WebSphere RFID Information Center 1.0 – A performance profile

Document version 1.0

Raghu Sakleshpur

Performance Lead - RFIDIC team

Contents

1

Introduction

INTRODUCTION

In addition to testing performance of the product we also provide implementation guidelines

This paper describes our performance evaluation of WebSphere RFID Information Center version 1.0 (RFIDIC) as used in a manufacturing and distribution center of a supply chain network. We provide information on product performance, a description of our solution implementation and guidance on how to devise and implement an RFIDIC solution smoothly while avoiding common pitfalls

RFIDIC 1.0 has been developed as per the EPCglobal Electronic Product Code Information Services (EPCIS) 1.0 specification. A brief overview of RFIDIC 1.0 is provided in the next section of this chapter. For a comprehensive description of RFIDIC 1.0 see reference [1].

EPCIS is a fairly new standard and as of yet there is no common industry specific benchmark. We developed a typical RFID solution based on query and event volume data for a midsize packaging plant and distribution center of a large US pharmaceutical distributor using RFIDIC version 1.0. We used this solution to test, measure, and quantify the performance of the product.

This chapter provides an overview of the RFIDIC product. The second chapter describes the use case solution that was used to benchmark the product. The third chapter provides implementation details for the solution and describes the tests that were performed for measurement. The fourth chapter covers results and findings. The fifth chapter provides a discussion of the Informix database tunings that were used and includes guidelines for an implementer. Finally, we provide a chapter on our conclusions, a glossary and a list of Appendices and References.

INTRODUCTION

Performance Testing focuses chiefly on the following components

- Event Capture
- Query Interface

Overview of Websphere RFID Information Center 1.0

The WebSphere RFID Information Center product provides a scalable and secure repository for data generated by sensor networks for any supply chain environment. It is compliant with the EPCglobal EPCIS 1.0 specification.

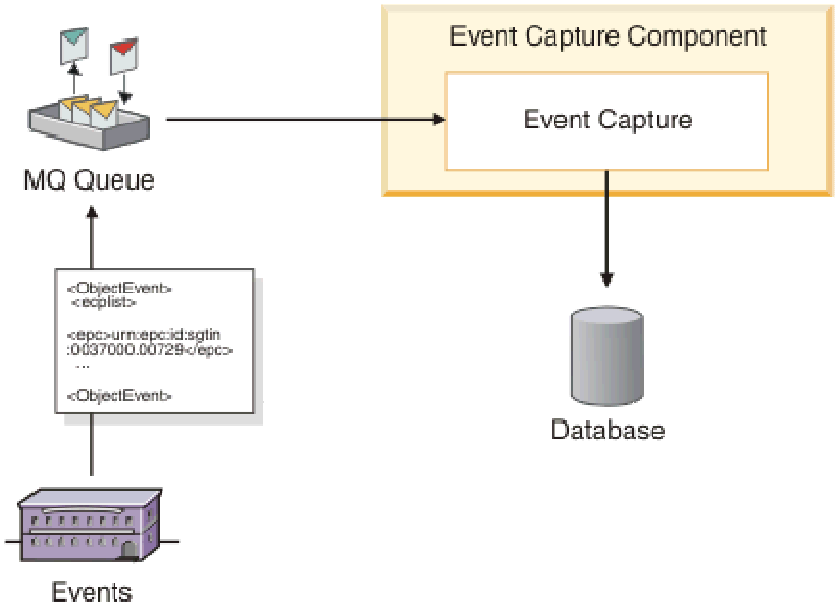
The RFIDIC 1.0 product can be divided into the following components.

- Event capture
- Query component
- Reporting
- Metadata
- MetaData management (MDM)
- Security

Metadata, MDM and Security components are used indirectly in all solution implementations. Event Capture, Query component and Reporting are the primary focus for measuring performance in this paper and are introduced in the following sections.

Event capture

The primary function of the Event capture component is to read incoming events (in xml format) and save them into an embedded database. RFIDIC 1.0 uses Informix Dynamic server for the backend database engine. The events can be read from a static file on the filesystem or from a Java message queue server. We used WebSphere Message Queue server, bundled with the product. Figure 1.0 illustrates the event capture process.



Query

The query component allows the user to query persisted data from an Informix database. Queries can be run in two ways.

- Webservices
- Data Browser

The Webservices interface is a programming API for client applications for querying and browsing the data. The Data Browser is more of an out of box interactive user interface to query the underlying database using a standard Web browser.

The Data browser lets users query from any remote system and has the ability to query both events and MDM data such as products, locations and hierarchies. It allows the users to create custom queries on event data and has the ability to save them for future sessions.

We tested querying with both Webservices and the Data browser.

Reporting

RFIDIC 1.0 includes the IBM Alphablox product for reporting. The product has also been tested to work well with the open source reporting tool called BIRT (Business Intelligence and Reporting Tool). BIRT is an Eclipse-based tool that integrates easily with RFIDIC 1.0 and provides reports in browser or PDF formats.

RFIDIC 1.0 is a Java application that is packaged with the following products that can be used in an RFIDIC solution:

- Informix Dynamic Server version 10.0
- Tivoli Directory Integrator version 6.1
- WebSphere Application Server (Express edition) version 6.0.2
- WebSphere Message Server 6.0.1
- Alphablox version 8.4

Our implemented solution used all of these products. For reporting, however we did not use Alphablox. Instead, we tested with SQL queries using the SQL interface to the Informix Dynamic Server. We do not expect to see any performance implications with queries being submitted via a reporting tool versus the Informix SQL interface

2

A Pharma Case Study

In order to accurately benchmark the product we developed an RFIDIC 1.0 solution for a typical supply chain use case, with a pharmaceutical packaging plant and distribution center.

We used two testing scenarios to test the implemented solution.

- Tests to measure response time
- Tests to load and stress the system.

We also tested typical hardware configurations to determine optimal requirements for memory, CPU and disk space.

