1999

Application of mainstream object relational database to real time database applications in industrial automation

Saugato Mukerji
University of Wollongong

Recommended Citation

Mukerji, Saugato, Application of mainstream object relational database to real time database applications in industrial automation, Master of Science (Hons.) thesis, Department of Computer Science, University of Wollongong, 1999. https://ro.uow.edu.au/theses/2794

APPLICATION OF MAINSTREAM OBJECT RELATIONAL DATABASE TO REAL TIME DATABASE APPLICATIONS IN INDUSTRIAL AUTOMATION

A Thesis submitted in fulfillment of the requirements for the award of the degree

M.Sc. (Hons.)

from

UNIVERSITY OF WOLLONGONG

By

SAUGATO MUKERJI, B Tech EE, MBA

Department of Computer Science

1999.

Acknowledgements

I thank my supervisor Dr. Janusz Getta for his guidance, support in completion of my thesis. I am particularly grateful for the encouragement to pursue independent ideas.

I also acknowledge the support I received from Howard Chapman, Husseyin Erkoru, John Conran and other colleagues and finally my employers BHP Steel for supporting me as an approved student.

Contents

Section 1.0 Introduction

Abstract

1.1 Overview 1

1.1.1 The Problem 5

1.2 Review of Previous work 7

1.3 Directions – Emerging Trends 10

1.4 Case Studies 13

1.5 How the rest of this document is organized 16

Section 2.0 Real-time vs Mainstream Object Relational Database

2.1 Real-time databases 19

2.2 Relational databases 23

2.3 Comparison of Relational and Real-time Database Features 26

Section 3.0 The Solution

3.1 Implementation Real Time Database Features – Problem1

3.1.1 Implementation issues – significance of java in the database 28

3.1.2 Automatic Replication for online querying 31

3.1.3 Remote Access and administration tools 33

3.1.4. Change of State Detection by Triggers 35

3.1.5 Alarm Management Subsystem 39

3.1.6 Statistical functions, averaging, integration, moving average.. 48

3.2 Implementing Compression – Problem2

3.2 Need for online real time synchronous compression. 49

3.2.1 High throughput Data storage estimate 51

3.2.1.1 Strip Mill Data Compression results 53

3.2.2 Synchronous Compression technique applied to high volume .. 56

3.2.3 algorithm design for Adaptive Tuning 64

3.2.4 Using tables with many columns to improve data rate 65

3.2.5 Effect of sparseness on amount of space used per row 66

3.2.6 Effect of sparseness on Tables with Varray Columns 70

3.2.7 Partitioning a high throughput database for administrative ease 76

3.3 Implementing Software Engineering approach - Problem 3

3.3.1 Database configuration tools and API available in a typical Real-Time database for Application development 78

3.3.2 Autogenerating Table definition from java class definition 79

3.3.3 Generating a java Wrapper Library to access Tables 81

3.4 Performance Issues – Problem4

3.4.1 IP21 Real Time Database benchmarks with multiple processes.. 83

3.4.2 Relational Database benchmarks 85

3.4.2 Update/Insert rate limitation in Relational Databases 91

3.4.3 Using Solid State Disk as shared memory 94

3.4.4 Shared memory using memory based file systems 96

3.4.5 A proposed design of a shared memory data server 99

Section 4.0 Conclusions

4.0 Conclusions 106

Section 5.0 References

5.0 References 109

Section 6.0 Appendix

6.1 Solid State disk performance figures

6.2 Source for java triggers that generate automatic updates to a GUI running in a browser when a table gets updated

6.3 Object relational table read/write benchmark code in PLSQL

6.4 Source code for Synchronous Compression prototype

6.5.1 Application code to test autogenerated accessor functions

6.5.2 Primary data class

6.5.3 Autogenerated database access helper

6.5.4 Autogenerated table creation script that maps the java class data members

6.6 Scripts used in section 3.2.6

6.7 Vendor benchmarks related to 3.4.4

6.8 Code for testing JDBC performance Ref 3.4.1

6.9 PLSQL Scripts and batch files for testing performance with multiple concurrent processes Ref 3.4.1

7.0 Glossary

7.0 Glossary

Abstract

This thesis examines the proposition that because of recent huge increases in processing power, disk and memory capacities the commercial mainstream object relational databases may now be a viable option to replace dedicated real-time databases in industrial automation. The benefits are lower product cost, greater availability of trained manpower for development and maintenance and lower risks due to larger installed base and larger number of platforms supported. The issues considered in testing this proposition were performance, ability to mimic critical real-time database features, replication of the real-time database application development and administration tools and finally the low overhead high speed, real-time data compression facility available in real-time databases. An efficient yet simple real-time compression algorithm was developed for use with relational databases and benchmarked. Extensive comparative benchmarking has been done to convincingly prove the proposition. The results overwhelmingly show, that for a majority of industrial real-time database applications, the performance offered by a commercial object relational database on a current platform are more than adequate.

1.0 Introduction

1.1 Overview

Until recently memory resident Real-time Databases had a specialized niche which matched a set of functional requirements necessary for deterministic real world automation systems. Commercial mainstream database products were usually too slow and failed to meet the deterministic performance levels, which are required by the real world processes.

Recent dramatic increases in processor performance, easy availability of large amounts of cheap RAM estimated at A$1500 per giga byte and the emergence of low cost fast sub 10ms disk drives of more than 10GB capacity together have created the reason for the topic of this research. This study looks at the possibility of using commercial mainstream database products in real world automation applications.

The main application of real-time databases in automation is described here. The Real-time database has often been used as a shared memory record structure, which keeps track of the different states of the manufacturing process for each item. At any time there may be several items in different stages (or states) of the manufacturing process. A number of cooperating applications work together to help transform the input raw material to the finished product while conforming to a series of pre-calculated intermediate stages. Before the manufacturing process begins for a input item, its input characteristics and planned output schedule are obtained from other enterprise level systems and stored into the real-time database. Then some other applications read this schedule data and use process dependent mathematical models and tuning parameters based on previously measured data to generate a detailed manufacturing setup for each stage. As the item proceeds through each stage the measured data is gathered and fed into the Real-time database. One or more application processes monitor the variation between the setup parameters and the returned measurements. The deviations are subjected to the same models to calculate the adjustments, which must be applied to the setup for the next stage(s) to ensure the finished product meets the specified tolerances and has the desired properties. On completion of the manufacturing, the tuning parameters are recomputed and the record structures used for the item just manufactured are snapshotted and stored, to allow future postmortem of aspects including product defects, model performance, change in process operating conditions. The critical aspect here is the fast access to the measurement, setup, schedule data in the real-time database for timely completion of the required setup generation for each stage of manufacturing. Typically the most stringent requirements of the time allowed for such data access and model computations may be sub second where critical manufacturing stages may be 3-6 seconds.

In the last decade an additional functional requirement has emerged for real-time database applications in industrial automation systems. It is now required that the actual parameters measured during the manufacturing process be logged to sub 100 millisecond resolutions and be easily retrievable. This requirement is driven by the need to be able to provide proof of manufactured product quality as a statutory legal requirement for 7 years. Another important reason for such logging of process data is the need to attain higher product quality and tolerance to meet the quality standards specified by the customer and to reduce wastage and improve the process yields by manufacturing to tighter tolerances. To improve product quality it is very important to be able to replay incidents where lapses in quality occurred. This allows the process engineers to analyze and rectify the underlying problems. To respond to these needs, real-time databases now implement mechanisms for compressing and storing such large volumes of process data in proprietary formats on disk. However such proprietary data storage is obviously not attractive since it is less accessible due to the non-open nature.

