com.

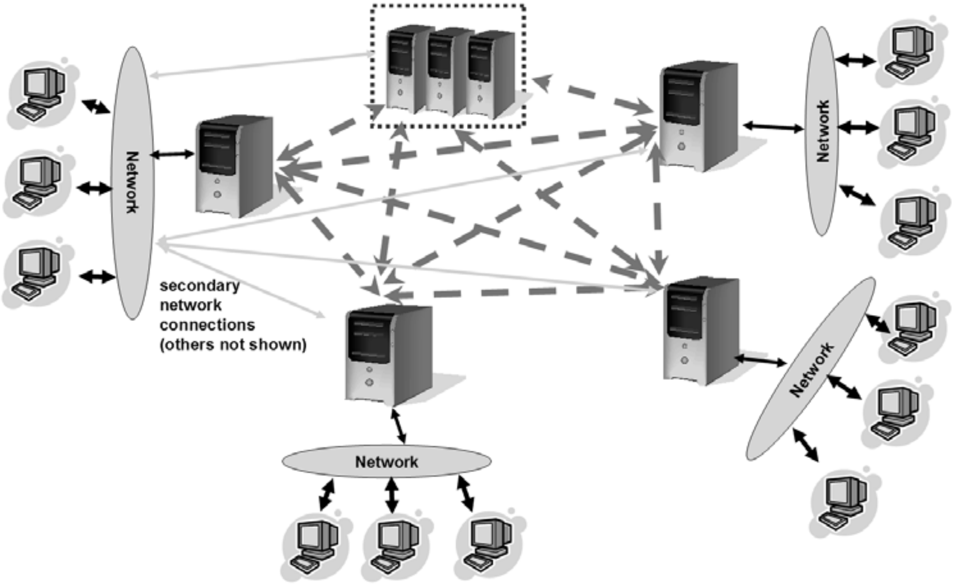
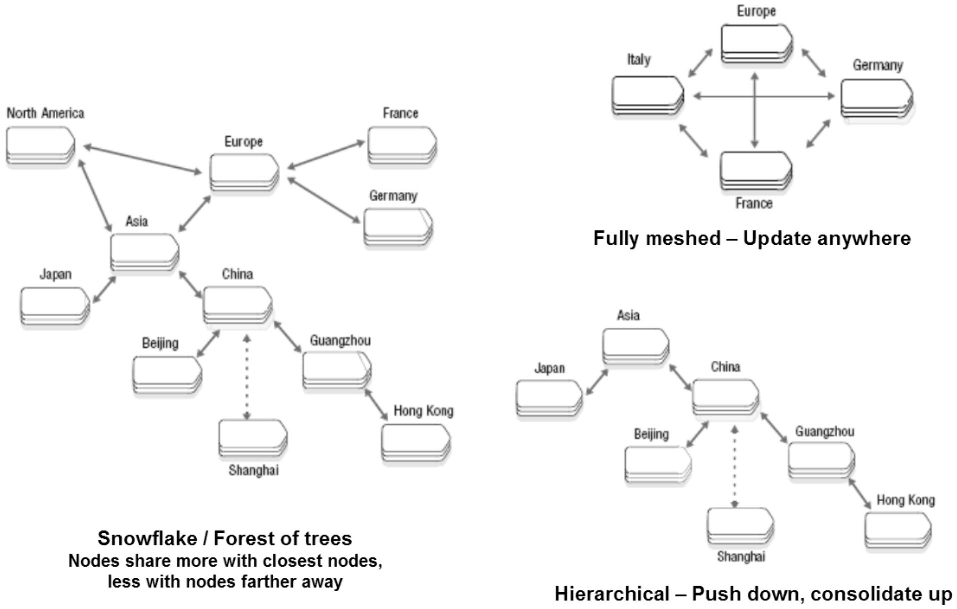


Figure 4.2: Enterprise Replication (ER) in IDS



Snowflake / Forest of trees
Nodes share more with closest nodes, less with nodes farther away

Fully meshed – Update anywhere

Hierarchical – Push down, consolidate up

Figure 4.3: ER topologies

You can define rules so that certain data is always pushed in one direction, such as replicating pricing changes from corporate to retail locations while copying sales aggregations for the day back to the central server. You can define other rules such that some or all data is shared with the closest nodes, either geographically or from a business perspective, while some of the same or perhaps other data is shared with other nodes. You are not locked in to one topology, though, for your environment. You can decide what you need for each data element and build the rules accordingly.

As if having two different replication technologies to solve different needs wasn't enough, you can use both HDR and ER together, as shown in Figure 4.4.

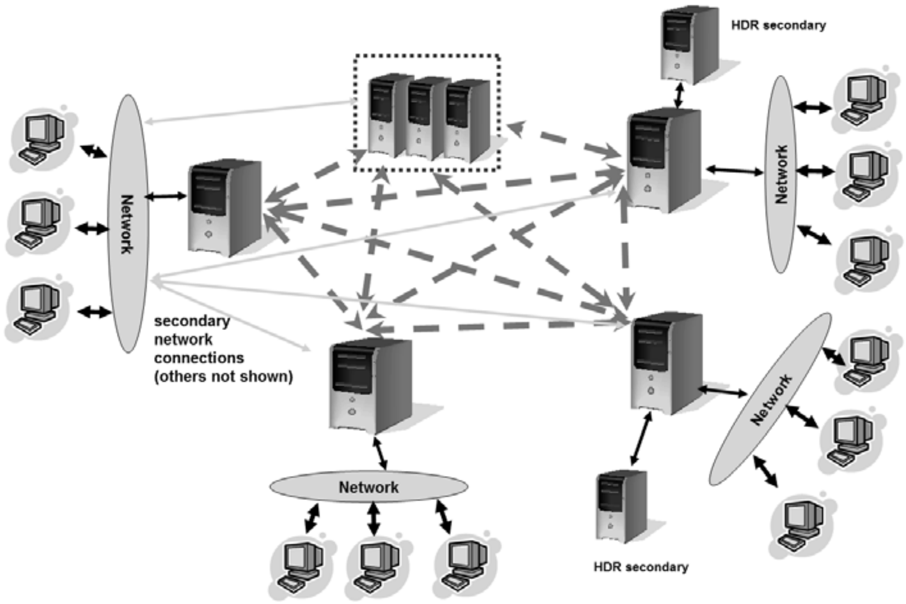


Figure 4.4: Using ER and HDR together

Depending on the business needs, any IDS instance can be protected against failure through HDR while participating in an ER replicate with one or more other instances on the network. No other data server has this depth and breadth of offerings. I feel a little like a TV knife salesman, but in this release, there's more — a lot more!

Enhancements to Enterprise Replication

The past several IDS releases have provided a number of functional and speed enhancements to Enterprise Replication. In this and the next release of IDS, the attention has shifted to HDR. That said, IDS 11 does include a few nice enhancements to ER.

Support for Renaming Objects

One longstanding limitation of ER replicates, whether master replicates or not, was that you couldn't rename a database, table, or column. You can do so now, but you must execute the command(s) on each instance in the replicate set. Although you can make the changes from one instance through a distributed transaction, executing the transaction on all nodes could take some time.

Better yet, you should use master replicates introduced in IDS 10, make the change in the master, and then have the ER mechanism push the change(s) out to all members of the replicate set.

Dynamic ER Configuration

All ER configuration parameters are defined in the **$ONCONFIG** file, and, to date, they've been read only at instance start. If you wanted to make a change, you had to restart the instance.

IDS 11 adds new options to the **cdr** utility that let you change any ER parameter already set as well as add or delete many parameters that either weren't set at instance start or need to be removed. With the functionality introduced in Version 10 to verify and resynchronize schemas and data without interrupting user operations, ER has become completely autonomic.

Truncate Table Support

An extension to the first ER enhancement, the **truncate table** SQL command is now replicated and executed within replicate sets while the set is active.

Optional Trigger Execution During Synchronization Operations

By default, when you define a replicate set, triggers on target tables are not executed when data is received as part of a standard replication operation. The assumption is that data changes that occur as a result of a trigger on the source table(s) will also be replicated to all target servers also receiving the triggering data change. If you need to, you can change this behavior using the **–firetrigger** flag in the replicate definition.

With the resynchronization functionality added in IDS 10, there were some cases where target triggers needed to fire as part of the operation and others where customers wanted to tightly control the synchronized data and avoid all triggers. IDS 11 adds the **–firetrigger** option to the synchronization commands as well, letting you set all triggers to always fire, to fire only if the **–firetrigger** option is part of the replicate definition, or to never fire.

Enhancements to HDR

Okay, this is where it gets *really* exciting. In IDS 11, IBM has extended HDR functionality to include the ability to have more than one secondary server, to have multiple backup instances share the same physical disk, and to create and maintain near-line copies of the data.

All at the same time.

With seamless failover.

And this is just the beginning of what will be happening with HDR!

New Components to Support HDR Enhancements

Radical and market-changing enhancements such as those available with this IDS release often require additional changes and enhancements to the data server. Let's look at two new parts of IDS's plumbing that make HDR's expanded functionality possible.

The first part is a new communications protocol called the Server Multiplexer (SMX). As the name implies, the SMX acts as a communications bridge between two or more data servers. It uses a "multiple-in, single-out" protocol to intelligently manage and distribute multiple simultaneous communications requests in a very efficient manner.