Events

In this use case, we assumed that RFID readers are strategically placed in a packaging plant and a distribution center at five different locations to capture events as they occur. These events are typically transmitted via Java Message Queues to our solution by a middleware application such as WebSphere Premises Server.

Each location corresponds to a read point at a business location. The observance of an object at a location by the RFID reader at a given time would constitute an event. We describe the five different types of events used in our test case below. We also assumed eight packaging lines in the packaging plant.

Bottle commissioning event: In the packaging plant, a bottle is the smallest unit with an RFID tag. At this location a new bottle gets commissioned with an RFID tag twice every second per packaging line, resulting in 16 bottles being commissioned every second. We provide the XML description for a typical bottling event in Appendix A.

Case commissioning event: An empty case gets commissioned with an RFID tag once every 72 seconds on each packaging line. Case commissioning events look very similar to the bottle commissioning events. The EPC tag used for each case will be that of the parent tag type in the XML event description provided in Appendix A.

Unit to case aggregation event: At this location, the RFID tag on each bottle packed into a case gets associated with the RFID tag of the case. A typical case contains 144 bottles. This event occurs once every 72 seconds per packaging line. Only those EPCs that have been commissioned previously will be present in these events. The XML description for the aggregation event contains one parent EPC for the case and a child EPC for each of the 144 bottles in the case. Examples are provided in Appendix B.

Turntable verification event: By the time packaged objects move across the RFID reader at this location, the individual cases are already packaged into a pallet. A pallet contains 50 cases and therefore contains 7250 EPCs in one single event. This event occurs once every 3600 seconds per packaging line. A typical Turntable verification event is an Aggregation event similar to the one in Appendix B with 50 parent (aggregated) EPCs.

Shipping portal event: This is the last read point before the pallet gets shipped. This event occurs once in 3600 seconds per packaging line as well. The XML event description at this location is exactly the same as the event in Turntable verification step, except for the difference in read point location and business location fields.

The five events are summarized in table 1 below.

Event Description	Arrival Rate for 1 queue (Bottling Line)	# of EPCs per event	Total event arrival rate for 8 queues (Bottling Lines)	Total EPC arrival rate
Bottle Encoding	2 every second	1	16 every second	16 every second
Case Commission	1 every 72 seconds	1	8 every 72 seconds	8 every 72 seconds
Unit to Case Aggregation	1 every 72 seconds	145	8 every 72 seconds	1160 every 72 seconds
Turntable Verification	1 every 3600 seconds	7250	8 every 3600 seconds	58000 every 3600 seconds
Shipping Portal	1 every 3600	7250	8 every 3600	58000 every 3600

	seconds		seconds	seconds

table 1. Event and arrival Summary

Queries

The solution supports querying the data in parallel as the events are captured and persisted in the database. The queries are generated by the Distribution Center (DC). The DC has the following processes:

- DC Receiving
- DC Picking
- DC Shipping
- Product Authentication

Table 2 below summarizes the queries generated by each of the processes in the DC.

Query	Description	Arrival Rate	Source
getBatchNumber	Returns the Batch Number for a given EPC. 2 queries per case.	30 cases/min	DC Receiving
getChildren	Returns all the EPCs for a given parent EPC. 2 queries per case	30 cases/min	DC Receiving
getParent	Return parent EPC for a given child EPC	150 units/min	DC Picking
getProduct	Return the product GTIN for a given EPC	150 units/min	DC Picking
getHierarchyNodes	Return zero or more product hierarchy nodes for a given EPC.	150 units/min	DC Picking
getBatchInfo	Returns batch number and disposition	150 units/min	DC Shipping
getEventInfo	Returns object event with commissioning step else a	150 units/min	Product Authentication

	query parameter exception		

Table 2. Query Summary

The solution implements these queries to run as a Java client via the Webservices protocol supported in RFIDIC 1.0.

Reporting

The solution required two types of reports. These reports had to be executed on a database that already contained captured data. The reports were run in parallel with event capture and simple query execution.

Report1: Count and group EPCs by location for one hour.

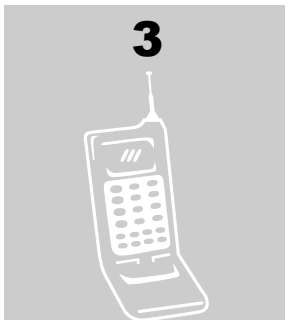
Report2: Count and group EPCs by location for one week.

Product and location data

We assume the Master data in this solution consists of 62 different products. These products are grouped under two hierarchies. The first hierarchy has 31 products under Hierarchy A and the second hierarchy has 31 products under Hierarchy B

For locations, we had 10 locations including the 5 locations already described for the 5 events above. All the locations are grouped under a single hierarchy.

Although no Master Data figures in event capture measurements, some of the queries and reports have to access Master Data tables.



Implementation details

This section provides details on the solution implementation, the hardware and software configurations and the tests that were actually performed.

Hardware configuration

We used an IBM xSeries 4 CPU (8 core) Xeon processor system with 8 GB memory (xSeries 366) with a RAID array (DS4300 with EXP710) for disk storage. The storage array was directly connected to the server under test. We located the database data files on a Linux filesystem created on a RAID 5 array.

Software configuration

Red Hat Linux AS4 (Update 4) was the operating platform. We, installed, configured and used the following IBM blue stack software bundled with RFIDIC 1.0 distribution on the test system:

- Informix Dynamic Server version 10.00 UC5
- WebSphere Application Server - Express version: 6.0.2.0
- IBM (Tivoli) Directory Integrator version 6.1 and version 5.2
- WebSphere Message Queue Server version 6.0.1

Solution specifics

For Meta Data, we made the following modifications to the default EPCIS MetaData.xml file:

- Created a separate parentID table. We took this step primarily to improve query performance and avoid a join of the childEPC and event tables in the embedded Informix database. The childEPC, Event and EPClist were the most populated tables.