Another equally important issue associated with storing 7 years of sub 100 millisecond resolution data is the sheer volume of the data. Uncompressed, this quickly exceeds even the huge 100Gb to 200Gb capacities of commercially available RAID arrays. Assembling and administering larger batteries of RAID or other storage solutions is a very expensive and commercially unviable proposition. The challenge therefore is to devise compression algorithms that compress the data by a factor of 10 or more while preserving the intrinsic accuracy of the data. Such compression also must not compromise the easy online and open access to data by tools like SQL without any intermediate manipulation or massaging

At this time several projects have used a mainstream relational and a real-time database together, to provide the fast performance of the real-time database coupled with the scalable and open data storage offered by the mainstream relational database. While having two databases does address the problem, it is not without a price. There is additional software life cycle cost associated with licensing and administering two separate database products. All application software has to be able to interface with two different databases. Further maintenance problems arise when software changes have to be synchronized in the relational database, in the real-time database and in the application software. This synchronization process is potentially error prone and can lead to expensive downtime in a 7x24 scenario and is also a strong disincentive to make process improvements requiring software changes.

The emerging object relational features now being incorporated into mainstream relational databases too are of great relevance to developers of automation applications. The object relational features allow a natural mapping of the application objects being manipulated in the program code to customized object data types and object relational tables. This makes it easier to log and retrieve the application objects to and from the database without any error prone hard coded mapping. Real-time databases have always provided developer configurable record structures for such a natural mapping for application objects and also come with a library to easily access the record structures.

We have also disproved the common misconceptions that object relational table access is less efficient conventional table access. We found the results were similar for both. The likely explanation is the fact that the object relational features are only a thin wrapper on the relational DBMS , hence it does not degrade performance. The time taken to execute the wrapper code is negligible as compared to the time taken for the disk i/o based journalizing of transactions in the relational DBMS (this is central to the operation of a relational DBMS and can not be switched off).

1.1.1 The Problem

Global Objective is:

To replace real-time database with a commercial relational DBMS.
The sub problems are

1. Performace and throughput issues.
2. Compression of online data.
3. Ease of development.
4. Replication of important real-time database constructs and features.

1. To benchmark the performance of real-time databases and real-time databases when performing read/write operations (on a single field and full record) from a single application and from multiple applications concurrent applications. The intention is to measure and quantify the limitations in using mainstream relational database. This information should provide a means to select automation applications where the requirements can be adequately met by a mainstream relational database. The benchmarking also leads to recommendations on how to overcome limitations observed in using relational databases in real-time database applications.

2. To design a online low overhead compression algorithm that can be used when loading hundreds of columns of real time process data, at sub 100 millisecond intervals, into a mainstream relational database. The ability to do this compression may allow a relational database to completely replace a real-time database or replace a combination of a real-time and a relational database. Equally important side benefit of such compression is the ability to make the data small enough to fit into a commercially available and relatively inexpensive RAID array. The point here is the fact that the direct and indirect costs associated with the disk storage system rises sharply once the capacity exceeds the commonly used RAID arrays (currently 100 to 200Gb).

3. To analyze the important features of an established real-time database and to attempt to realize the same functionality using the common functions and features of a leading mainstream relational database. These features make it easy to implement real time automation solutions without writing a lot of application code.
4. To investigate the software engineering issues involved in seamlessly auto-generating the object relational database datatypes, tables and accessor classes using scripts. We developed such a script using the popular Perl scripting language. The autogeneration script parses the class stubs generated by a modern case tool and generates the object relational database datatypes, tables and accessor classes. With this approach the software changes are much easier since the engineer is only required to make the change in the case tool and then run the script. The error prone synchronization is eliminated.

1.2 Review of Existing Work

While a large amount of research has occurred in real-time databases it has been mainly aimed at devising algorithms and approaches to enhance the performance of real-time databases operating under resource and time constraints.

Nobody seems to have considered using a mainstream object relational database in real-time applications and how with the great increases in processing power they may now be a viable solution in certain applications. So in our study we may have the privilege to tread down a relatively un-trodden trail when we attempt to measure the performance of mainstream object relational database when applied to a real-time application environment.

We did not find any published work on using mainstream object relational databases in real-time application environment in industrial automation.

In reviewing publications on real-time databases, the following issues were found to be related to this study.

1. Timeliness of transactions in real-time databases.
2. Transactions with temporal data constraints in real-time databases.
3. Implementation of a real-time database system by retrofitting a commercial database.

1. Significant work has occurred in analyzing the importance of timeliness of transactions in real-time databases. In many applications timeliness is most important and other aspects of a transaction like completeness, accuracy, consistency of results may have to be traded off to achieve timely processing. The paper "Real-time Databases" by K Ramamritham which appeared in Distributed and Parallel databases, Vol 1, 1993, [1] examines the issues involved in such a trade off. Interestingly while the paper is now 7 years old and the processing power of cpu(s) and resources like speed of disks, memory capacity have all increased hugely, the issues discussed in the paper remain relevant. This is due to the fact that as the processing power has

increased so has the expectation of users, who now expect to run a more detailed online model, sample and process data 10 times faster or more and expect to store the hugely increased volume of data in the same time interval. Therefore though not identical constraints very similar to those considered in the paper are alive and well today.

2. Some real-time database applications also have data with time constraints. The paper "Scheduling Access to Temporal Data in Real-time Databases" by Ming Xiong et al [6] shows how a forced wait policy can be implemented to delay a transaction that is attempting to use data that has expired or is about to expire, by forcing the transaction to wait, till it can be assured that valid fresh data will be used. Examples of such transactions where using expired data is not acceptable are auto pilot systems and programmed stock trading. It is easy to appreciate this if we consider a programmed trading system which is calculating the volume and type of the buy or sell decisions after sizable calculation based on the trend produced by share price updates. Assume it is about to commit the order and the data deadline expires. It is right to hold the order back because the share price trend could have reversed. However if the new trend value is similar then instead of wasting time in recalculation of the order, the halted transaction should be revived to save time and increase chance of making a profitable trade.

3. We found one example of an experiment to retrofit a commercial database by altering its internals to make it work as a real-time database. The paper "Implementation of a Real-time Database System" by Aranha, Narayanan,Muthukrishnan,Ganti,Prasad and Ramamritham [17]which appeared in Information Systems 1997 describes an experimental work in which a commercial database system Genesis was adapted to create its real-time version RT-Genesis to support the firm deadline transaction requirements. The adaptation allowed priority based program execution where priorities determined deadlines of transactions under the sole control of the database system. Conflict resolution over data and other

resources were done on the basis of the transaction deadlines. The paper considered a database populated by a mixture of tables, some of which had only 10 rows whereas others had 100 rows. Transactions with randomly created different priorities were applied to test the performance. Transactions, which failed to meet the timing constraint, were aborted since they were deemed to be useless. The paper described comparative performance achieved by the scheduling, buffer management and lock management algorithms used in RT-Genesis. While the algorithms performed as expected the results indicate above a certain rate of arrival aborts started occurring and the performance was well short of a typical real-time database. There seems to be an assumption that it is essential to be able to perform transactions with a 2 phase commit and frequent occurrence of non trivial SQL queries. In our experience in automation applications, much of the need for conflict resolution can be designed out by recognizing the following common aspects. The process is sequential in nature, that only one object can occupy a physical process zone at a time and that the tables can be designed in such a way that the minimum number of applications share a table. The typical usage of real-time databases in automation is as a shared memory. The transactions are often a single field update, which are performed more efficiently using an API call and execute without significant delay. SQL procedures are used only for non-repetitive one off functions where the logic is complex and coding the implementation using the real-time database API will be laborious. By designing with the factors outlined above it should be possible to sidestep resource and data conflicts when implementing applications in automation using relational databases. We agree though that SQL and ability to roll back unsuccessful transactions are important in real time commercial applications like e-commerce.