The SMX uses a full-duplex communications protocol, which means that messages are sent without waiting for an acknowledgment of receipt from the target. Lest you think this is a severe flaw or weakness, the replication systems that sit on top of this protocol do monitor for returned acknowledgments to make sure the data communicated to the target(s) is being received. If a target falls too far behind in its acknowledgments, that server is removed from the replication set.

The SMX also supports data encryption, so if your environment requires strong encryption of all data traffic, even between the failover instances, you can enable it as discussed in Chapter 2.

As befits IDS and its legendary reputation for ease of use, no administrative input is required to instantiate the SMX protocol unless you want to use encryption between the nodes. It just runs on its own.

You can view and analyze threads associated with the SMX protocol using the **onstat** utility. Flag options to this utility for SMX activity include **-g smx** and **-g smx ses**. When the SMX protocol is being used to send or receive data, a "send" and "receive" thread is created for each defined instance as a part of the new HDR functionality.

The second change in IDS's plumbing relates to the way index creation is logged and transmitted to the secondary instance(s). Currently, when an index is created on an HDR primary instance, the secondary instance must be online and available to receive the updates created as part of the index creation. With Continuous Log Restore (CLR) and other new features, the target instance(s) might not be active, a circumstance that would prevent you from making necessary changes to the production instance.

For this reason, the logging of index creation in an HDR environment has changed to break the creation into multiple "transactions" as far as the logical logs are concerned. Each "index transaction" is logged independently of user transactions. The full content of the created index pages is included in the logical log(s), which is also different. As a result, when the logical log(s) are rolled forward on the CLR and other secondary instances, the index is completely re-created.

A new **$ONCONFIG** parameter, **LOG_INDEX_BUILDS**, controls whether index logging occurs. You can either set this value in the file before starting the instance or dynamically activate it and update the **$ONCONFIG** file using the **onmode** utility.

You should be aware that when index logging is turned on, it is used for all instances in an HDR cluster, including overloading the index transfer used for the HDR secondary. Depending on the size of the index created, there is likely to be a significant increase in logical log usage and volume. You'd be wise to ensure that your log backup mechanism is working properly and that you have a sufficient number, and size, of logical logs created so that some logs can be archived off and don't all fill while you're trying to back up the first log. If that happens, the instance will either suspend operations or begin to dynamically create numerous new logical logs, which you'll need to drop after the build is completed and the log records have been transferred to necessary target instances.

Continuous Log Restore Servers

The easiest way to describe Continuous Log Restore servers is to say that they use log shipping from the primary to the CLR instance, establishing a near-line replication environment as shown in Figure 4.5.

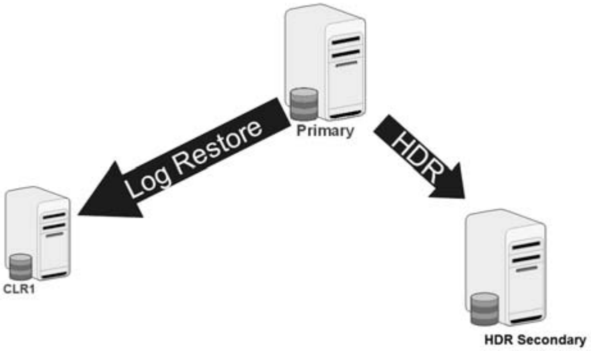


Figure 4.5: Conceptual drawing of CLR functionality

This functionality is actually an enhancement to the **ontape** and **ON-Bar** utilities that permits an instance to remain in a perpetual log roll-forward state and to apply new logical logs on demand.

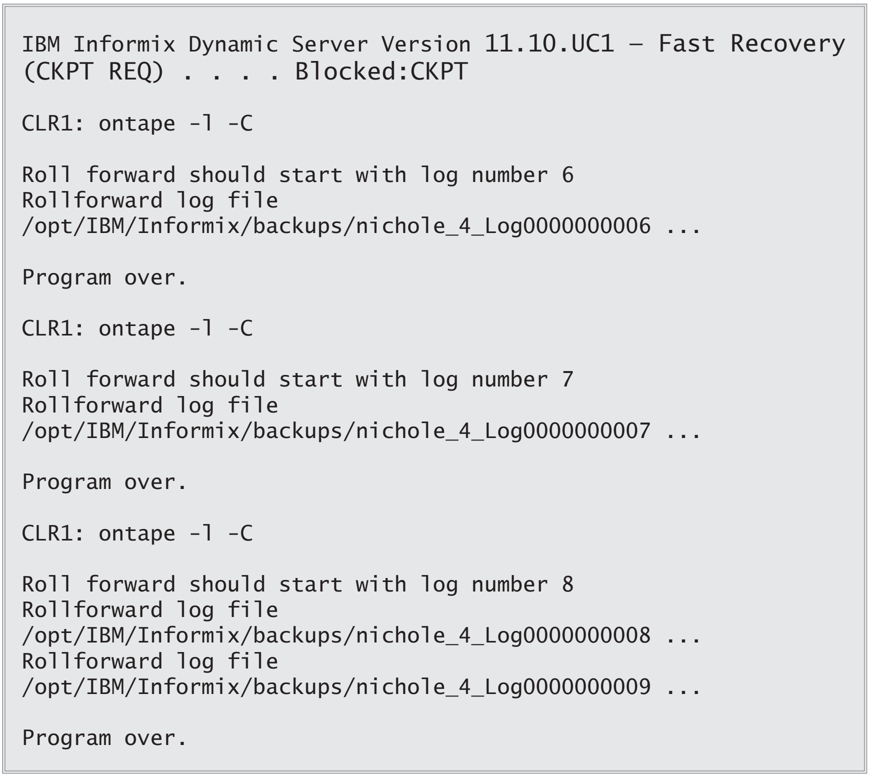
The major application of this feature is rather straightforward. We have a number of customers with IDS instances deployed in areas of the world where the telecommunications infrastructure is not completely reliable. They cannot rely on constant network availability to build an offsite failover location as required by a "regular"

HDR secondary. These customers still want to provide a measure of failover and disaster recovery protection. CLR instances enable them to do so.

Other customers with high network availability can use CLR to create a fourth layer of failover redundancy through the creation of a “bunker” backup. You’re already familiar with the first layer: the HDR secondary. The second and third layers are new, and I’ll explain them in the sections that follow.

From an implementation perspective, as the primary backs up full logical logs, these logs are copied to the CLR instance(s). The assumption is that log backups will occur to disk because such backups are easier to access and transfer to the remote physical servers, but the logs can certainly be extracted from tape; it’s just harder.

The instance(s) on the remote physical server(s) don’t have to be active to participate as CLR instances. Once the logs are on the remote physical server(s), they can either be applied to active CLR instances in recovery mode or just left in place on disk to be applied in the event that a failure occurs and the CLR instance(s) need to be activated. If the instance(s) are active, though, you use the new **-C** (capital “c”) flag to apply the logs and leave the instance in roll-forward mode, as illustrated in Figure 4.6.



```
IBM Informix Dynamic Server Version 11.10.UC1 — Fast Recovery
(CKPT REQ) . . . . Blocked:CKPT

CLR1: ontape -l -C

Roll forward should start with log number 6
Rollforward log file
/opt/IBM/Informix/backups/nichole_4_Log0000000006 ...

Program over.

CLR1: ontape -l -C

Roll forward should start with log number 7
Rollforward log file
/opt/IBM/Informix/backups/nichole_4_Log0000000007 ...

Program over.

CLR1: ontape -l -C

Roll forward should start with log number 8
Rollforward log file
/opt/IBM/Informix/backups/nichole_4_Log0000000008 ...
Rollforward log file
/opt/IBM/Informix/backups/nichole_4_Log0000000009 ...

Program over.
```

Figure 4.6: Logical logs being rolled forward on a CLR server

In the figure, the logs are being applied manually, but this process could be easily automated by the database scheduler discussed in Chapter 3.

At any time, you can transition a CLR server from roll-forward mode to a standard operating mode by using the new **-X** (capital "x") flag:

```
Clr1: ontape -l -X
Program over.

Clr1: onstat -
IBM Informix Dynamic Server Version 11.10.UC1 – Quiescent . . . .
```

Remote Standby Secondary Servers

Remote Standby Secondary (RSS) servers are like having additional HDR secondary instances, as Figure 4.7 illustrates.

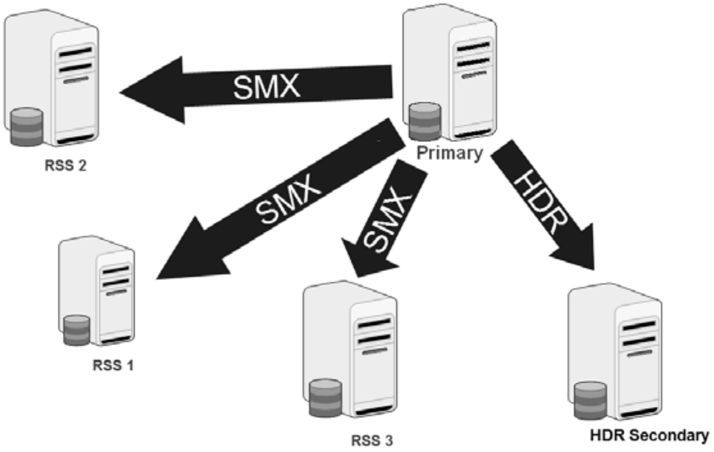


Figure 4.7: Conceptual drawing of RSS functionality

This feature is not just 1 to N HDR secondary, though. An RSS instance behaves slightly differently from the "regular" HDR secondary instance. First, you can have as many RSS instances as you like. Because RSS works on top of the SMX communications protocol, it uses full-duplex asynchronous protocols as opposed to synchronous protocols for an HDR secondary. A side effect of this communications protocol is that an RSS instance cannot be promoted to become an HDR primary. It can, however, become an HDR secondary. Finally, RSS instances don't recognize the value of the **DRAUTO** parameter. As a result, RSS servers should be considered disaster recovery instances, not high availability (HA) instances.