- Interned all possible fields. We discuss the benefits of interning in the next section.

- Added an additional field for batch number in the event table because our events contained batch numbers which was an extension field to the standard EPCIS schema.

We created the RFIDIC database schema using the deployMetadata.sh script. This script is part of the RFIDIC 1.0 distribution.

We loaded the Product and location data, using the import-masterdata.sh script. This is part of RFIDIC 1.0 product.

We wrote a separate EPC to GTIN Handler to intern EPCs in the EPClist, parentID and childEPC tables. To intern all other fields (especially those in the event table) we used the SimpleVocabHandler that is distributed as part of the samples with the product.

We wrote solution queries in Java to be executed via Webservices interface from a remote client.

We wrote assembly line scripts in IBM (Tivoli) Directory Integrator (TDI) to read events from WebSphere MQ and pass the events to the RFIDIC event capture component. For more information on TDI and writing TDI assembly scripts see reference [4].

We created event files to simulate realistic data using ToXGene an XML generator tool. For more information on ToXGene see references [5].

We used an internally developed Java tool to read generated event files and write to a message queue to throttle the rate at which messages were written to the queue. We needed to throttle the rate of incoming events to reflect the realistic scenario of incoming events at the rate described in Chapter 2 Table 1.

We wrote the TDI assembly lines to wait for incoming events on a message queue. The events were immediately passed onto the Event Capture component of RFIDIC 1.0 as soon as they showed up on the queue.

Queries

We implemented simple queries using the Webservices solution answer query mechanism in RFIDIC. We wrote these queries in Java. We developed a shell wrapper to execute these queries at various intervals. We also used the NamedAnswerSolutionQueries.wsdl schema distributed with the product under RFIDIC_HOME/schemas to execute the queries.

For reporting queries, we used SQL to write and execute the queries locally on the system running the RFIDIC solution. Using RFIDIC 1.0, we do not expect to see any significant performance implications, running the reporting queries locally via SQL interface versus running them remotely via a reporting tool such as Alphablox or BIRT.

Tests performed

We performed two categories of tests. One focused on measuring the response time. The other focused on volume loading and stress testing the system.

In each of the test scenarios described in table 3 below, we ran TDI assembly lines to read from queues and write to the Event Capture component. These assembly lines waited until the next event showed up on the queue. We throttled the writing of events to the queue to simulate realistic event arrival rates. We submitted Webservices queries via the webservices client. We submitted reporting queries as SQL scripts that would be submitted via a reporting tool such as BIRT or Alphablox.

In the case of event bursts we did not throttle the incoming event rate. All events were written to the queue one after the other to get consumed by event caapture interface right away.

To stress the system for continuous periods of time, we ran the tests for atleast 72 hours continuously and ensured that the system was up and accessible.

Scenario	Event arrival	Query Arrival	Description
Event Capture	Refer to table 1	Refer to table 2	Ran and measured event capture performance. Wrote events into 4 JMS Queues and 4 TDI assembly lines.
Event Capture	Refer to	Refer to table	Ran event capture from 4 different queues using 4 different TDI assembly lines.

+ Query	Table 1	2	Executed simple queries in Java from a remote client via Webservices to simulate query arrival rate
Event Capture + Query + Reports	Refer to Table 1	Refer to table 2	Ran event capture from 4 different queues using 4 different TDI assembly lines. Executed simple queries in Java from a remote client via webservices to simulate query arrival rate. Ran two report SQL queries on the system continuously once every half hour for 72 hours.
Event burst Capture	Events written into queues unthrottled	Queries were submitted in a continuous loop	Ran event capture from 4 different queues using 4 different TDI assembly lines.
Event burst capture + query + Reporting	Events written into queue unthrottled	Queries were submitted in a continuous loop	Ran event capture from 4 different queues using 4 different TDI assembly lines. Executed simple queries in Java from a remote client via Webservices to simulate query arrival rate. Ran two report SQL queries on the system continuously once every half hour. Ran tests for 72 hours.

Table 3. Test Scenarios

Findings & results

Because RFIDIC 1.0 provides a platform to implement solutions, we had to create a typical solution to quantify its performance. Like any solution built on a database server, the performance of any implemented solution is greatly affected by the underlying database schema and database tuning.

RFIDIC 1.0 allows solution developers to use metadata to cleanly define solution-specific schema changes to the default EPCIS metadata for a database schema. In designing the database schema, we had to consider the nature and contents of the events that were persisted and also factor in the type and frequencies of queries and reports that would be executed on the persisted data.

In our solution we decided to separate the parentID field (parent EPC) into a separate table. This helped reduce the disk space consumed by the event table because object events contain a parentID field. The parentID field is a variable character string field of length 100. This space would be empty for all object events in the event table. Separating the parentID into a parentID table also improves queries that need to join childEPC table and the parentID tables because the parentID table would be much smaller in size compared to the event table.

Event capture performance

The Event Capture component of RFIDIC 1.0 is quite light weight and typically gets executed as a standalone Java application. It is not CPU intensive and most time was spent in disk I/O in all the tests that were done. As the size of the database grows, indexing the database and striping the DBspaces across multiple disks are crucial to improving performance.

For large event objects typically as in aggregation events, increasing the JVM stack size benefits performance. This can be easily achieved by

modifying the script rfidic-tdiserv.sh to invoke the JVM with larger stack size using the –Xmx option.

We also observed that event capture scales across multiple queues quite efficiently. In all our testing with event capture, we tested with multiple queues (up to 5 of them). We executed each assembly line in a separate JVM, connecting to a different queue.

Query performance

We looked at two types of non-OLTP type queries: simple queries and reporting queries. By definition, these queries did not update any data but only queried the database to return suitable values.

Simple Query Performance:

For each field in a table on which a join or a search is performed, create an index on that field for the table.

Simple queries were queries were written to obtain specific results given specific input parameters. We created indexes on all tables on the columns that were involved in the join operation for a query. We analyzed the query plan to create suitable indexes depending on the query being performed.