1.3 Directions – Emerging Trends

Looking at the current emerging products aimed at the real-time end of the database market we observed the following common trend. Several new memory resident relational databases have appeared. These products which mimic to a large extent the features of a mainstream relational database. These new products claim to be 10 to 20 times faster because of the fact that the code is designed to operate the relational database entirely in the main memory. These products offer the familiar SQL and C/C++/Java API's as development tools aiming to attract developers experienced in the relational databases to develop high performance web based real-time applications. TimesTen, PolyHedra and Angara are examples of such databases.

We will look briefly at the features supported by these new memory resident relational databases

Angara Database Systems

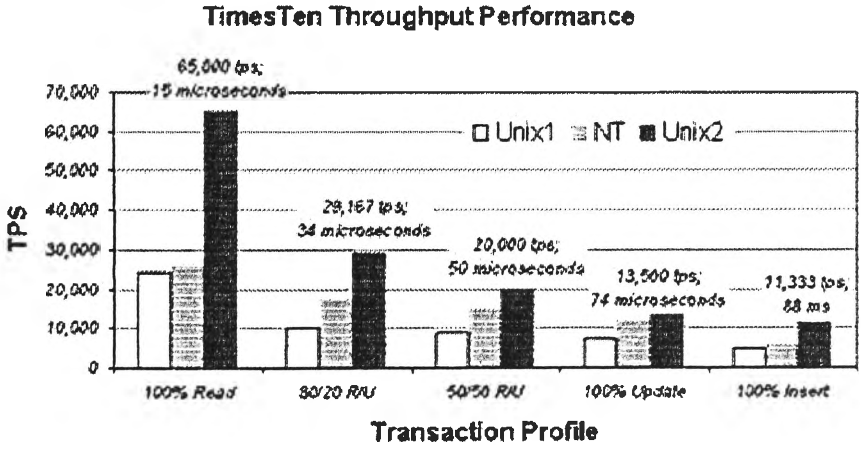
Angara Database Systems, Inc. is a developer of ultra high-performance memory-resident relational databases. The Angara main memory database (MMDB) is a software product with a relational database engine explicitly designed to handle in-memory data. This engine, run in conjunction with a standard disk-based database management system (DBMS), and accessed through a C application programming interface (C API) or a standard SQL interface, handles small read-intensive or temporary tables over ten times faster than the cache of any standard disk-based DBMS. Using the Angara MMDB, while simultaneously delivering the ability to handle extremely large data sets, sophisticated backup and recovery functionality, extensive administrative tools and visual programming tools of the standard disk-based DBMS.

TimesTen

TimesTen is a relational database optimized for in-memory performance. TimesTen databases reside entirely in memory at runtime--a principle that allows for highly efficient instruction paths for performing relational operations, as compared to fully cached data in a disk-optimized RDBMS. The flip side to performance is the size constraint of memory, which presents a practical limit to the kinds of applications that will derive the most benefit. In-memory databases should contain highly active, performance-critical data, such as subscriber profiles, open orders, service records, translation tables, etc. Ideal applications are those that demand the fastest response times possible, and those that ask the database to perform extraordinary amounts of processing within an ordinary response interval.

TimesTen is focused on enabling application innovations at the forefront of Internet computing, and in the infrastructures of today's voice and data networks. With the performance opportunities provided by TimesTen, application designers now have the ability to include greater amounts of data and sophisticated on-the-fly database processing. And because of TimesTen's efficient design, a workload that used to require much larger systems can often be handled by PC-class server hardware running TimesTen.

Presented below is a performance benchmark achieved by a recently launched non mainstream memory resident database



The reason for presenting this benchmark is to show the performance achievable by non mainstream recent memory resident database(MRDBs) products. Low cost systems delivered 25000 reads and 5000 writes per second when used in a read only or write only operation. These access rates are already good enough for many real-time applications and are helping the new MRDB's find ready application as low cost high performance web server backends. Typically automation real-time applications are developed using mainstream products with a significant installed base. The reason is the typical 10 year+ post commissioning life cycles of automation projects. The project managers are unlikely to adopt non mainstream MRDB products (even though they offer better benchmarks) due to concerns about the future availability of necessary support and upgrades.

The results shown above are from three different platforms: a small, rack-mountable 2-CPU UNIX system favored in voice and data network solutions; a 4-CPU Pentium II Windows NT server; and a new mid-range UNIX server with a state-of-the-art memory subsystem, using 2-CPUs for this test. In the performance chart, we will refer to these systems as UNIX1, NT, and UNIX2, respectively.
To anyone familiar with traditional database system performance, these results speak for themselves. The UNIX2 machine, with its highly optimized memory subsystem, generated an impressive 65,000 read transactions per second, using only 2 CPUs. In application terms, an internet commerce system could compare a purchase intention with over 100,000 previous orders, and recommend additional items that were most often purchased, all in less than 2 seconds. When measured by response times, all of the workloads represented in this benchmark completed in microseconds (millionths of a second), meaning the time to look up a phone number or network address is practically immaterial.

1.4 Case presentation of two different real-time database applications in steel rolling

Case 1 Plate rolling mill control system

Outlined here is a recently completed successful industrial automation project. The design and system architecture aspects have been highlighted.

The mill control computer (MCC) system was implemented using an object oriented design and implemented in C++ at the server and Java at the GUI front end.

The server was implemented as a set of co-operating executable applications that referred to a common shared memory visible to all the applications. A leading memory resident real-time database was used to provide the shared memory. The shared memory was accessed using a user written access library. The access library in turn was coded using the low level C language API library provided with the real-time database. The access library allowed the application programmers coding in C++ to access the real-time memory a without having to learn the low-level API.

Each class in the applications was auto generated into a real-time database record using PERL and SQL scripts. The Access API provided the means to read or write at the granularity of a single field, an array field or the entire record. Writing or reading an entire record allowed complete mapping between the C++ class and shared memory record.

In this application the real-time database was used to store the process state, the setup parameters for each processing step and the measurements obtained from the process instruments at each stage. The above information was created or recorded for the item being manufactured from before the start of processing to the exit of the finished product.

In this case typically there could only be about 10 individual slabs aka pieces which were being processed. This was dictated by the fact that only one item could occupy a process state. This followed logically from the fact that each state represented an operation by a physical piece of equipment located at a specific position in the plate rolling line.