RSS instances are similar to an HDR secondary in that they maintain a complete and full copy of the source instance. You can use them for read-only operations provided there is some flexibility in time. Because of the asynchronous nature of the SMX protocol, there might be a slight lag between data being committed on the primary and it appearing on the RSS instance(s). Because an RSS instance can be promoted to an HDR secondary, it makes sense that an HDR secondary can be converted to an RSS instance.

There are several business cases for using RSS instance(s). The first is to expand the realtime failover capability of an HDR environment. You can create an HDR secondary and host its physical server in close proximity to the primary to cover hardware and other transient failures while maintaining one or more realtime RSS copies well off-site for disaster tolerance and eventual failover.

You can also use an RSS instance as opposed to an HDR secondary in environments where network connectivity is stable but throughput is poor. For example, a distant regional instance might be serviced only by a low-bandwidth connection — a dial-up line, perhaps. In this case, the network latency would be too great to support the synchronous requirements of an HDR secondary, but an RSS instance would be unaffected.

A third application would be to provide complete redundancy in the event that the primary instance fails, as illustrated in Figure 4.8.

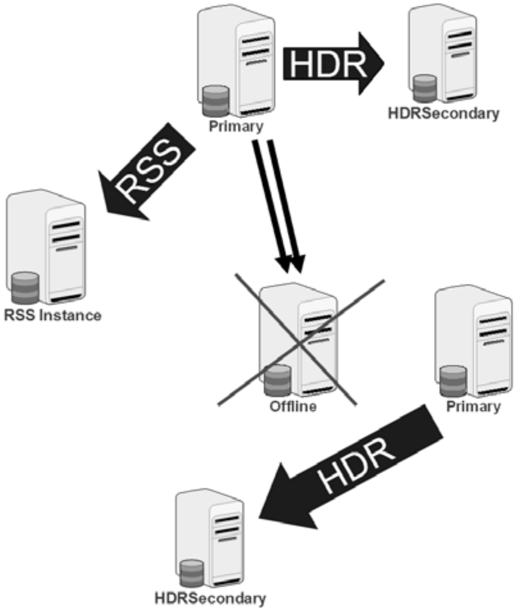
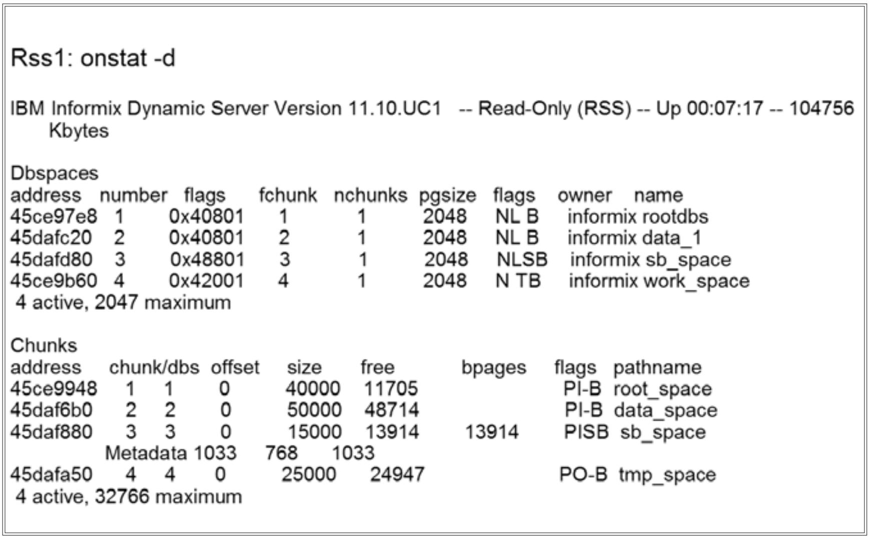


Figure 4.8: Using RSS functionality to provide redundancy when the primary fails

In this example, a primary is communicating to a secondary and an RSS instance. When the primary fails, the secondary becomes the HDR primary, and the RSS instance becomes the HDR secondary. In this way, even the loss of an instance doesn't compromise the failover and redundancy capability of the data server environment.

Instantiating an RSS instance differs little from instantiating an HDR secondary. You need to turn on the index logging facility, as I mentioned in the beginning of this section. A backup from the primary is restored to the RSS instance(s), and an **onmode** command is issued on both the primary and the RSS instance(s) to link them together. In an interesting twist, you can use an optional password when establishing the RSS connection so you don't have a random server trying to connect to your primary instance and copy your data.

Once an RSS instance is operational, it looks and behaves much like an HDR secondary. Figure 4.9 shows a screen shot of an **onstat -d** command executed on an RSS instance.



```
Rss1: onstat -d

IBM Informix Dynamic Server Version 11.10.UC1   -- Read-Only (RSS) -- Up 00:07:17 -- 104756
     Kbytes

Dbspaces
address   number  flags     fchunk  nchunks  pgsize  flags   owner   name
45ce97e8  1       0x40801   1       1        2048    NL B    informix rootdbs
45dafc20  2       0x40801   2       1        2048    NL B    informix data_1
45dafd80  3       0x48801   3       1        2048    NLSB    informix sb_space
45ce9b60  4       0x42001   4       1        2048    N TB    informix work_space
 4 active, 2047 maximum

Chunks
address    chunk/dbs  offset   size    free       bpages    flags  pathname
45ce9948    1   1     0        40000   11705                PI-B  root_space
45daf6b0    2   2     0        50000   48714                PI-B  data_space
45daf880    3   3     0        15000   13914      13914     PISB  sb_space
           Metadata 1033       768     1033
45dafa50    4   4     0        25000   24947                PO-B  tmp_space
 4 active, 32766 maximum
```

Figure 4.9: Output from an RSS server onstat command

There are some new flags marking each space, and the space's status is "inconsistent"; otherwise, the instance looks more or less the same as a "regular" instance. An **onstat -u** shows a number of additional active sessions; they are used to manage replication. Other **onstat** options display the number of log pages sent and received, where the RSS instance(s) are as far as responding back to the primary, SMX information, and more.

Shared Disk Secondary Servers

Shared Disk Secondary (SDS) servers are mirror instances that share the same disk devices as the primary instance. Figure 4.10 depicts the SDS server environment.

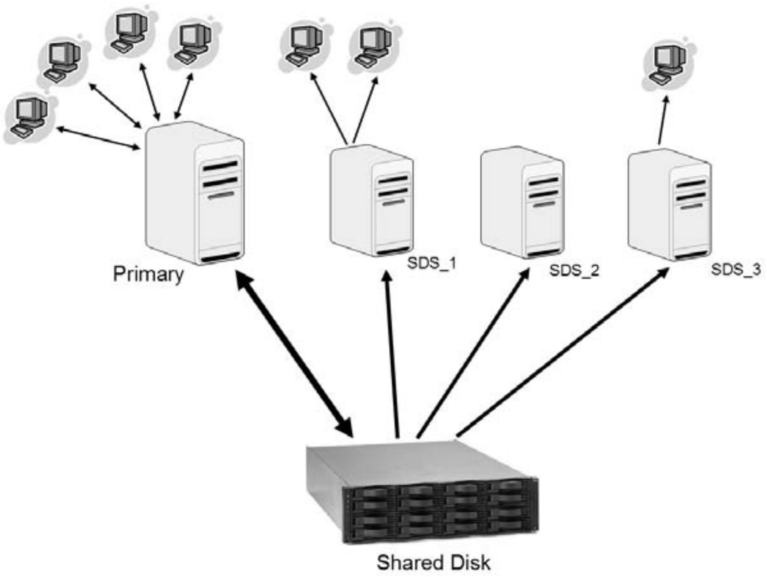


Figure 4.10: A conceptual view of an SDS server environment

You can use this functionality to provide availability and failover options where the storage devices are network-mounted — the physical server(s) hosting the SDS instance(s) can be anywhere on the network and provide failover and redundancy capabilities. The weakness to an SDS instance is that it doesn't protect against disk failure if the mounted devices fail. You'd be wise to use a combination of SDS and RSS technology to build a highly fault-tolerant environment.

Because the SDS instance(s) share the same devices as the primary instance, almost no setup is required to instantiate the SDS instance(s):

> Note: Because the SDS instances share the same disks as the primary, it obviously wouldn't be wise to use the "initialize" flag when starting the instance!