For each field in a table on which a join or a search is performed, we created an index on that field for the table. This resulted in an index scan of the table being searched, instead of the time consuming sequential scan.

After creating the required indexes for each query, all queries returned in well under a second.

Findings summary

Query response time for simple queries was consistently below one second, even on a database with upwards of 420 million rows. We executed the queries repeatedly (as per the arrival rate cited in chapter 2) in parallel with event capture.

We found that the queries executed via webservices from a remote client were equally fast, and always returned under one second.

When newer queries are added, either existing indexes need to be modified or newer indexes may need to be created. Data on time consumed is summarized in table 4.1.

Reporting

Reporting queries are longer queries that involve summarizing data over periods of time. In addition to Informix tunings and indexing, these queries also require partitioning or fragmenting indexes by time. This is described in greater detail in Chapter 5 (Database tuning).

Findings summary

Report queries once cached easily returned in less than one minute. Fragmenting queries involved creating indexes on the event table by fragment index. In our experiment data was fragmented by month. We found that fragmenting by month was sufficient to get acceptable performance.

We tested the reporting queries in conjunction with event capture as per the scenarios outlined in chapter 2. Table 4.1 summarizes reporting query response times.

The maximum response time was observed for both reports when the reports were run the first time after a reboot of the system. The high response time was due to the fact that the database shared memory segment containing the buffer cache was empty and each row access would result in a disk I/O. Once the buffer cache warmed up the reports consistently ran under a minute.

QueryType	Max Response	Avg Response	Min Response	Notes
Reporting Query1	6min 05 seconds	56.865 seconds	54.257 seconds	Returned 6 rows
Reporting Query2	6min 20seconds	58.524 seconds	55.013 seconds	Returned 6 rows
Simple Query1	0.454 seconds	0.245 seconds	0.105 seconds	Returned 1 row
Simple Query2	0.895 seconds	0.496 seconds	0.194 seconds	Returned 432 rows
Simple Query3	0.396 seconds	0.260 seconds	0.0886 seconds	Returned 3 rows
Simple Query4	0.422 seconds	0.292 seconds	0.0925 seconds	Returned 1 row
Simple Query5	0.315 seconds	0.143 seconds	0.0430 seconds	Returned 1 row
Simple Query6	0.475 seconds	0.278 seconds	0.0786 seconds	Returned 1 row

Simple Query7	0.310 seconds	0.201 seconds	0.0445 seconds	Returned 1 row

Table 4 Query respone times

Database sizes

We built and tested the data sizes to simulate the existence of 3 months, 6 months and one year of data. We conducted the tests after capturing 100 million events, 300 million events and 425 million events. We tested and measured query and event capture performance on the data that was already saved in the database.

The power of interning

Interning

Internal numeric representation in memory of a large string or object in the database

In RFIDIC 1.0, we use a technique referred to as "interning" to convert a string (VarChar) data object to a canonical (numeric) representation. Typically we write a handler to "intern" and "unintern" fields in the event XML file. The reasons for the advantages of interning are two fold.

- At the time of event capture, the handler will intelligently avoid a database access if the interned value of a field had already been cached in the memory pool. Since most fields in the event table are known to have values over a fixed small range, interning will greatly improve the speed of event data persistence.

- The data value stored in the database for the particular object is also the interned numeric value. This approach greatly reduces the consumed database space.

Table 5 compares and highlights the benefits of interning. As noted in the table, not saving the XML as a blob also greatly helps in reducing the size of the overall database.

For this particular experiment, we used a simple object event that is 918 bytes long. We captured a total of 10,000 events using the RFIDIC 1.0 event capture interface to obtain the numbers. With interning and without saving the entire XML event description as a blob in the database, we found that the database consumes less space than the space consumed by the original raw XML text.

Note that the total database size measured does not include indexes that need to be created for optimal query performance.

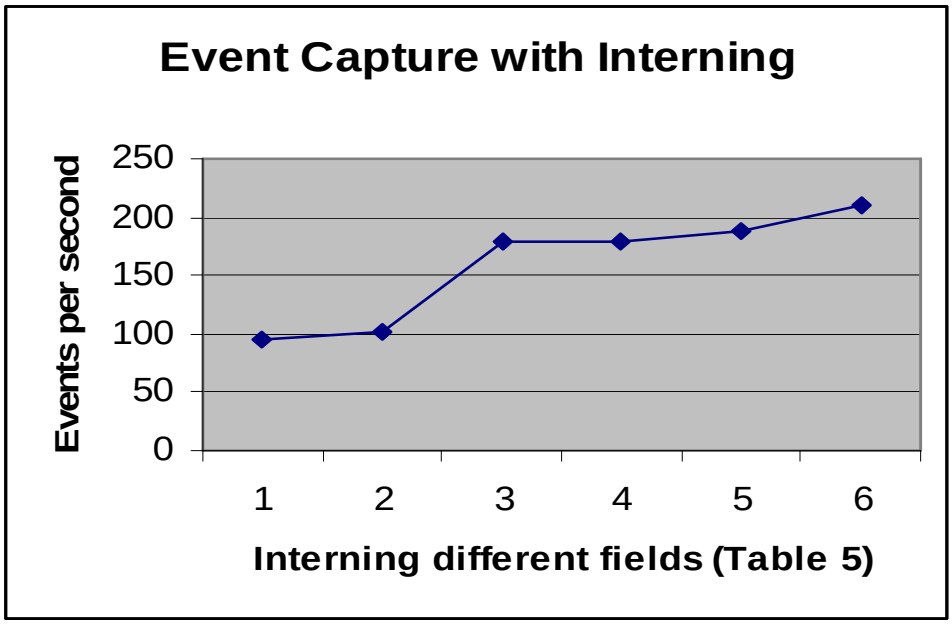
Particulars	Time taken	Database size
No interning Xml event saved in blob	1m56.000seconds 94.33 events/sec	~22.27 MB
No save of xml No interning	1m39.008seconds 101.01 events/sec	~21.31 MB
No save of xml No save of header No interning	55.976 seconds 178.647 events/sec	~7.927 MB
No save of xml No save of header Interning: ReadPoint, bizloc, epc, action, bizStep, disposition	55.669 seconds 179.633 events/sec	~6.44 MB
No save of xml No save of header Interning ReadPoint,bizloc, action, bizStep, disposition	53.45seconds 187.09 events/sec	~5.96 MB
No save of xml No save of header Interning: Readpoing, bizloc, action, bizStep, disposition TDI:2 assembly lines to read	47.428 seconds 210.845 events/second	~5.96 MB