We implemented 15 instances of each such record type in the database. Each such instance record had a common part plus a postfix that consisted of a number between 1 and 15. Such record types were called piece dependent since they contained rolling setups, measured data and schedule information related to a specific piece. The database consisted of over 200 record types of which about 150 were piece dependent. The entire set of 150 records associated with a specific piece was collectively called a piece file.

Each individual item was allocated a free piece file prior to the start. During the rolling process the piece file is used to store all the data related to the item. At the end of the rolling process the piece file is copied to a separate historical data storage system and the piece file is released for use by new items.

In this application there is no attempt to record the historical data in the real-time database consequently its footprint in memory as well as its disk usage remains constant.

Case 2 High speed data logger system for a strip rolling mill

A strip rolling operation is a high-speed operation where the strip velocity can be as high as 20 meters/second. As a statutory requirement the manufacturer is obliged to maintain and record the in process measured parameters for a number of designated quality variables. It is desired to record this information at the granularity of every meter, this means the data rate could be as high as 20 readings/ second.

At the end of the rolling process which is around 200 seconds all the observations are retrieved and a number of statistical computations performed on the data to establish the quality status of the finished product. A summary of the statistical quality results is forwarded to the customer and the detailed data is available on demand for a number of years.

Additionally the maintenance engineers use this per meter data to perform post trip analysis. The data must be time stamped immediately on capture and should be time-stamped to a resolution of 1 ms or better. The accuracy of the timestamp is important to determine the variable sequence, which caused the trip or incident. Typical post trip queries select a number of related parameters over the trip duration. If this information is located in archived history files it has to be retrieved before the query can be applied.

Since this data is arriving at a very high rate 4000 reading/ second(200 inputs scanned every 50 ms) and each value must be recorded with timestamp. It was observed that it took 15 bytes to store one timestamp and the associated reading. The data stream needs to be compressed by rejecting the values that are not significantly different for the previously recorded observations. Such compression techniques are built in the real-time database history subsystem.

1.5 How the rest of this document is organized

This sub section describes how the rest of this document is organized. The reader may use the roadmap presented here to navigate the document.

Section 2

The following section 2 presents the typical features of real time and mainstream object relational databases and compares the features. The section intends to make the reader familiar with the typical features of the two database types to allow a better understanding of the implementation details and design decisions proposed in the following section.

Section 3

The following Section 3 titled "The Solution" deals with the outlines the feature by implementation of real time database using a mainstream object relational database.

3.1

The first subsection 3.1 starts by explaining the significance of the recent introduction of embedded Java in the object relational database. Section 3.1.3 describes the implementation of the shared memory access API . Section 3.1.4 outlines the implementation of the powerful change of state - event detection and action triggering construct central to real time databases. This implementation is done as a combination of triggers and Java stored procedures. The section 3.1.5 ends with the implementation of the alarm subsystem using a combination of a master alarms table with views for selecting specific subsets. Again Java stored procedures, JDBC, triggers are utilised as the means of turning alarms on and off and receiving operator acknowledgement. Section 3.1.6 describes the remote access tools.

3.2

This subsection starts by explaining the role of the online real time compression feature of real time databases. Next the implementation of the fast online synchronous real time compression algorithm developed by us is explained and possible approaches for further improvement are pointed out. Subsections 3.2.4 through to 3.2.6 explain how the storing the compressed data using a few tables with a large number of columns is efficient and that the compressed data can be efficiently stored by saving it as sparse data into relational tables. The rejected data are represented by null. Storing null instead of data gives a compression of 1:10. The actual results achieved using compression are presented next. The concept of partitioning the stored compressed data is explored. The ability to make read only the partitions containing the historical data is explored to simplify the backups.

3.3

This subsection is devoted mainly to software engineering issues and starts by explaining the database configuration tools and access API offered by real time databases. The section outlines the need to provide similar ease of development when developing a real time application using only a mainstream object relational database. The section 3.3.2 describes our implementation of the technique for directly generating the database schema from the data members in the Java source files describing the application classes. The Section 3.3.3 describes the script to directly generate the Java wrapper library to access the table rows mapping the class instances on the application code. Again this auto-generation of the wrapper code is done by directly processing the Java source file with a script.

3.4

This section contains performance benchmarks for real-time and relational databases explores other techniques to implement fast-shared memory between co-operating applications that together offer a real time industrial automation solution. Solid state disks and software driven file systems in memory are two promising approaches explored.

Section 4

Our conclusions and recommendations are presented in this section. We try to identify applications in industrial automation where mainstream object relational databases are already a viable alternative to dedicated real-time databases. We also try to point out applications, where at this time there is no alternative to dedicated real-time databases.

Section 5

Contains a list of relevant references

Section 6

This section is the appendix and contains code listings,
scripts, programs runs and screen captures

Section 7

This section is a glossary

2.0 Real-Time vs Mainstream Object Relational

2.1 Typical features of real-time databases

This section describes in more detail typical features of real-time databases. **The purpose of this section is to explain the features which we will mimic in a subsequent section using an object relational database.** These features have evolved over time to serve the needs of industrial automation systems. For the purpose of our study we will describe in detail some the features of the IP21 real-time database.

2.1.1 Shared memory access using the API library

All the application programs using the real-time database open a connection to it by executing a API call. Applications can then freely access the records of the database using the API calls described below. Each API call executes within 10s of microseconds. The database is locked while any write call executes. All the API access to the database are performed in chronological order of receipt. While this seems to be unsophisticated it assures deterministic behavior. Application software writers use the appropriate API from the library to make the database appear as a shared memory between different applications. A popular real-time database was clocked at 100000 separate read/write API calls per second. Higher rates of data transfer are achieved by the array read write API calls.

API calls are grouped in the following functional groups.

- Read/write single data base functions for data types including
  Shorts
  Longs
  Char buffer
  Timestamp
  Real
  Double

- Read/write array fields of the datatypes supported for single read/write.

- Read write single fields of any type with ASCII data

- Read/Write multiple fields of the same record in a single call

- Read / write multiple fields in different records in a single call

2.1.2 Change of state(COS detection)

The IP21 real-time database has a template record called COSACTDef to create COSACT application records for monitoring COS events and generating activations on detecting the COS event. The COSAct records contain a reference to the field to be monitored and have a field containing the type of COS change to be recognized. The COS event types are 'all' , 'change to default', 'non default' and 'none' . The COSAct record also contains a list of records to activate on detecting the specified COS event. When the change of state occurs in a monitored field, the COSAct record activates the record(s) in its list. Applications execute an API function that blocks until it detects an activation. The call returns the reference to the activated record. The application then decides the functions to execute based on the source of the activation. Thus an update of a field in the database by an application can make a different application execute a specified function. This is a powerful loosely coupled run time reconfigurable low overhead event driven switching facility

2.1.3 Data base structure

Real-time databases have for a long time allowed great flexibility in defining the records in the database. It is allowed to have a record containing a mix of fixed area fields and multiple repeat area blocks. Each repeat area may contain one or more fields. This flexibility allows a far superior and simple modeling of the real world objects than what can be achieved by a series of related relational tables. In IP21 the DefinitionDef record is the master template record which was used to define the other template (aka definition) records. Any number of instance or application records as required by the application can be created using the template records.