1. Tell the primary there will be SDS servers.

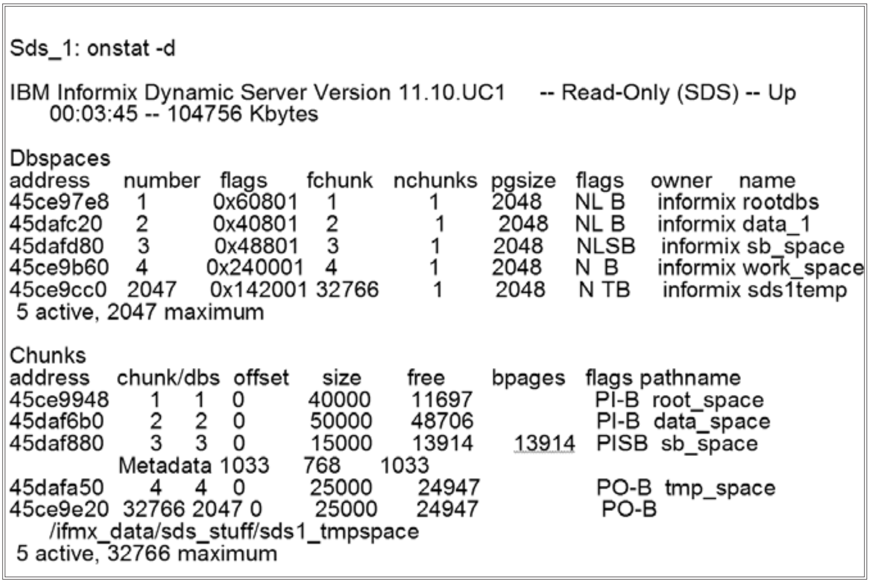
2. Copy the **$ONCONFIG** file from the primary to the physical server(s) hosting the SDS instances.

3. Change the name(s) of the instance and the instance number in the SDS **$ONCONFIG**.

4. Set the three SDS-specific **$ONCONFIG** parameters.

5. Turn the instance on.

Because SDS instances are read-only, they cannot use the same temporary dbspace(s) for implicit or explicit temporary tables to support SQL operations. One of the SDS-specific parameters you must set is a "local" temporary dbspace. This local temporary dbspace is visible only to the SDS node, as the output of an **onstat -d** command illustrates in Figure 4.11. Compare this output with that in Figure 4.9, which accurately reflects what the primary instance sees as configured dbspaces and devices.



```
Sds_1: onstat -d

IBM Informix Dynamic Server Version 11.10.UC1     -- Read-Only (SDS) -- Up
     00:03:45 -- 104756 Kbytes

Dbspaces
address    number  flags      fchunk  nchunks  pgsize  flags    owner    name
45ce97e8   1       0x60801    1       1        2048    NL B     informix rootdbs
45dafc20   2       0x40801    2       1        2048    NL B     informix data_1
45dafd80   3       0x48801    3       1        2048    NLSB     informix sb_space
45ce9b60   4       0x240001   4       1        2048    N  B     informix work_space
45ce9cc0   2047    0x142001 32766     1        2048    N TB     informix sds1temp
 5 active, 2047 maximum

Chunks
address    chunk/dbs  offset   size     free     bpages   flags pathname
45ce9948      1    1    0      40000    11697             PI-B  root_space
45daf6b0      2    2    0      50000    48706             PI-B  data_space
45daf880      3    3    0      15000    13914    13914    PISB  sb_space
              Metadata 1033    768      1033
45dafa50      4    4    0      25000    24947             PO-B  tmp_space
45ce9e20   32766 2047 0        25000    24947             PO-B
     /ifmx_data/sds_stuff/sds1_tmpspace
 5 active, 32766 maximum
```

Figure 4.11: dbspace output from an SDS instance

With the SDS instance(s) instantiated, the primary uses the SMX protocol to send log information to the SDS instance(s). Included with the log information is the Log Sequence Number (LSN), which the primary and SDS instance(s) use to verify shared memory consistency.

When the SDS instance(s) receive log records, they use the records to update the buffer cache so it reflects the cache as it exists on the primary. When a checkpoint occurs on the primary, its buffers are flushed to disk, but the instance will retain any buffers that SDS instance(s) have not acknowledged as having applied to its cache. SDS instance caches are cleared when the instance receives the checkpoint record with its accompanying LSN and determines that all previous LSN records have been applied. The SDS instance communicates this fact to the primary, which releases any remaining buffers flushed as part of the checkpoint.

From a recovery and failover perspective, an SDS instance should be the "lead-with" instance to promote when a failure occurs. You could then follow with the HDR secondary and then any RSS instances after the RSS instance has been promoted to HDR secondary. You can, in fact, promote any SDS instance to be the primary while the cluster is online and functioning. All the other instances will automatically adjust and reconnect to the new primary. It's pretty cool to watch happen! If you don't want to fail over to an SDS instance, you should know that using the **DRAUTO** parameter to automatically switch to the HDR secondary in the event of a failure will automatically shut down the SDS instances.

You can monitor SDS instances just like RSS instance(s), by using **onstat** flags that show SMX activity and statistics as well as SDS-specific information. The primary instance will show a heavier session load, with six sessions specifically dedicated to SDS administration in addition to SMX and other new overhead.

Putting It All Together

So how does all this work together, and what can you do with it? As Fred Barnard is alleged to have said, "A picture is worth a thousand words."

Figure 4.12 shows users connected to the primary instance executing transactions. Other users could be connected to the other instances, but to keep the diagram somewhat easy to read, I haven't shown them.

Location A has sets of rack servers along with the main network-attached storage array. The storage array is mirrored using array-based functionality. The primary is also publishing updates to an HDR secondary and an RSS instance at remote locations.

Imagine that a failure occurs, shutting down half of the equipment at Location A. The cluster automatically reconfigures as shown in Figure 4.13. Applications reconnect automatically and continue processing.

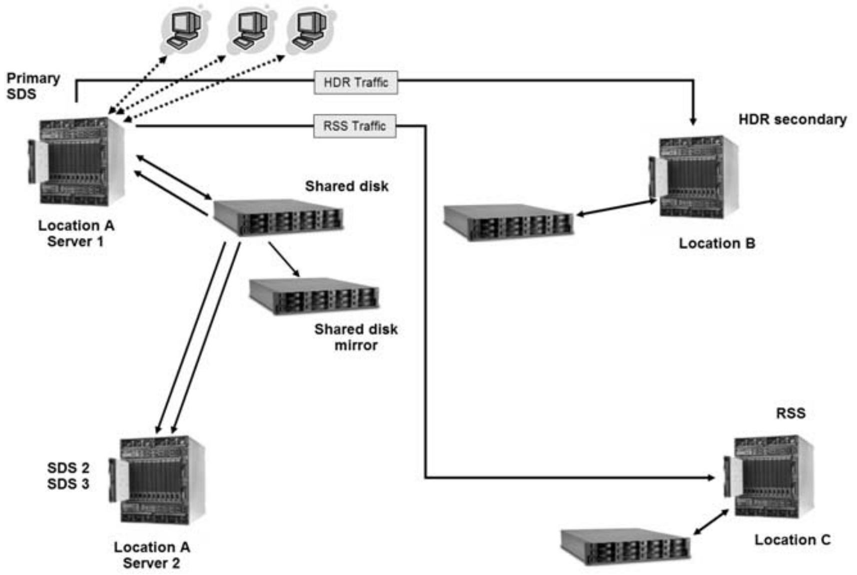


Figure 4.12: Original data server processing environment

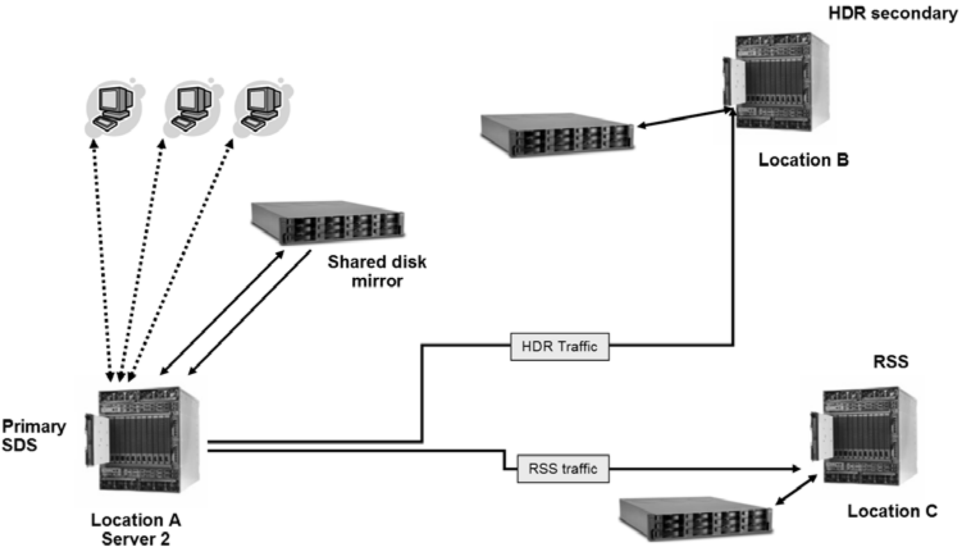


Figure 4.13: The first failure occurs, and the cluster reconfigures.

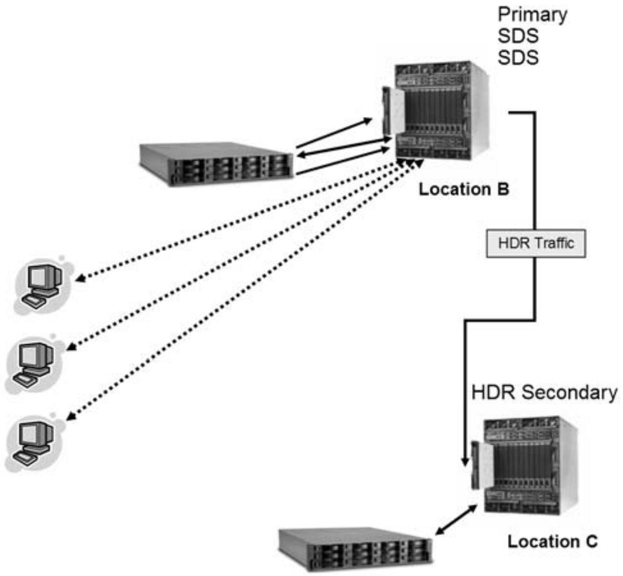


Figure 4.14: The second failure occurs, and the cluster reconfigures again.