Table 5. Event capture response and database size

Below we provide two charts to highlight the benefits of interning. In both charts, each co-ordinate on the X-axis corresponds to a row in table 5.

Chart 1 shows that event capture rate increased with each additional field in the XML event description that was interned.

Chart 2 shows that database size decreased as we increased the number of fields that were interned in the XML event description.



Char :1 Event capture with various fields interned (Table 5)

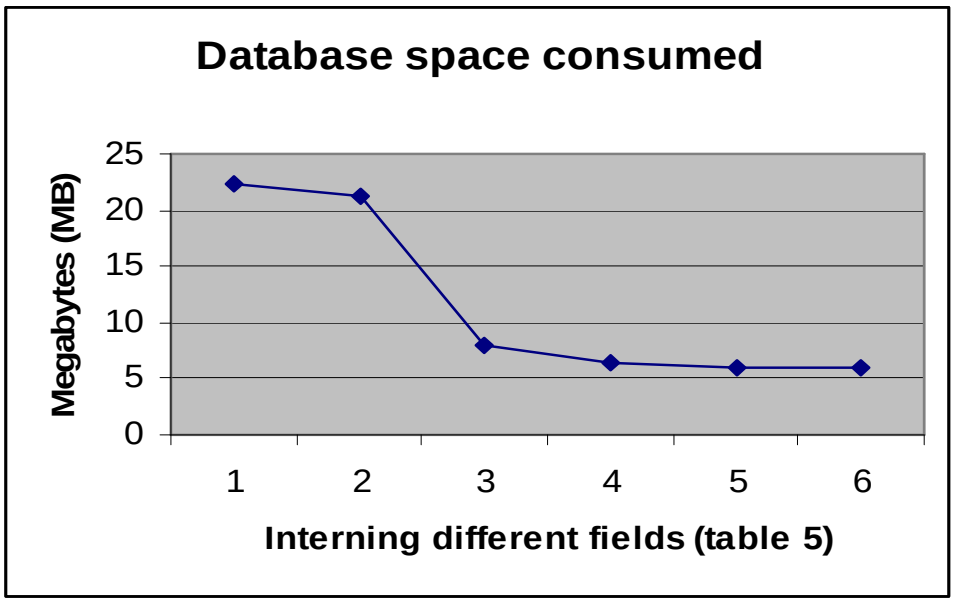


Chart 2: Database space consumed with various fields interned (Table 5)

5

Database Tuning

This chapter highlights some of the tuning we performed to improve performance of the test solution.

Although prior working knowledge of database servers is recommended, one need not be an Informix database administrator to implement these tunings.

Once we finalized the database schema and captured the data, we tuned the database and added appropriate indexes for optimal query performance.

Informix server configuration

Identifying and separating the different database tablespaces was the first step towards achieving a high performing database configuration. We created the following dataspaces to span multiple different disks (volumes). We created more than one dbspace for tablespaces and took care to distribute them onto different physical disks.

- Physical Log Space
- Virtual Log Space
- Disk space for temp space
- Disk space for tables
- Disk space for indexes

We provide example scripts to create dbspaces, temp space and virtual log files in Appendix C.

The table below describes onconfig (Informix server configuration file) file variables pertaining to logging and temp files that can be updated and tuned for better performance .

Tunable	Description	Notes
PHYSFILE	The PHYSFILE configuration parameter specifies the size of the physical log. This parameter indirectly affects checkpoints because whenever the physical log becomes 75 percent full, a checkpoint occurs.	Having a larger value set of 500000 to 1000000 reduces the frequency of checkpoint.
PHYSDBS	Points to the physical log file name. By default this would be pointing to rootdbs.	This needs to be updated to the name of the physical log space that gets created.
LOGFILES & LOGSIZE	Specifies the number of logical-log files that the database server creates during disk initialization. To change the number of logical-log files, add or drop logical-log files. LOGSIZE specifies the size of each virtual log-file.	Onparams command can be used to add and drop logical-log files. This value gets updated automatically when onparams is used. Having a larger number helps reduce the occurance of checkpoint for write intensive activities.
DBSPACETEMP	By default the variable is blank, implying that the tempspace would be configured on rootdbs.	Set this to point to the value of temp spaces that are separated by colon.

Table 6: database tunables

Informix Dynamic Server supports fragmenting indexes by expression. We provide an example script that creates indexes fragmented by months. See Appendix D. Queries need to specify the fragment to search in as part of the query where clause. This is especially important with reporting queries and will greatly reduce the cost of the query performance.

Disks and Filesystem

In most of our testing we used RAID 5 disks/volumes spanning 3 or more disks. A Linux ext3 filesystem was created on each RAID 5 volume. Separate, Informix dbspaces for log space, tempspace, data space and index space were created. These dbspaces were distributed on different RAID 5 volumes. Informix refers to this as the cooked mode of writing to disk. RFIDIC 1.0 only supports writing to disks in cooked mode.

In our experiments we observed better event capture throughput when disks were used without configuring RAID 5 volumes. We also observed that distributing the various db spaces across multiple RAID5 volumes on separate sets of 3 or more disks helped the system to perform better especially during Event Capture.