2.1.4 Highly configurable alarm handling and event logging

Real-time databases typically have an in-built alarm handling subsystem which automatically inserts and removes alarms from pre configured alarm summaries based on occurrence of alarm events. Alarm summaries contain specified grouping of alarms by specifying the selection criteria. Alarm events could be any one of values moving past preset limits, the alarm state being set by program code for derived alarms, users acknowledging alarms and alarms timing out.

2.1.5 Compression and history storage

Real-time databases usually provide compression and historical storage for time series data. However this is usually an auxiliary feature and in no way impacts on the real-time database if there is no space on the disk to store more history. The default operation is to do a round robin and overwrite old history. The whole approach is to ensure availability and uptime whereas in relational databases the ensuring integrity of transactions is the highest priority even if it means sacrificing availability.

2.1.6 Inbuilt configurable PLC and SCADA protocol support

Real-time databases usually have a suite of layered products which allow easy communication with typical automation data sources such as PLC's DCS's and others SCADA systems. Integrating these communication links typically involves no compilation of protocol handler code. Mere configuration of special database records related to the layered products is sufficient.

2.1.7 Statistical and quality control functions

Real-time databases have a add on layered product to do averaging, integration weighted average, control charts. These features can be configured without compiling any code.

2.1.8 Scheduling or Timing records

IP21 real-time database has a template record called ScheduledActDef which allows one or more records to be activated when a predefined period expires after a specified starting time. Additionally the IP21 supports SQL stored procedures, which have a scheduling, feature, which allows the stored procedure to be auto, executed periodically.

2.1.9 ANSI SQL support

Most real-time databases support a variant of the ANSI SQL 92 or later. IP21 comes with both command line and GUI based SQL engine. The SQL can be invoked via code command line, by triggering a stored query using COS detection on a monitored database field or by a scheduling event defined in the database.

2.1.10 Stored SQL query records

SQL queries can be tested on the GUI SQL tool called SQL plus and can then be saved into a file. The query output can be directed to other database fields. Stored SQL queries are a powerful feature which are often preferable to coded programs when implementing application features.

2.2 Typical relational database features

Relational databases on the other hand conventionally have a different suite of features which have evolved according to the needs of the applications which used them. Not all of these features are of great help in a real time situation.

2.2.1 Online storage of transaction logs

Some central features specific to relational databases are the ability to maintain on disk a log of transactions(aka redo logs) and resulting ability to recover transactions upto the point of failure by restoring a previous backup re-applying the journalized transactions. There are further specialized features, which allow the archiving of these logged files before they are over written. This archiving of the transaction log ensures the ability to recover completely even when the online logs have been overwritten by subsequent transactions.

2.2.2 Tables grow and can auto extend if configured to do so

The typical usage of relational databases involves inserting more rows to store new data with persistence. Hence the database grows with time as more and more data is stored. Persistence by definition means the database can never be entirely memory resident and therefore needs to read and write the transactions to disk within seconds of its occurrence. The disk space has to be monitored carefully by an administrator since the database is designed to halt if data file space or disk space runs out. Relational databases can be configured to auto extend data files to acquire more disk space when required. However they will normally stop further operation if the disk fills up. In a real time situation such an occurrence will bring production to a halt.

2.2.3 Operation is not deterministic

The typical access of a relational databases is performed using SQL to perform select, update, insert or delete operations (called queries) on rows of one or more relational tables. Depending on the associated criteria and other factors like the size of the table, use of indexes and the number of tables involved the access time could vary significantly. In any write operations the associated rows and columns are locked to prevent other similar queries from interfering and compromising the results of a query that is in progress.

2.2.4 Failed or aborted transactions are automatically rolled back

Relational databases support rollback which means they store the before value to the rollback segment on disk before making the change. This information is guaranteed for the life of the transaction. If the user issues a rollback command or exits the session before completing the transaction with a commit the transaction is rolled back by restoring the changed data from the rollback segment. The rollback feature is very useful because it allows any transaction that failed to complete for any reason to be rolled back automatically to the state before starting the transaction. The downside is that a large amount of rollback segment is used for large sized transactions as often occur in large batch jobs. The ability to rollback transactions that have not been committed also causes complexity because the associated rows and columns are locked until the transaction is committed. This in leads to a second transaction failing if it attempts to update insert or delete information into the rows and columns locked by the uncommitted query. In case of selects the uncommitted transaction will not be reflected. So the accessing query must have the failure handling code built in to take care of such cases.

2.2.5 Performance cost of the rollback and redo features

These roll back and redo features of relational databases which provide the guaranteed integrity of transactions come at a cost, the numbers of separate transactions that can be processed per second is much lower. This can however be compensated to some extent by reading or writing many data values in the same transaction. We will explore this further.

Relational databases do not guarantee maintaining the chronological order when a number of operations are performed from multiple processes simultaneously.

2.2.6 Stored Procedure and Triggers

Relational databases allow SQL code to be stored in the database as stored procedures. The stored procedures execute faster as they have been preprocessed at creation. Now stored procedures can be written in Java in most current object relational databases.

Relational databases support a mechanism called triggers which allow SQL commands or a stored procedure to be invoked when the trigger condition occurs. Typically the trigger condition is an update insert or delete on a table at a row level or table level. The trigger action can be specified to occur before or after the triggering transaction is executed. Triggers have many uses. Generating audit logs is a important application which uses triggers.

2.2.7 Automatic replication and snapshots

Most current relational databases support automatic replication. This allows changes on tables in master database instances to be propagated automatically to the replication target databases instances. This is very useful since local users can query a local database and get fast response to queries without impinging on the master database. The data is however maintained in synch with the master database. The replication can be done at the granularity of a table. So only a sub set of the master data base tables can be selected for replication. The replication feature enables efficient operation of distributed databases and supports automatic store forward ability to be resilient to communication and other faults.

2.3 Comparison of relational and real-time database features

Conventionally relational and real-time databases offer a different set of features. These features have evolved based on the requirements of the typical applications of the relational and real-time databases. Here is a comparison of the typical features of both.

feature	Object Relational	Real-Time
Read/write call rate [1]	~500 calls /sec	~100000 calls /sec
COS(change of state detection)	Yes using triggers	Native support for COS detection
Auto launching query on COS detection	Yes	Yes
Switching on COS detection to cause remote method execution	Yes using java triggers	Yes
Alarm Handling Subsystem	No but can be implemented easily	Yes
Historical data storage	Yes	Yes
Automatic Timestamping of inputs to subsecond resolution	No	Yes
Drivers for common PLC's DCS's	No	No
Native averaging, integration of inputs, SQC functions	No	Yes
Replication master to standby database	Yes	Yes
SQL engine, stored queries	Yes	Yes
Logging of transactions to support recovery	Yes	No
Ability to rollback uncommitted transaction	Yes	No
CORBA support	Yes	No
Data Compression based on trends	No	Yes
A client GUI package for	No	Yes

[1] The Relational Database achieved 500 calls/sec on a 2 processor 400MHz pentium II using a JDBC prepared statement. Rates of 5000/ sec are achievable with PLSQL

dynamic displays of Alarms, field data and trends		
Tables grow and can auto-extend	No	Yes
Timing/Scheduling feature	No	Yes

3.1 Implementing Real-Time Database Features - Problem 1

3.1.1 Implementation of real-time features - significance of Java in the database

This section examines the issues involved in implementing the typical features of real - time databases discussed in the preceding section. We will demonstrate by example how almost all the important real-time database can be implemented using an object relational database. The availability of a full-featured programming language like Java as a native component of the object relational database has simplified the task of mimicking a real-time database.