Unfortunately, it's not your day, and Location A next goes completely offline. Not to worry! In this adjustment, notice that additional SDS instances have been manually added at Location B, and the instance at Location C has been promoted to HDR secondary.

While you're working to get Location A back up and running, an event occurs at Location B that takes it offline. Once again, the cluster reconfigures and work continues as shown in Figure 4.15.

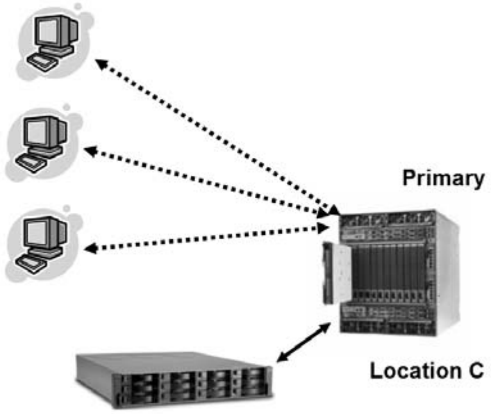


Figure 4.15: Another failure occurs, and the cluster reconfigures.

But that’s not all. What I haven’t shown is the possibility that you also could have set up one or more CLR instances and could have been copying the logical logs to them. In that case, you could bring at least one of these online and convert it into an HDR secondary instance, thus continuing to provide robust business continuity in the face of almost any kind of failure!

This page intentionally left blank

5

Easing Application Development

Well, your tour of new IDS functionality is almost over! I hope by now you've realized that this is no small release from a performance, administrative, security, and availability perspective. Like the animal this release is code-named after, IDS 11 is blazingly fast and out to hunt all competitors!

In this last chapter, we look at the enhancements aimed at application developers. I'll be honest; this is perhaps my weakest area of data server knowledge. It's not that I haven't written end-user applications; I have. But I used the Informix-4GL programming language, a remarkably easy and powerful fourth-generation language. Although procedural in nature, it let you do almost anything you wanted to do — except deploy in a point-and-click, graphical environment! Compilers were (and are) available for 4GL source code to create a Windows or more O/S-portable graphical implementation of the application, but I didn't use them. My applications were text-oriented, and having to use a mouse to change fields or move to the next screen would have slowed users down too much.

Regardless of which programming language you use, IDS has significant, broad, and deep application development support. Let's look at some of the new features included in Version 11 that continue IDS's tradition of easing application development.

Pulling the Trigger, Multiple Times

In building applications, there are two schools of thought. The first is to put all business logic in the application; the data server is considered just a storage repository. I don't think I'd be too far wrong to say that most application developers think this way. The second perspective to application development puts as much business logic into the data server as possible. With this approach, all operations of the same kind execute identically regardless of the application executing them, and a single set of business rules verifies all data. Most data manipulation occurs in the data server rather than large quantities of intermittent result-set data being pushed across the network to the lower-powered client.

If you couldn't tell, I support the second school of thought more than the first. I don't think the matter is exactly black and white, though; there are times when some work should be executed on the client. But if most business logic and rules will be executed in the data server, you'll most likely need to have things happen automatically when data is entered, deleted, changed, or even queried. This behavior is typically accomplished using triggers.

In IDS, you can create triggers on the usual SQL operations (**insert**, **update**, **delete**, and **select**) and can call user-defined routines (UDRs) written in C, Java, or Stored Procedure Language (SPL). IDS 10 introduced a feature called *trigger introspection*, which permits interrogation of data within the triggered UDRs. With this feature, you can trap error codes and conditions as well as return interim variable values, such as the serial or sequence number for a newly inserted row so it can be used in another part of the UDR.

Until now, the number and types of triggers you could create on a table have had some limitations. Although you could create multiple **select** and **update** triggers (provided the **update** triggers referenced different columns), you could have only one **insert** and **delete** trigger. In IDS 11, you now have the ability to define multiple triggers for all triggering actions, including **instead of** triggers on views as shown in Figure 5.1. (This figure was extracted from the **cr_trig.sql** example file in the IDS distribution.)

```
– Multiple triggers on the same Insert event on manufact table

create trigger manu_ins_trig1 insert on manufact
   before
      (insert into manu_operations_summary values (“New Manufacturer”))
   for each row
      (execute procedure manu_proc() with trigger references);

create trigger manu_ins_trig2 insert on manufact referencing NEW as new
   for each row
     (insert into manu_log values (new.manu_code,new.manu_account,"INSERT"));

-- Multiple triggers on the same Update event on manufact table

create trigger manu_upd_trig1 update on manufact
   before
     (insert into manu_operations_summary values (“Manufacturer info
changed”))
   for each row
     (execute procedure manu_proc() with trigger references);

create trigger manu_upd_trig2 update on manufact
   referencing OLD as old New as new
   for each row
     (insert into manu_log values (new.manu_code,new.manu_account,"UPDATE"));
```

Figure 5.1: Examples of creating multiple triggers on the same triggering event

Inside the trigger body, you can have **before**, **for each row**, and **after** actions as well as **old** and **new** variables. When multiple triggers are defined for the same triggering event, all **before** actions of the appropriate triggers are executed first, followed by **for each row** actions. Last, the **after** actions are executed. This process ensures there are no out-of-sequence execution issues if you need to drop, modify, and re-create one or more triggers.

You can use new Boolean operators with **case** or **if** constructs in UDRs called by triggers to manipulate values, particularly variables defined as **new**. Figure 5.2 shows examples (taken from IDS development documentation) of the new operators in use.

```
create procedure proc1()
   referencing OLD as o NEW as n for tab1; – new syntax.

   if (INSERTING) then  – INSERTING new boolean function
        n.col1 = ncol1 + 1;   – You can modify the new values.
        insert into temptab1 values(0,n.col1,0,n.col2);
   end if

   if (UPDATING) then  – UPDATING new boolean function
        insert into temptab1 values(o.col1,n.col1,o.col2,n.col2);
   end if

   if (SELECTING) then  – SELECTING new boolean function
             – you can access relevant old and new values.
        insert into temptab1 values (o.col1,0,o.col2,0);
   end if

   if (DELETING) then   – DELETING new boolean function
        delete from temptab1 where temptab1.col1 = o.col1;
   end if

end procedure;

create procedure proc2()
   referencing OLD as o NEW as n for tab2
   returning int;

   LET n.col1 = n.col1 * 1.1 ; – increment the inserted value by 1

end procedure;

create trigger trig_tab1 INSERT on tab1 referencing new as post
   for each row(execute procedure proc1() with trigger references);

create trigger trig_tab1 INSERT on tab2 referencing new as n
   for each row (execute procedure proc2() with trigger references);
```

Figure 5.2: New Boolean operators in UDRs called by triggers

There are several new pieces of syntax in this figure. The **inserting**, **deleting**, **selecting**, and **updating** keywords are the new Boolean functions. You can access and change the value of the **new** variables in the triggered UDRs. And, finally, the **with trigger references** phrase enables you to work through the trigger to get to the values.

XML in IDS

You'd have to have been living under a rock in the middle of the deepest ocean trench to have missed the fact that XML has emerged as the *lingua franca* of the Service Oriented Architecture (SOA). IBM released a significant enhancement to DB2 9 called "pureXML" that enables users to manipulate XML documents inside the data server in their native format. The documents are inserted and stored as an XML data type, operated on using XPath parameters and operations, and more. The technology enables businesses to satisfy regulatory compliance rules for saving all documents in their native and unaltered format.

There are, however, two approaches to XML and its use. The "document-centric" approach (implemented in DB2 9), treats XML data as a single object, while the "data-centric" approach breaks (or shreds) the XML document into its component parts and stores each part separately. IDS is taking a data-centric approach and leveraging several extensible data types to store XML in its native hierarchical format. For some customers, this path may be acceptable from a compliance perspective; others will also need to store the XML document in a BLOB column, effectively storing the data twice.

With IDS 11, you can publish the results of SQL operations in properly formatted XML through functions provided as part of the GenXML library. You can parse the data, serialize it, or even operate on it to determine whether specific components exist. Using the included Extensible Stylesheet Transformation (XSLT) functions, you'll be able to transform the XML document into other formats if necessary. Figure 5.3 illustrates some of the XML functionality available with the GenXML libraries.

```
SELECT First_name, extract(fname, ‘/amateur/call_sign‘,
‘/amateur/license_class’) “Call_Sign”, “License Class”
FROM ares_pool
WHERE is_active IS true;

First_name   Call_Sign                    License Class
__________   _________                    _____________
Craig        <call_sign>N5XKB</call_sign>   <license_class>Extra</license_class>
Ron          <call_sign>W5BRK</call_sign>   <license_class>Extra</license_class>
Carlton      <call_sign>W3DOE</call_sign>   <license_class>Extra</license_class>
Glen         <call_sign>KE5HEU</call_sign>  <license_class>Technician</license_class>
```

Figure 5.3: Examples of some supported XML syntax and results (part 1 of 2)

```
SELECT genxmlelemclob(ares_pool, “ares_pool”) FROM ares_pool;

<ares_pool>
<givenname>John</givenname>
<familyname>Doe</familyname>
<address>
<address1>1234 Main Street</address1>
<city>Anytown</city>
<state/>TX</state>
<zipcode>75234</zipcode>
</address>
<phone>972-555-1212</phone>
<call_sign>KE5JR</call_sign>
</ares_pool>
.
.

SELECT idsxmlparse( ‘<purchaseOrder poNo="124356">
<customerName>ABC Enterprises</customerName>
<itemNo>F123456</itemNo> </purchaseOrder>’)
AS PO FROM systables where tabid = 1;

<purchaseOrder poNo="124356">
<customerName>ABC Enterprises</customerName>
<itemNo>F123456</itemNo>
</purchaseOrder>
```

Figure 5.3: Examples of some supported XML syntax and results (part 2 of 2)

Use of this XML functionality requires the creation of a specific user-defined virtual processor (UDVP) named **idsxmlvp**. If you forget to define this virtual processor in the **$ONCONFIG** file or you don’t add it dynamically after the instance starts, an error message will be recorded in the **$MSGPATH** file when you try to use GenXML library functionality.