Shared memory tunables

Shared memory tunables that affect the size of the shareg memory segment are discussed in the table below. The overall goal should be to allocate and create the largest possible shared memory segment to avoid disk access for queries. All variables listed in table below can be tuned in the IDS configuration (onconfig) file.

Tunable	Description	Notes
LOCKS	Specifies the number of locks in the internal lock table	Lower values helps in not allocating memory needlessly.
CLEANERS	Specifies the number of page cleaner threads available for IDS	Having a large number helps reduce foreground reads as most page cleaning will happen in background.
SHMVIRTSIZE	Specifies the virtual portion of shared memory that needs to be allocated at IDS startup time.	This value must be as large as possible. The buffer cache resides in this segment and the larger the size the better the performance for reporting type queries.

Table 7: database shared memory tunables

Parallel data query tunables

Most of the tunables in the next table, except for PSORT_NPROCS and PDQPRIORITY are all tunable in the IDS server configuration (onconfig) file.

Tunable	Description	Notes
MAX_PDQPRIORITY	The max value that can be set is 100. Reducing this to a lower value gives higher priority(allocates more resources) to write/update activities.	Setting to the max value of improved report query performance.

PDQPRIORITY	Tells the IDS server the percentage of MAX_PDQPRIORITY to allocate resources for a particular query	This tunable needs to be set using th SQL set clause as part of the query. Setting to 100 tells the IDS server to allocate all DS query resource for the particular query.
DS_TOTAL_MEMORY	Informs IDS the percentage of SHMTOTAL to use for decision queries.	Set to a large value that is as large as possible. Setting it to same value as SHMVIRTSIZE gave better report query peformance
DS_NONPDQ_QUERY_MEM	IDS uses this value to allocate memory resources for non-PDQ memory	This value must be low. Set it to 128 for better report query performance
PSORT_NPROCS	When set as an environment variable it tells IDS the number of parallel sort threads the application can use.	Max value can be 10. Typically set it to the number of cores/CPUs on the system. On a 8 core box, setting it to 8 yielded best query performance.
BUFFERS	Number of BUFFERS in the resident portion of memory that cache database pages.	For DS queries, increasing BUFFERS too much will come at the cost of DS_TOTAL_MEMORY. Optimal query performance was observed when set to values between 50000 to 60000
lru_min_dirty, lru_max_dirty	These control how often pages get flushed to disk between full checkpoints.	When BUFFERPOOL is large, full checkpoints can take up system resources. Optimal values are to set 50 for lru_min_dirty and 60 for lru_max_dirty.

Table 8: Parallel Data Query tunables

Guidelines for Implementers

Solution implementers should plan the solution deployment with sufficient storage capacity. They need to ensure that the embedded database is sized and created for optimal performance. The following guidelines are provided to assist in this process.

- Estimate the total number of events that need to be handled over a period of one year. This information will help to determine the amount of disk space that will be required for deployment.

- Plan the storage (disks) deployment for best performance as well as for redundancy and integrity. See the section above on Disks and Filesystems for additional information.
- Tune the metadata for the database schema taking into consideration the incoming event contents (fields) as well as the type of queries (information) that will be desired based on the persisted data.
- Make use of interning to save database space consumed and improve event capture speed. Typically most fields of an event are internable including the EPC string!
- Based on our experiments, we found that an xml event could potentially take as little as ~60% of the actual xml event size in bytes, to persist in the database. At the other extreme it could take upto ~230% the size of an xml file to persist without any interning and persisting the headers and the xml event blob in the database. Refer to Chapter 4 (Power of interning) for additional information.
- Create indexes to improve query performance. Ensure that the query uses index path to search tables using in the query plan. Use the SET EXPLAIN ON directive in SQL to obtain the query plan.
- Layout the database dbspaces across multiple disks to avoid a single disk becoming an I/O bottleneck.
- Create indexes fragmented by an expression on the event table. Typically fragmenting the index by month will be sufficient to get optimal response times for reports. Refer to Appendix D for an example index creation script. Use the expression in the SQL queries.
- Tune the shared memory virtual size for best query and reporting performance. The tunables are discussed in Table 7 and Table 8 above.

Conclusions

We base these conclusions about RFIDIC 1.0 as an outcome of our performance and stress testing.

It is feature rich

We found in our testing that RFIDIC 1.0 is a highly scalable, adaptable development platform. We found that the product had enough features to allow a solution developer to build quick solutions out of the box and allow solution specific customizations. In our solution development, we found the ability to define and customize database schema accoring to the solution requirements quite was very useful. The Webservices support to query the database made it easier to allow existing applications or user interfaces to connect to the database repository.

It is fast

Both event capture and query access exceeded our expectation for our initial scenario solution requirements. Event capture was fast enough to persist hundreds of events/second even when the database server was tuned to support data querying and the database had more than 430 million rows in some tables.

Query response times were consistently under one second even when the database had upwards of hundreds of millions of events (table rows). The query response time was not affected in the stress testing scenarios where queries were executed in conjuction with event capture.

Reporting queries required tuning at the database server end, especially when the underlying database size started to increase beyond 100 million rows. Once we distributed the dataspaces across multiple disks, indexed tables with fragments, and tuned the database server for query performance, we saw vast improvements in query response times and the

reporting queries were able to return under a minute consistently. We describe the tuning details in Chapter 4.

It is scalable

We found our solution scaled across a 2 CPU desktop server as well as on a 4 CPU lab server. Query response time was consistent across all the database sizes that we tested with. The largest database that we tested was one having upwards of 510 million events. Query response time scaled well (response time decreased) with increasing memory size for the parallel data query cache. Event capture response times were consistent both on a low-end server with smaller sized database (tens of millions of rows) and the high-end server with hundreds of millions of rows in the table.

It is durable

As part of our testing scenario, we did stress testing that ran both event capture with event bursts (sending events unthrottled into the queue) and querying repeatedly both simple and reporting for a total of 72 hours. The tests ran well beyond 72 hours and we were able to stop and shutdown the tests and product gracefully.