3.1.1.1 Fast read and write to the database using access API.

As we will demonstrate later advent of open interfaces like JDBC have allowed easier access to the database from application programs. We will demonstrate how an API can be auto generated to allow the application programmer to access the Tables in the object relational database with the same simplicity as available using the real-time databases and their proprietary API. Infact an interesting side issue is portability of applications so developed across different platforms and operating systems. In case of the IP21 real-time database it was limited to NT4.0 and with some additional work to Aix, HPUX and VMS. However if the applications are developed with Java and a JDBC wrapper is used to access the database the applications are portable automatically. Portability is guaranteed by WORA the central paradigm of Java. Write once run anywhere. Almost all popular platforms now support Java and JDBC ie NT4.0, Unix dialects including Solaris, Aix, HP-UX, SCO, Linux, VMS, AS400, and many more. The mainstream object relational database tested by us is available in all the operating systems listed above. We will investigate and compare the level of performance achieved by a object relational database using a JDBC wrapper when providing a shared memory behavior. The throughput achieved will define the applications which may be implemented using a object relational database instead of a real-time database.

3.1.1.2 Automatic change of state detection

Typically the ability to configure automatic detection of changes of state to specified fields has been a design construct used in real-time database application in implementing efficient event driven applications that provide fast response without any ongoing overhead. Without this feature the application programmer is forced to poll and test for change of state and this leads to a continuous CPU overhead. Relational databases have a construct called trigger which can allow specified action to be taken using a SQL commands when the triggering condition is encountered. The triggering condition is a any of an Insert, update or a delete on a table. The trigger action can be at the granularity of a field or a row and can be initiated both before and after the triggering event. While triggers were powerful, previously they could only execute SQL statements or launch stored SQL procedures. While this was a powerful feature it still did not provide full blown programming language support. Using trigger initiated stored procedures it was quite hard to invoke methods in other applications. To do so an external C language function had to be written and linked and the communications functionality would have to be built into the external function. With Java now embedded in the database triggers can be written in Java. The full communication features of Java including sockets and IIOP are available. We will develop a Trigger which uses the java.net.Multicaster to multicast the COS data monitored by a trigger to GUI's running in a browser. The COS detection will be done by a trigger written in Java. In fact this is a common COS detection application in real-time databases ie value changes in the database are monitored and automatically transmitted to the GUI without any application code.

3.1.1.3 Alarm and event logging interface

A typical alarm subsystem in a real-time database allows the creation of a series of alarm records. Each alarm record has a set of configurable attributes. The alarm records may feature in one or more alarm summaries depending on the value of the AlarmState attribute being on and a series of other conditions associated with the alarm being satisfied. Thus an alarm may feature in multiple alarm summaries and the alarm record contains a reference to the alarm summaries it may be visible in. The alarm summaries attempt to group the alarms by criteria including function or by the plant location or by maintenance discipline. When an alarm goes from alarmState OFF to alarmState ON it becomes the most recent alarm in one or more summaries and shows up on the top of the summary. The alarm may be turned on by an application program writing a ON value into the alarmState field when the scenario related to the alarm occurs. This turning on of the alarm can also be done by value in IO input records crossing alarm limits. Alarm summaries are automatically displayed on GUI when the screen containing a reference to the summary is opened. We will use tables and views to create an alarm system in a object relational database which will behave like the alarm subsystem in a real-time database. In many ways the object relational design may seem more elegant and powerful because we are able to use constructs which are native to relational databases. Java will again allow easy addition of features and integration to GUI and the client application programs.

3.1.1.4 Compression of IO-input data and storage to disk history

Real-time databases offer a facility to timestamp data written to IO-input records and add the freshly timestamped data to the history along with the timestamp. A limited amount of the history is kept in memory. There is normally a facility to save the values that fall off the history array in memory to a disk history file system. This disk history facility can be turned on and off on a per point basis. There is a facility to apply compression to the input data and save only those that satisfy the compression parameters. There is also an API available to insert data directly into history. This is often required when remotely acquired and timestamped data has to be saved in disk history. We have attempted to reproduce this functionality using relational tables and JDBC with our own compression algorithm coded in Java.

3.1.2 Automatic replication for online querying

In a typical 7x24 hour automation application the deterministic performance requirements often make it impossible for multiple users to do online querying on the table data. The problem gets worse when the span of the queried data is several years and has to be extracted from a number of different partitions. Such data warehousing features are better handled in object relational databases as compared to conventional real-time databases.

In the IP21 real-time database used for comparison the historical data is streamed into predefined filesets. When a fileset fills up the datastream switches to the next fileset in round robin group. It is the job of the user application to archive the just filled up fileset to preserve the information. It is necessary to manually restore the older archived filesets before any data contained in them could be retrieved. Naturally this process becomes even more cumbersome when the time span of the query involved several filesets.

The processor loading resulting from executing queries is unpredictable and likely to impact on the deterministic performance requirements of the primary system. Therefore the best solution is to run a standby system which continuously replicates the primary system. All user queries should be run on the standby system.

In case of IP21 the loading the archived primary fileset into the secondary real-time database is manual.

The Oracle 8i object relational database has built in support for automatic replication. We can specify that the replication be done synchronously. Synchronous data propagation occurs, when an application updates a local table, and within the same transaction also updates all other replicas of the same table. Consequently, synchronous data replication is also called real-time data replication. Use synchronous replication only when applications require that replicated sites remain continuously synchronized.

Asynchronous replication is the other option. In this case procedural replication can offer performance advantages for large batch-oriented operations operating on large numbers of rows that can be run serially within a replicated environment.

Alternately we can use the update and select together to access the table information.The end result is to update a target table when the snapshot is periodically done.

All user querying is done on the standby object relational database. This way the primary is not impacted when the query(s) execute.

The snapshot can be triggered automatically to maintain the standby and master in synch.

3.1.3 Remote access and administration tools

Increasingly organizations are capitalizing on their investment in an enterprise-wide high speed LAN by using centrally located common system administration resources and engineering to control and administer database installations which are geographically widely dispersed. This feature proves extremely useful. It allows expert troubleshooters to login and solve the problem remotely. In some cases there could be the vendors support staff who are able to participate in the diagnosis and resolution..

Utilizing a software agent running on the local installation facilitates the remote access. The remote user runs a client application on a desktop computer somewhere on the enterprise wide LAN(inside the company firewall),

In case of the IP21 real-time database, the IP21 administrator is a client-end software that provides the remote access and control. The IP21 administrator can be installed on any NT4.0 based PC. The administrator attempts to discover all IP21 nodes in the subnet. using a broadcast protocol. Alternatively the user can prompt the IP21 administrator with the IP of the remote IP21 instance. The IP21 administrator then makes contact with an agent running on the local node and establishes communications. The client can then perform all database operations from the remote node. Another similar remote access compliant tool is SQLPLUS. This allows sql to be executed on a remote IP21 host and display the results on the users desktop window.

The Oracle database offers its own SQL remote client as well as the point and click OEM(Oracle enterprise manager). Both the tools allow remote access and administration of the Oracle database instance. The OEM in fact generates the SQL to perform the administrative task specified using the point and click GUI interface. The OEM allows a DBA to instantly view all the Database nodes running on the LAN.