Although it is difficult to predict what will be in a future release of a product, it would be reasonable to assume that you’ll see more IDS XML functionality in the near future.

DataBlade Integration

As I mentioned in Chapter 1, one of the greatest things about the IDS data server is its ability to adapt and manage work the way you need, rather than according to static design theory created to help manage-lower functionality data servers. You can add data and processing functionality to IDS through the use of new data types and functions created to execute specific tasks. You can either build your own or purchase the functionality you need from any IBM IDS authorized reseller.

In earlier releases of the server, all DataBlades were for-charge products requiring an additional license fee and installation for their use. This policy had one exception: The Spatial DataBlade was included with Enterprise Edition licenses. With this release, though, several other DataBlades are included with the server. This section briefly covers the bundled Blades and their functionality. Contact your authorized IDS reseller to find out about any additional license fees that may be due for using this functionality.

Binary

When I first saw this item on the feature sheet I was excited — bitmapped indexes in IDS? Sweet! Alas, I was mistaken. This DataBlade provides the ability to create indexable data types using small opaque binary–encoded strings.

As I explained in Chapter 1, IDS has long supported the ability to store “nonstandard” data either as black-box data streams (in the form of a simple BLOB) or as intelligently parsable data types stored in smart BLOBspaces. Unfortunately, most of these types can’t support indexes, and they require the use of metadata for direct access. With the Binary DataBlade, you can create columns using two new data types, store them in-table, and create indexes on their contents for fast access.

The **binary18** data type is a fixed-length data type holding 18 bytes. Because this data type is fixed in length, unused space will be right-padded with zeros until the column length reaches 18. The **binaryvar** data type is a variable-length type that can hold up to 255 bytes of information.

Because they are binary types, there are some rules about the data these new types can contain. The data

- must be represented in an even-numbered byte length
- can use only standard ASCII character codes
- must conform to the hexadecimal range of values (0 to F)

```
create table my_binary_tab
  (col1 smallint,
   col2 binaryvar);

insert into my_binary_tab values (1, ‘0X3031dd33e4353a3738e9’);
insert into my_binary_tab values (2, ‘90312a34f4353b4938c0’);
```

Once you’ve created and populated columns with these types, you can use functions provided by the Blade to execute bitwise **and**, **or**, **xor**, and **complement** (a.k.a. **not**) operations as well as standard **count**, **distinct**, **max**, and **min** operations. Character-oriented operations such as **like** or **matches** are not supported, though.

Basic Text Search

Text operations in SQL are expensive and hard to construct properly to catch all the possible string variations. Comparison operators are also very limited, using only **like** and **matches**. Building a search string that will find the desired result set becomes less and less likely as the length of the text to search increases.

With this release, IDS is including the Basic Text Search DataBlade (BTS) as an interim step between the limited SQL functionality and the full-featured, for-charge Excalibur Text Search DataBlade. You invoke BTS functionality through a function call to the text search "engine," which is running on its own VP. For example, the query

```
select employee_num
   from employee
   where bts_contains(first_name, 'C*n')
```

returns the employee numbers of all those whose first name begins with "C" and ends with "n." The query

```
select address_info
   from property
   where bts_contains(property_condition, 'delap~')
```

performs a fuzzy search using "delap" as the search root. You can execute proximity searches to determine whether key words are within a specified distance from each other by structuring the predicate to look like this:

```
select address_info
   from property
   where bts_contains(description, ' "mountain cabin"~10 ')
```

In this example, the result set will include property information where "mountain" and "cabin" are within 10 words of each other.

There's more to the BTS than the three examples shown here. It contains other functions you can use to conduct other kinds of searches.

At present, the Basic Text Search DataBlade supports only ASCII characters stored in the **char**, **(l)varchar**, **clob**, and **blob** data types. Unicode characters are not supported in this release. The DataBlade uses an overloaded index that must be created on each column on which you want to execute Blade functionality. Because of its structure, this index must be stored in an external dbspace. The IDS documentation explains in greater detail the functionality, requirements, and limitations of this Blade. I encourage you to read through this information to fully appreciate what the BTS Blade can do.

Geospatial

The Spatial DataBlade has been available for several years, and, as I mentioned at the beginning of this section, customers with IDS Enterprise Edition licenses have been able to use it without an additional license fee since IDS 9.4. In Version 11, all IDS customers, regardless of licensed edition, can use this Blade without an additional license fee. This DataBlade enables analysis based on location and proximity using the industry’s fastest access method: the R-tree (Region Tree) index.

As businesses like yours look into expanding markets or solidify those they have, every relationship becomes important. Geography is an important part of any relationship. Without the Spatial Blade, it’s impossible to ask a standard data server questions such as, “Show me all rooms available for the next five days within 45 miles of the patient’s location equipped with a defibrillator.” Standard SQL can’t describe a circle or a radius, its ability to evaluate time is limited, and, as I’ve just explained, searching text is extremely hard. With IDS’s object-relational capabilities, you can extend the data server’s functionality with this Blade to handle all these types and relationships easily. In fact, the SQL statement to ask the earlier question is trivial, as Figure 5.4 shows.

```
create table HospitalRooms  (
  name varchar(128) not null,
  equipment document not null
  where st_point not null,
  occupied set( period not null ));


select h.name
   from HospitalRooms h, patients p
   where st_within (h.where, st_buffer(p.location,’45 Miles’ ))
      and p.Name = ‘Jane Doe’
      and DocContains(H.Equipment, “defibrillator” )
      and not booked (h.bookings, period(today,today+5));
```

Figure 5.4: Using the IDS Spatial DataBlade to easily solve a complex question

Node

The Node DataBlade is a fully supported version of technology that’s been available on the International Informix Users Group (IIUG; http://www.iiug.org) and IBM DeveloperWorks (http://www-128.ibm.com/developerworks) Web sites for quite a while. This Blade provides the ability to accurately model hierarchical data in its native format. XML is one example of a hierarchical data stream, but so are objects such as your company’s organizational chart, network designs, biological kingdoms, fabric classification systems, and more.

A design such as the one shown in Figure 5.5 is impossible to model in a standard relational data server. The closest such a server can come requires self-referencing tables and either set processing or highly recursive SQL operations to find and return data. Both options become exponentially more difficult to implement and expensive to execute as the hierarchy expands in depth or width. The **node** data type that this DataBlade contains “understands” data point interrelationships from a lateral and horizontal perspective.

The Node Blade provides functions such as **depth**, **compare**, **ancestors**, **getmember**, and others, giving you direct access to the data you need instead of convoluted recursive or set processing. For example, obtaining the name of an employee’s manager is as simple as

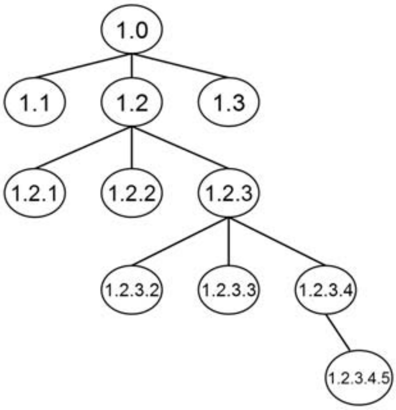


Figure 5.5: A hierarchical data model

```
select fname
   from employee
   where  emp_id = getparent(‘1.3.5.3.7’);
```

If you reorganize and move an employee (1.3.5.2) from her manager (1.3.5) to a new department and manager (1.6.1), you can “graft” the employee or even an entire branch of the hierarchy with a simple function call:

```
execute function graft (“1.3.5", ”1.6.1", “1.3.5.2")
(expression) 1.6.1.7
```

Columns defined as a **node** data type are limited to 256 bytes but are replicated through IDS’s replication mechanisms.

Concurrency and Optimization Enhancements

IDS 11 offers several new functional enhancements in this area. Perhaps the greatest is the addition of a new transaction isolation level; it will help prevent sessions that are changing data from blocking sessions that are reading data, and vice versa.

Concurrent Optimization

The default isolation level for a logged database is currently **committed read**. This level ensures that rows read and returned to a session have been committed to disk.

The instance checks to see whether a shared lock could be placed on the row in question before it is returned. It doesn't actually lock the row, so it acts like a dirty read in terms of speed. **Committed read** isolation does not prevent queried data from being changed by another process, even if the query is executed as part of an explicit transaction. It does, however, prevent the return of data locked for update, insert, or deletion by another session before the query is executed.

When this happens, the query must wait for the SQL Data Manipulation Language (DML) operation to commit or roll back before it can continue. From an end-user perspective, the database will appear to be hung or nonresponsive, and, depending on the length of time required to commit or roll back, an application or data server timeout may occur. The query will have to be re-executed, wasting time.