Appendix

Appendix A

Example Bottle Encoding Event:

```
<?xml version="1.0" encoding="US-ASCII"?>
<!-- generated by ToXgene Version 2.3 on Fri Aug 25 15:54:50 PDT
2006 -->

<epcis:EPCISDocument xmlns:epcis="urn:epcglobal:epcis:xsd:1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:epcglobal:epcis:xsd:1 .\EPCIS.xsd"
schemaVersion="1" creationDate="2005-07-11T11:30:47.0Z">
 <EPCISBody>
   <EventList>
     <ObjectEvent>
       <eventTime>2005-04-03T01:01:01.82706:00</eventTime>
       <recordTime>2005-04-03T01:01:01.827-06:00</recordTime>
        <epcList>
       <epc>urn:epc:tag:sgtin96:2:000173.0069101.13777218</epc>
        </epcList>
       <action>ADD</action>
       <bizStep>BOTTLE ENCODING</bizStep>
       <disposition>ACTIVE</disposition>
       <readPoint>
          <id>urn:epcglobal:fmcg:loc:0000000089101.RP-1527</id>
       </readPoint>
       <bizLocation>
       <id>urn:epcglobal:fmcg:loc:0000000089101.A23-49</id>
        </bizLocation>
        <extension>
          <batchNumber>40</batchNumber>
         </extension>
    </ObjectEvent>
   </EventList>
  </EPCISBody>
</epcis:EPCISDocument>
```

Appendix B

Example of Unit to Case Aggregation Event.

```
<?xml version="1.0" encoding="US-ASCII"?>
<!-- generated by ToXgene Version 2.3 on Wed Sep 06 02:49:35 PDT
2006 -->

<epcis:EPCISDocument xmlns:epcis="urn:epcglobal:epcis:xsd:1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:epcglobal:epcis:xsd:1
../../../EPCglobal-epcis-1_0.xsd" schemaVersion="1"
creationDate="2005-07-11T11:30:47.0Z">
  <EPCISBody>
    <EventList>
      <AggregationEvent>
        <eventTime>2005-04-03T01:01:01.827-06:00</eventTime>
        <recordTime>2005-04-03T01:01:01.827-06:00</recordTime>
        <parentID>urn:epc:tag:sgtin-
96:1:000173.0069101.86777218</parentID>
          <childEPCs>
       <epc>urn:epc:tag:sgtin-96:2:000173.0069101.13777218</epc>
       <epc>urn:epc:tag:sgtin-96:2:000173.0069102.13777219</epc>
       <epc>urn:epc:tag:sgtin-96:2:000173.0069103.13777220</epc>
                          …
                          …

       <epc>urn:epc:tag:sgtin-96:2:000173.0069119.13777360</epc>
       <epc>urn:epc:tag:sgtin-96:2:000173.0069120.13777361</epc>
      </childEPCs>
      <action>ADD</action>
      <bizStep>UNIT To CASE AGGREGATION</bizStep>
      <disposition>ACTIVE</disposition>
      <readPoint>
       <id>urn:epcglobal:fmcg:loc:0000000089103.RP-1527</id>
      </readPoint>
      <bizLocation>
       <id>urn:epcglobal:fmcg:loc:0000000089103.A23-49</id>
      </bizLocation>
      <extension>
       <batchNumber>10</batchNumber>
      </extension>
      </AggregationEvent>
    </EventList>
   </EPCISBody>
</epcis:EPCISDocument>
```