Another important benefit of these GUI based remote access tools is that they allow the DBA to view the database status quickly and efficiently without having to remember the gory details of the system tables and views and clunky SQL syntax needed to extract relevant parameters from them. OEM is also useful since the security can be setup once in the OEM security manager and the access can be securely performed by a authorized DBA by logging in at the OEM console.

3.1.4 Change of state detection by triggers

Detection of the change of state(COS) of any field in the database due to an update or insert and the ability to trigger specific functions based on the change of state is a typical feature of real-time databases. This COS detection is the central technique behind the event driven functions of real-time databases. COS detection allows a real-time database to deliver efficient data driven switching without having to resort to polling.

Typical examples of event driven functions are

1. COS driven GUI updates for process values
2. Exception based alarm updates to alarm screens
3. Alarm and even logging to history tables/files
4. Automatic Initiating control tasks based on change of state of hi and low limit alarms
5. Initiating of control tasks due to COS of database field due to operator key/mouse input

Such event driven functionality is very useful in providing a fast response to the COS event while monitoring a large number of potential COS event sources.

In a typical real-time database application in steel rolling the data from over 1000 individual fields out of the real-time database needed to be displayed on the mimic screens. The maximum lag between the value changing in the database and the mimic screen icon displaying the value being updated was specified as 1 second.
There were 30 operator GUI terminals, each of which could be displaying any one of 50 different mimic screens. Each mimic contained between 1 to 100 fields, drawn from the total set of 1000 monitored fields.

As is apparent any attempt to provide a polled solution with a similar sub second response will be prohibitive in terms of processing power and will impose a severe performance penalty on account of the background polling action.

We have attempted to show how such a COS driven solution for GUI update could be implemented using database triggers coupled with a Java stored procedure.

COS detection and function Invocation implementation using a Java enabled object relational mainstream database

We implemented the COS detection feature and invocation of functions using Java stored procedure triggers. In end of 1998 and 1999 vendors have started integrating Java and object features into their mainstream relational database products. The result is very powerful because it brings allows the stored procedures and triggers written in Java to be stored in the database. The Java code is actually executed in a JVM(Java virtual machine), which is integrated into the database. This provides a significant performance benefit since the net layer is eliminated. Being able to code in Java allows the usage of all the standard Java packages and programming techniques with potential reuse of code. The current product still runs the Java stored procedure as interpreted code. We were not able to notice any slowness in our Java trigger execution. It may be noted that the vendors have promised to allow the Java stored procedure to be natively complied in the next release of their product. This should take care of performance limitations in case of applications with heavier throughput.

Implementing COS on table cell to update GUI update

To do this we used a GUI applet that ran in IE4.0 from and was hosted from a web server running Apache (1.3.6 for Win32). We were able to shoe horn a small Java application, which used the java.net.MulticastSocket class to multicast strings on the LAN into a stored procedure.

We then created a row trigger, which invoked the stored procedure on detecting any change of state in the stringValue column of the TestClass table.

```
SQL> desc testclass
 Name                           Type
 INTVALUE                       NUMBER
 FLOATVALUE                     NUMBER
 SHORTVALUE                     NUMBER
 DOUBLEVALUE                    NUMBER
 STRINGVALUE                    VARCHAR2(10)
 DATEVALUE                      DATE
 STRINGVALUEARRAY1DIM           VARCHAR2(40)
 INTVALUEARRAY1DIM              NUMBER_ARR_5
 FLOATVALUEARRAY1DIM            NUMBER_ARR_6
 SHORTVALUEARRAY1DIM            NUMBER_ARR_7
 DOUBLEVALUEARRAY1DIM           NUMBER_ARR_8
 INTVALUEARRAY2DIM              NUMBER_ARR_6
 INTVALUEARRAY3DIM              NUMBER_ARR_24
 DATEVALUEARRAY1DIM             DATE_ARR_4
```

Automatic Update Generation to GUI

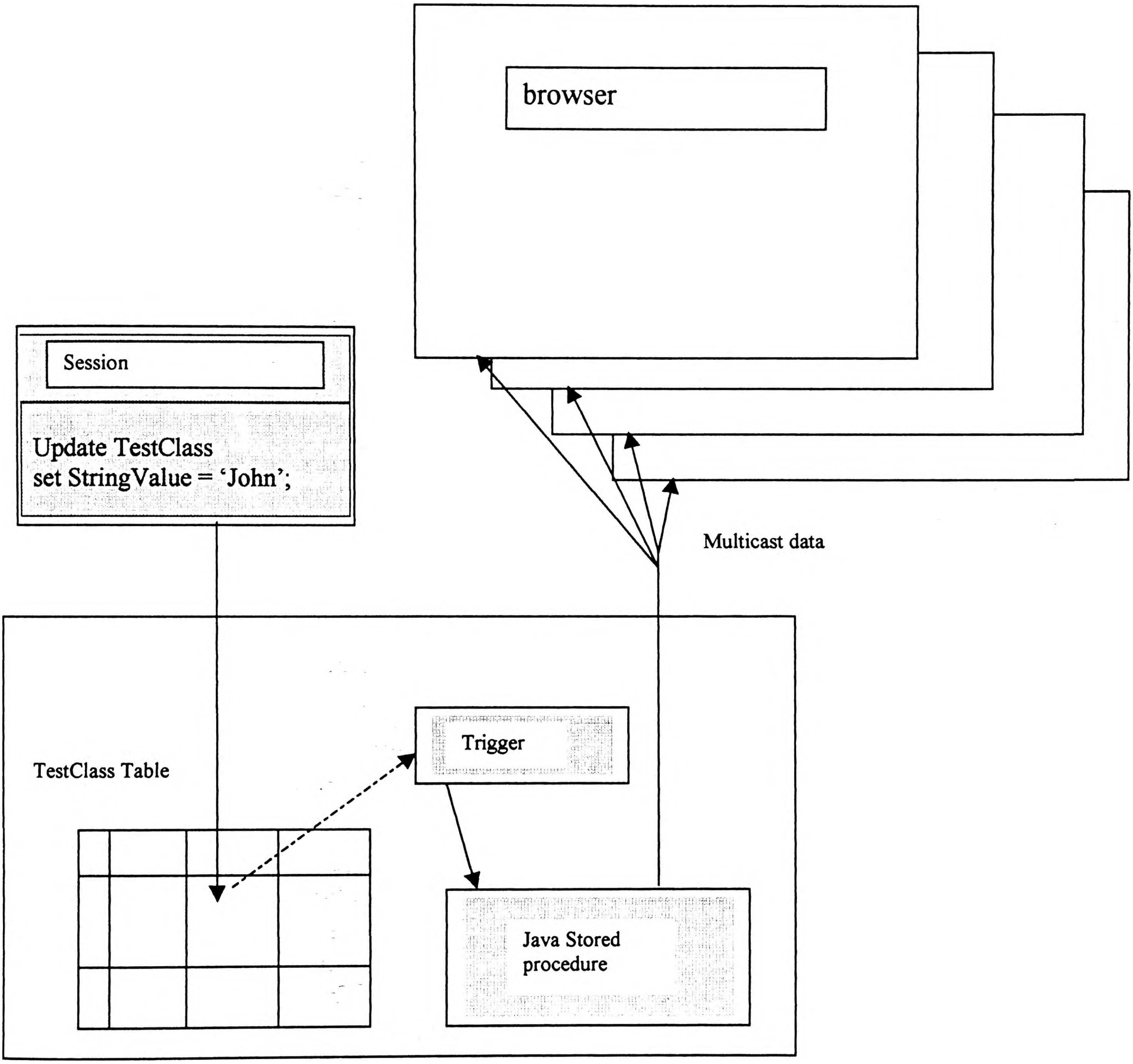


Fig 3.1

3.1.5 Alarm management subsystem

In this section we describe the implementation of an alarm management system in an object relational mainstream database. The alarm management system implemented here mimics the alarm management features available in the IP21 real-time database.