IDS 11 provides an enhancement that lets applications query committed data that another session could potentially change. The mechanism is a new **last committed** keyword option on the **committed read** and other isolation levels.

When **last committed** is active, if a query tries to read a row that's been locked for an SQL DML operation, the previously committed version of the row will be returned to the query. Transactional integrity is preserved because the returned value(s) haven't been changed; it's just possible they might change. Even if a change has occurred, the change hasn't been committed, so it's possible that the change could be rolled back. This behavior differs from the standard **dirty read** isolation level, which returns the current value of data regardless of its transaction state. As a result, queries can receive interim values that are not finally committed but are rolled back. At this point, the query result set contains "bad" data.

You can extend the **last committed** functionality to the **dirty read** isolation level as well. You enable the option either through the **last committed** SQL syntax or by using the **USELASTCOMMITTED $ONCONFIG** parameter (or an environment variable of the same name). When set through the **$ONCONFIG** file, the parameter overrides any session-specific setting.

IDS supports this functionality for operations against almost any data and index type other than **collection** data types and external tables created through the Virtual Table Interface (VTI). Obviously, you need to enable transaction support or a logging mode on the database and table (remember, you can have non-logged tables in a logged database) to return a previous view of a row.

A few restrictions apply to this functionality. For example, it isn't supported for operations against tables using page-level locking, tables locked in exclusive mode, or tables accessed through an R-tree index. It won't work in operations executed against a High Availability Data Replication (HDR) secondary or against Remote Standby Secondary (RSS) or Shared Disk Secondary (SDS) instance(s) because all they have is fully committed data.

Nor does the functionality work in distributed SQL operations or on operations executed against Enterprise Replication (ER) replicates. The **onstat** utility has been enhanced to show last committed values as well as the full isolation level.

Directives for ANSI-Compliant Syntax

Before Version 11, the use of optimizer **join** directives was restricted to operations using IDS-specific syntax or extensions. If an operation was written in strict adherence to ANSI syntax, directives weren't supported. Now, your application developers can use the following classes of directives:

- Optimization goal — **all_rows** or **first_rows**
- Access method — **avoid_index_sj**, **avoid_full**, **avoid_index**, and so on
- Explain — **explain** and **avoid_execute**
- Order — for **left outer** and **inner** joins only

Several ANSI **join** methods aren't currently supported, including **use_nl**, **avoid_nl**, and **hash** directives such as whether or not to use a **hash join** and which are the **build** and **probe** tables.

ISTAR Extended Type Support

ISTAR is the name of the server-side connectivity libraries facilitating client/server and instance-to-instance communications. Years ago, ISTAR (along with ICONNECT, the client-side libraries) was separately licensed and installed. Now, the server-side libraries are bundled with the data server, and you can select which client-side libraries you want through the Client Software Development Kit (CSDK), Java Database Connectivity (JDBC), or other drivers.

There were limitations on which servers could communicate with each other through ISTAR. For example Informix Extended Parallel Server (XPS), the Informix shared-nothing architecture data server, didn't support connectivity to IDS until recently. Although IDS-to-IDS instance communications was always supported, a few restrictions applied with respect to the types and functions you could use when executing operations to "remote" instances. All the basic built-in data types and standard SQL functions were supported, of course, and IDS 10 added support for most of the extended data types. With IDS 11, several additional types are now supported:

- **lvarchar**
- **boolean**
- **distinct of boolean**
- **distinct of lvarchar**
- **distinct** of non-opaque built-in types

Columns defined as **blob** or **clob** data types cannot participate, however. You also can't implicitly or explicitly cast a UDT to a supported type to execute a cross-database/instance operation.

This release also provides support for the execution of all kinds of UDRs in a cross-instance operation, not just those built using SPL. You can't, however, call a UDR with an explicit **out** parameter.

Derived Tables

Generally speaking, when an SQL operation is created, the **from** clause indicates the table, view, or iterator function that's the data source for the operation. IDS now supports the ANSI-standard syntax to use the results of an encapsulated **select** statement in the **from** clause as shown in Figure 5.6.

```
select sum(col_1) as sum_col1, col2
   from
     (select col_a, col_b from my_tab)
      as virt_tab(virt_col1, virt_col2)
   group by virt_col2;
```

Figure 5.6: Support for the ANSI select in the from clause syntax

With this functionality, your application developers may no longer need to create and populate temporary tables. The encapsulated **select** statement can execute almost everything a "normal" **select** statement can, including aggregation, sorting, grouping, and so on. You can create simple or more complex **join** conditions as part of the encapsulated **select**, including **union** and **outer**. This functionality should make it easier for partners and others to migrate their applications to IDS because this commonly used syntax no longer needs altering to run natively on the IDS data server.

Common Application API

One of IBM's goals is to ease the development burden for those creating applications for its data servers. To that end, IBM is developing a Common Client API for all its tier 1 data servers. With this API, developers can create their application once and be able to deploy against any of the supported IBM data servers. Obviously, the application must not use server-specific syntax to access functionality exclusive to one server because it won't work with another data server. Check the machine and release notes that accompany your distribution of IDS 11 to see which components of the API are available in this release.

Web Feature Service

Upon first glancing at this section title, you might think to yourself, "What, hasn't IDS supported Web browser syntax and Web publishing until now?" As the wise Master Po might gently counsel his protégé, "Grasshopper, you do not see clearly yet."

In this case, the “feature” being referred to is a geospatial object, such as a lake, mountain, or road, that has a name and a shape associated with it. This new data server functionality is an implementation of the Open GeoSpatial Consortium (OGC) Specification 1.1 of the Web Feature Service Implementation. In effect, an interface has been created that permits Web-based requests for geographical features to occur using platform-independent calls.

This functionality has been added to both the Geospatial and Geodetic DataBlades and incorporates XML with Geographic Markup Language (GML) extensions to encode the features being requested for transport to the calling function. Using these two DataBlades, you can now build database-driven services that can feed either flat-earth or round-earth (with a time dimension) data directly out of the data server into a Web-based or other graphical application for use with text or numeric data.

Index Self-Join Access Method

As I began to learn about this new feature, I thought this was really cool stuff! As you know, indexes permit relatively fast access to data based on discrete and ordered values. Indexes are most efficient when they contain only a few columns and the values in the columns are reasonably unique, especially in the first two to three columns of a composite index.

When indexed data values are not reasonably unique (e.g., male vs. female), a significant portion of the table’s contents will be returned, necessitating additional processing either by the application or by other data server function calls. This new feature leverages the data server’s query rewrite functionality to logically join the table to itself to create unique combinations of indexed values, resulting in extremely fast access to the needed data. In effect, a series of mini-index reads are consolidated to create the result set, as the next few figures illustrate.

First, consider the following SQL operation on a table with an index on columns c1, c2, and c3:

```
select * from tab
   where c1 >= 1 and c1 <= 3
     and c2 >= 10 and c2 <= 11
     and c3 >= 100 and c3 <= 102;
```

In IDS 10 and earlier versions, because of the highly duplicate values in c1 and c2, the optimizer wasn’t able to discard very many rows based on index values, as shown in Figure 5.7.

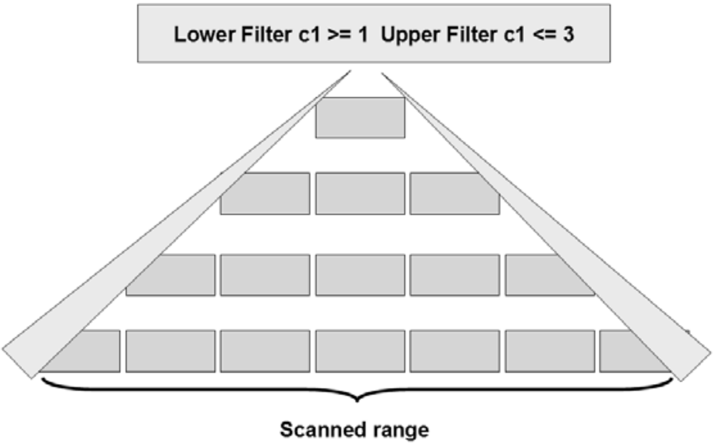


Figure 5.7: IDS 10 and earlier index exclusion based on highly duplicated index values

In IDS 10.xC4, the optimizer was able to somewhat intelligently apply an **upper** filter based on the original value ranges, which helped decrease the search range somewhat, as illustrated in Figure 5.8.

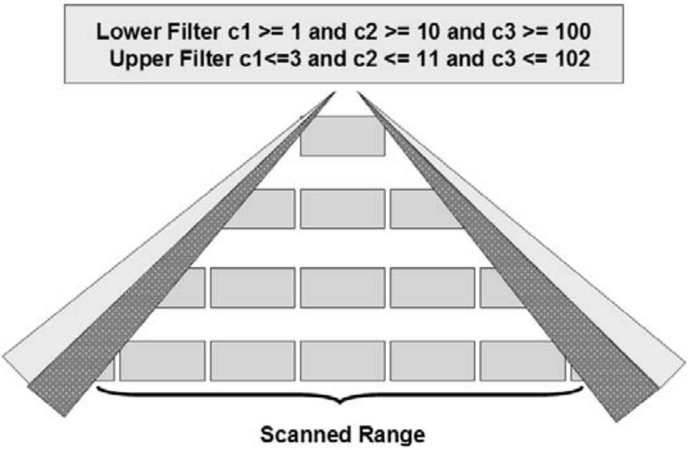

Lower Filter c1 >= 1 and c2 >= 10 and c3 >= 100
Upper Filter c1<=3 and c2 <= 11 and c3 <= 102
Scanned Range

Figure 5.8: Upper boundary exclusion available with IDS 10.xC4

In IDS 11, improvements to the optimizer enable it to intelligently rewrite the predicate clause into a series of scans joined through a **left outer join** as follows from the earlier example:

1. An index scan for unique combinations of c1 and c2 is made.
2. For each unique pair, an **inner** index scan is executed using the following filters:
   - c1 = c1
   - c2 = c2
   - c3 > 100
   - c3 < 102

The result, as Figure 5.9 shows, is a drastically reduced range of possible matches that are returned at index, rather than near sequential scan, speeds without the creation of implicit or explicit temporary tables.