Appendix C

This Script creates dbspaces for tables, indexes, log files and temp files.

```
#########################################################################
#############
## Licensed Materials - Property of IBM
##
## Governed under the terms of the International
## License Agreement for Non-Warranted Sample Code.
##
## (C) COPYRIGHT International Business Machines Corp. 2006
## All Rights Reserved.
##
## US Government Users Restricted Rights - Use, duplication or
## disclosure restricted by GSA ADP Schedule Contract with IBM
Corp.
##
##   Title:IFXConfig.sh
##   Description:
##
##
#########################################################################
#############
#!/bin/sh
# determine where script was called from
script_location=`dirname ${0}`
# determine RFIDIC_HOME and source the rfidic-env.sh here
if  [ -n "$RFIDIC_HOME"  ]
then
        echo "env had RFIDIC_HOME $RFIDIC_HOME"
elif [ -e $script_location"/../etc/RFIDICServer.xml" ]
then
        RFIDIC_HOME=$script_location"/.."
else
        RFIDIC_HOME=/opt/ibm/WebSphere/RFIDIC
fi

if [ -d $RFIDIC_HOME ]
then
        source $RFIDIC_HOME/bin/rfidic-env.sh
else
        echo "RFIDIC_HOME not found $RFIDIC_HOME"
        exit -1
fi

INFORMIXDIR=/opt/IBM/informix
INFORMIXSERVER=epcisperf2
INFORMIXSQLHOSTS=/opt/IBM/informix/etc/sqlhosts.epcisperf2
ONCONFIG=onconfig.epcisperf2
INFORMIXSHMBASE=-1
PATH=$INFORMIXDIR/bin:$PATH

export DB_LOCALE=EN_US.utf8
```

```
export CLIENT_LOCALE=EN_US.utf8
export DBNLS=1

export INFORMIXDIR INFORMIXSERVER INFORMIXSQLHOSTS ONCONFIG
INFORMIXSHMBASE PATH

# Create dbspaces for tables

touch /u04/Perf-DBs/505m_DB/dbspace5
chmod 660 /u04/Perf-DBs/505m_DB/dbspace5
onspaces -c -d dbspace5 -p /u04/Perf-DBs/505m_DB/dbspace5 -o 0 -
s 50000000K
for i in 0 1 2 3 4 5 7 8
for i in 6 7
do
if [ ! -f /u04/Perf-DBs/505m_DB/dbspace5.$i ]
then
       touch /u04/Perf-DBs/505m_DB/dbspace5.$i
       chmod 660 /u04/Perf-DBs/505m_DB/dbspace5.$i
       onspaces -c -d dbspace5$i -p /u04/Perf-
DBs/505m_DB/dbspace5.$i -o 0 -s 50000000K &
fi
done

# Log space for Physical log
touch /u04/Perf-DBs/505m_DB/physlogspace1
chmod 660 /u04/Perf-DBs/505m_DB/physlogspace1
onspaces -c -d physlogspace$i -p /u04/Perf-
DBs/505m_DB/physlogspace$i -o 0 -s 25000000K &


# logspaces are for virtual logs
for i in 1 2 3
do
if [ ! -f /u04/Perf-DBs/505m_DB/virtlogspace$i ]
then

       touch /u04/Perf-DBs/505m_DB/virtlogspace$i
       chmod 660 /u04/Perf-DBs/505m_DB/virtlogspace$i
       onspaces -c -d virtlogspace$i -p /u04/Perf-
DBs/505m_DB/virtlogspace$i -o 0 -s 20000000K &
fi
done


# commands to create virtual log files

for i in 1 2 3 4 5 6 7 8 9 10
do
      onparams -a -s 200000 -d virtlogspace1
done

for i in 11 12 13 14 15 16 17 18 19 20
do
      onparams -a -s 200000 -d virtlogspace2
done
```

```
for i in 21 22 23 24 25 26 27 28 29 30
do
      onparams -a -s 200000 -d virtlogspace3
done

# Creating temp space

for i in 1 2 3 4 5 6 7 8
do
if [ ! -f /u04/Perf-DBs/505m_DB/tmpspace$i ]
then
       touch /u02/Perf-DBs/505m_DB/tmpspace$i
       chmod 660 /u02/Perf-DBs/505m_DB/tmpspace$i
       onspaces -c -d tmpspace$i -t -p /u02/Perf-
DBs/505m_DB/tmpspace$i -o 0 -s 5000000K &
fi
done

# Creating Index spaces

for i in 0 1 2 3 4 5 6 7
do
if [ ! -f /u04/Perf-DBs/505m_DB/indexspace5.$i ]
then
       touch /u04/Perf-DBs/505m_DB/indexspace5.$i
       chmod 660 /u04/Perf-DBs/505m_DB/indexspace5.$i
       onspaces -c -d indexspace5$i -p /u04/Perf-
DBs/505m_DB/indexspace5.$i -o 0 -s 50000000K &
fi
done
```

Appendix D

Script to create an index fragmented by expression (eventtime field).

```
SET ISOLATION TO DIRTY READ;
SET PDQPRIORITY 100;

drop index evt_time_idx2;

create INDEX "informix".evt_time_idx2 on
eventschema.event(eventtime, event_id, bizloc) using btree
FRAGMENT BY EXPRESSION
      eventtime between '2007-01-01 00:00:00.000' and '2007-01-
31 23:59:59.999' IN eventindspace1,
      eventtime between '2007-02-01 00:00:00.000' and '2007-02-
28 23:59:59.999' IN eventindspace2,
      eventtime between '2007-03-01 00:00:00.000' and '2007-03-
31 23:59:59.999' IN eventindspace3,
      eventtime between '2007-04-01 00:00:00.000' and '2007-04-
30 23:59:59.999' IN eventindspace4,
```

```
      eventtime between '2007-05-01 00:00:00.000' and '2007-05-
31 23:59:59.999' IN eventindspace5,
      eventtime between '2007-06-01 00:00:00.000' and '2007-06-
30 23:59:59.999' IN eventindspace6,
      eventtime between '2007-07-01 00:00:00.000' and '2007-07-
31 23:59:59.999' IN eventindspace7,
      eventtime >= '2007-08-01 00:00:00.000' IN eventindspace8,
      eventtime between '2006-09-01 00:00:00.000' and '2006-09-
30 23:59:59.999' IN eventindspace9,
      eventtime between '2006-10-01 00:00:00.000' and '2006-10-
31 23:59:59.999' IN eventindspace10,
      eventtime between '2006-11-01 00:00:00.000' and '2006-11-
30 23:59:59.999' IN eventindspace11,
      eventtime between '2005-04-01 00:00:00.000' and '2006-08-
31 23:59:59.999' IN eventindspace14,
      eventtime between '2006-12-01 00:00:00.000' and '2006-12-
31 23:59:59.999' IN eventindspace12,
      eventtime <= '2005-03-31 23:59:59.999' IN eventindspace13;
--update statistics HIGH FOR TABLE eventschema.event(eventtime,
event_id);
UPDATE STATISTICS LOW FOR TABLE 'EVENTSCHEMA'.event;
UPDATE STATISTICS MEDIUM FOR TABLE eventschema.event(bizloc);
--update statistics HIGH FOR TABLE eventschema.childepc(epc,
event_id);
commit;
```

Glossary

RFIDIC	Radio Frequency Identifier Information Center
EPC	Electronic Product Code
EPCIS	Electronic Product Code Information Services
OLTP	OnLine Transaction Processing: Write intensive applications
DSS	Decision Support Systems: Read only queries like Reporting queries
IDS	Informix Dynamic Server
TDI	Tivoli Data Integrator
WAS	Websphere Application Server
WMQ	Websphere Message Queues

References

For more information

[1] IBM WebSphere RFID Information Center Overview

[2] Informix Dynamic Server v10.0 Information Center (http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp)

[3] Informix Dynamic Server Performance Guide (http://www.informix.com.ua/doc/9.40/ct1t9na.pdf)

[4] Tivoli Data Integrator (http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp)

[5] ToxGene (http://www.cs.toronto.edu/tox/toxgene/)

IBM United States of America

Produced in the United States of America

IBM Director of Licensing
IBM Corporation
4205 South Miami Boulevard
Research Triangle Park, NC 27709 U.S.A.