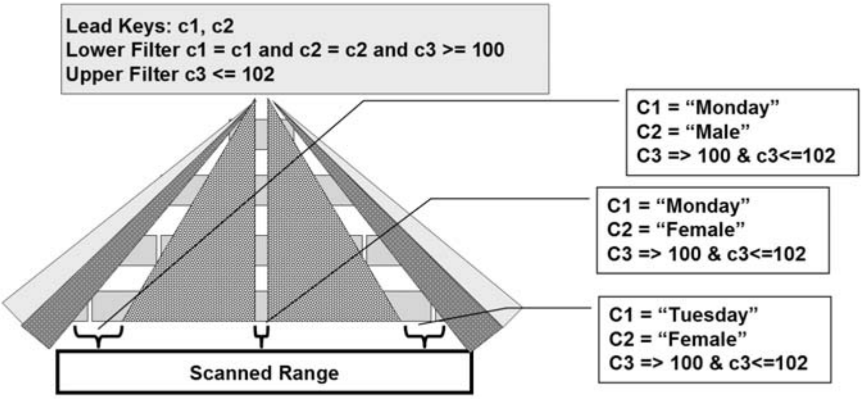


Figure 5.9: Data exclusion based on index self-join functionality

Named Parameters

In a function, you can reference input parameters in two ways: either by their ordinal value (first, second, third parameter passed) or by an identifier such as "cust_num," "amt_sold," and so on. Depending on which programming language you use when creating IDS-oriented applications, you have to know which language and interface supports which input-referencing method. For example, if you use Java, you have to use ordinal values.

IDS 11 supports the JDBC 3.0 specification permitting named parameters. With named parameters, your application developers can, based on the function, pass in only those parameter values that matter, leaving others empty to be filled with default values, or they can pass in parameters in any order.

Auto Re-Optimization of Prepared Statements and Stored Procedures

Before this release, UDRs written in SPL were parsed and optimized at creation time. Their SQL access plans remained static unless you executed an **update statistics** operation with the appropriate options to force the re-optimization of the procedures. If you didn't update the statistics regularly and the structure of the table changed, when the UDR was called, it would abort with a –710 SQL error code.

IDS 11 checks and, where possible, re-prepares a UDR written in SPL if there are changes to the table's schema. Under some conditions, the –710 error will still be returned — for example, on **select * from . . .** operations or where the data type of a column changes. The error is returned as a safety precaution to the calling operation because the column count is different and/or the data type has changed, and the calling operation may not be prepared to receive these changes.

You can turn on this functionality through the **AUTO_REPREPARE $ONCONFIG** parameter or the similarly named parameter to the **set environment** SQL command. If errors occur during the re-optimization process, they will be returned to the calling operation.

Competitive Stored Procedure Support

For me, there's nothing better than taking business away from a particular data server competitor. It's so easy from a data server perspective. This competitor has an antiquated architecture that can't scale and a horrendous ownership cost because of all the administrators needed to keep the data server running on a daily basis, not to mention the huge hardware bill required to get any performance. Their server has limited business-continuity functionality, and their main product actually decreases overall throughput as you add to the "cluster." From a technical perspective, I can walk over this one in my sleep.

They do have commanding mindshare, though, and many applications have been written that execute against their data server. IBM has a Migration Toolkit (MTK), downloadable for free from the IBM IDS Web site, that can easily move a database and its data from that source data server into IDS. What the toolkit has been lacking, though, is the ability to migrate stored procedures written in the data server's proprietary SQL syntax. This shortcoming has hampered the ability to migrate applications because most of them use stored procedures written in that language.

IDS 11 begins the process of including new functional extensions to the IDS SPL to provide similar functionality to the most commonly used competitive syntax. In addition, several keywords are being added to comply with ANSI SQL92 and later requirements.

This release adds the following functionality:

- Labeled statements and the **goto** operator — IDS 11 provides the ability to branch to a specific labeled area inside the SPL block, as shown in the first example in Figure 5.10. Some restrictions apply as to where the labeled statement may be and what its scope can include, so check the documentation for more information.
- **loop** statements with **exit** and **continue** keywords — To date, the procedural constructs in IDS SPL have been relatively limited, supporting just **if** and **case** statements. Now, you can create statement blocks that loop until or while a specific condition exists and then exit, as shown in the second set of examples in Figure 5.10. You can start the loops with the **loop**, **while**, or **for** keyword. A bit of a warning, though, on using this new functionality: Without careful debugging, you can create an infinite loop that never returns.
- Labeled **loop** statements — You can attach an identifier to a **loop** block, as illustrated in the third set of examples in Figure 5.10. This ability is supported only for loops constructed with the **while** and **for** keywords. You can use it to exit nested loop statements.
- Conditional labeled **loop** exits — Like the capability described in the previous bullet, with this support not only can you label a **loop** block, but you can also **continue** or **exit** from the loop based on a variable condition inside the **loop**, as shown in the last example set of Figure 5.10. Without this functionality, you'd have to write additional procedural steps to verify the variable's value and then take the appropriate action. The new support provides the same functionality with fewer steps. Like the previous capability, you can use it to exit from nested loop statement blocks.

```
Example 1:
define my_counter integer;
   let my_counter = 0;
   begin
        <the_beginning>
        begin
                let my_counter = my_counter + 1;
        end;
        if my_counter < 10 then
                goto the_beginning;
        end if;
   end;
end procedure;
```

Figure 5.10: Examples of new SPL procedural syntax

```
Example set 2:
loop
   if pressure is null then
      continue;
   if pressure >= 3 then
      exit; – exit loop immediately
   end if;
end loop;

while (i < 10) loop
   let i = i + 1;
   if  i < 4 then
      continue;
   if i  > 5 then
      exit;
   end if;
end if;

for i in (1 TO 5) loop
   let i = i +1 ;
   if  i < 5 then
      continue;
   if i  > 5 then
      exit;
   end if;
end if;

Example set 3:
<get_bigger>
while (i < 10)
   i = i +1 ;
end while get_bigger;

<<get_bigger2>
for i in 1..5
   i = i +1 ;
end for get_bigger2;

Example set 4:
while (my_count > 0) loop
let my_count = my_count + 1;
         exit when my_count = 4;
 end loop;

<outer>
loop
let x = x + 1;
  <inner>
  while ( i > 0 ) loop
    let x = x + 1;
    exit inner when x = 4;
    exit outer when x > 7;
  end loop inner;
let x = x+1;
end loop outer;
```

Figure 5.10: Examples of new SPL procedural syntax

Conclusion

Thank you for taking the time to read through this short introduction to Informix Dynamic Server 11. I hope you now understand how this data server is unique and unmatched in the marketplace in terms of functionality, ease of use, and performance. With it, you can build richer data processing functionality easily. Its business continuity features are deeper and broader and perform better than any other offering. The data server takes care of itself, letting you use hardware and personnel for other things that create greater business value.

IDS is, as the Olympic motto states, *Citius, Altius, Fortius*.

IBM Informix Dynamic Server 11

The Next Generation in OLTP Data Server Technology

IBM Informix Dynamic Server (IDS) is the optimal data server for high-transaction OLTP and integrated environments. Start learning about IDS today with this guide to the latest features and their value to your business. IDS helps businesses achieve Information On Demand by providing unmatched high reliability, "nearly hands-free" scalable administration, blazing-fast OLTP, and extensive platform coverage.

- Download IDS 11 trial software to benefit from the latest features: *ibm.com/informix/ids*
- Take your skills to the next level with *Administering Informix Dynamic Server, Building the Foundation* (MC Press, 2008), and IDS 11.
- Keep up to date on the available Information Management resources by visiting *ibm.com/software/data/education*.

CARLTON DOE had over 10 years of Informix experience as a DBA, Data Server Administrator and 4GL Developer before joining Informix in 2000. During this time, he was actively involved in the local Informix user group and was one of the five founders of the International Informix Users Group (IIUG). He served as IIUG President, Advocacy Director, and sat on the IIUG Board of Directors for several years. He is best known for having written *Informix-OnLine Dynamic Server Handbook* and *Administering Informix Dynamic Server on Windows NT* both of which are currently undergoing revisions for a second edition. He is the author of numerous IBM whitepapers and technical articles and co-authored several IBM Redbooks. Carlton currently works for IBM as the Informix Technical Specialist for the America's Geographies and is a Sr. Certified Consulting IT Specialist. He lives in Dallas, TX with his family and his neighbor's dog "puppy."

MC PRESS

MC Press Online, LP
125 N. Woodland Trail
Lewisville, TX 75077

Price: $14.95 US/$18.95 CN

ISBN 978-1-58347-075-6
51495
9 781583 470756