The implementation of the alarm management subsystem consists of a master alarm table called SYSTEM.ALARM_MASTER. This table has 1000 rows each defining an alarm. To turn on a specific alarm the application process has to update the ALARM_STATE field master alarm table. Note this act of turning on an alarm could also be done by a trigger which monitors a field in the database and compares it against Hi and Low limits. If after an update the field breaks the Limit the alarm is turned on.

The recently turned on alarm may be a member of several views. The different views are used to logically group the alarms. Different users may open the view that contains the alarm group of interest to them. Examples of such groupings are:

1. plant areas ie. furnace, roughing mill, finishing mill, down coilers,
2. engineering functions like electrical, mechanical, instrumentation
3. functional groupings like operator alarms, maintenance alarms

In this case there are 12 views called ALARM_VIEW1 through to ALARM_VIEW12 which are subsets of the SYSTEM.ALARM_MASTER table.

To make an alarm visible in ALARM_VIEW4.
turn the alarm_state column is 'on' and the alarm_view4 column is set to 'view4'.
To make the alarm disappear from ALARM_VIEW4
turn the alarm_state = 'off' or by set the alarm_view4 column = 'off';

The alarm subsystem can be controlled by setting or clearing the alarm in the SYSTEM.ALARM_MASTER table from the user applications.

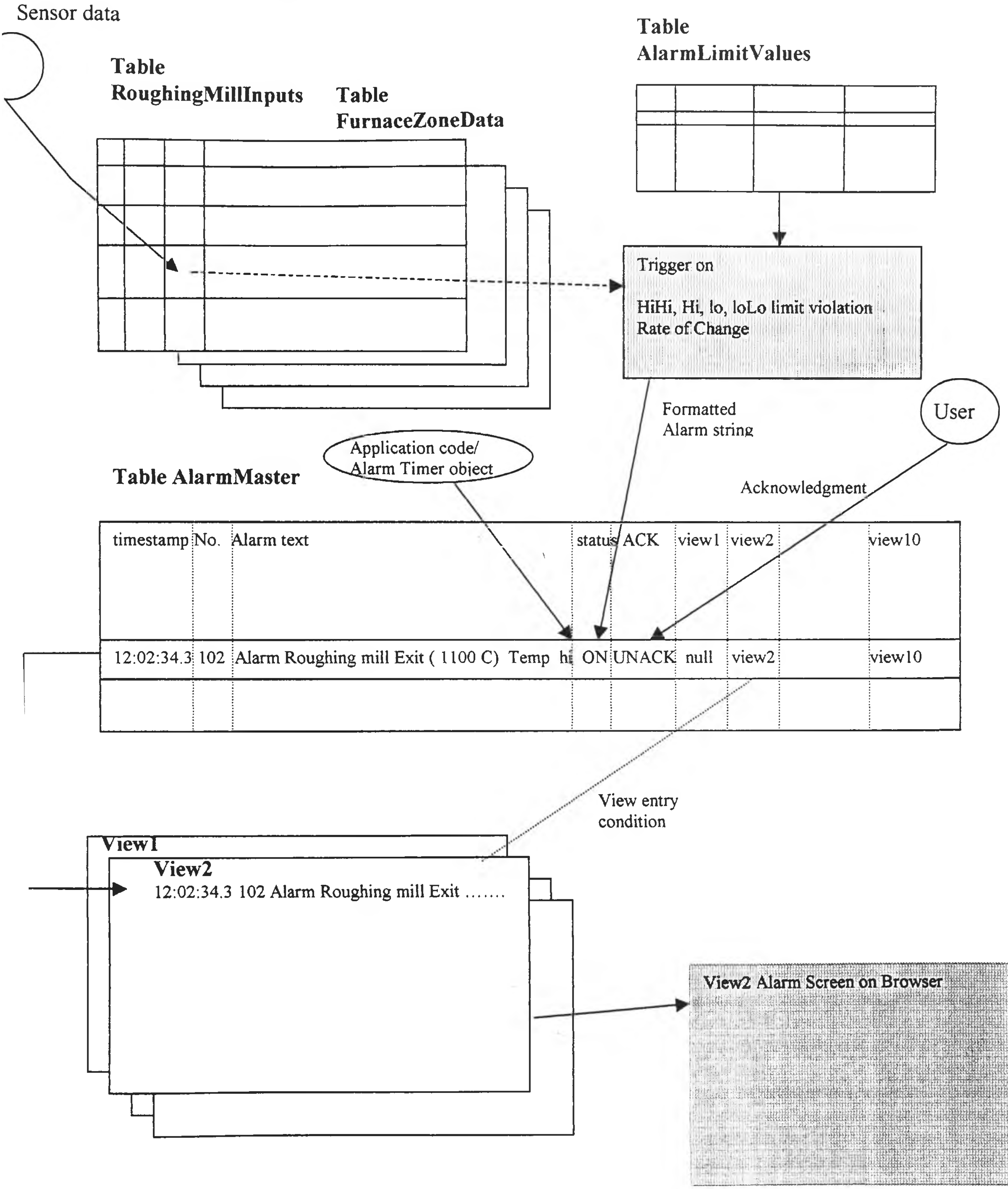


Fig 3.2 Alarm Subsystem Implementation

```
SQL> desc alarm_master
```

Name	Type
ALARM_NUMBER	NOT NULL NUMBER(5)
ALARM_TIME	DATE
ALARM_PRIORITY	NUMBER
ALARM_TYPE	VARCHAR2(10)
ALARM_ACK_REQUIRED	VARCHAR2(3)
ALARM_STATE	VARCHAR2(3)
ACK_STATUS	VARCHAR2(10)
ALARM_ENABLE	VARCHAR2(8)
ALARM_ISOLATED	VARCHAR2(8)
ALARM_VIEW1	VARCHAR2(8)
ALARM_VIEW2	VARCHAR2(8)
ALARM_VIEW3	VARCHAR2(8)
ALARM_VIEW4	VARCHAR2(8)
ALARM_VIEW5	VARCHAR2(8)
ALARM_VIEW6	VARCHAR2(8)
ALARM_VIEW7	VARCHAR2(8)
ALARM_VIEW8	VARCHAR2(8)
ALARM_VIEW9	VARCHAR2(8)
ALARM_VIEW10	VARCHAR2(8)
ALARM_VIEW11	VARCHAR2(8)
ALARM_VIEW12	VARCHAR2(8)
ALARM_TEXT	VARCHAR2(200)
ALARM_FORMAT	VARCHAR2(200)

```
SQL>
```

1. Show the current active alarms in view4

```
SQL> select alarm_number, alarm_priority, alarm_time from alarm_view4;

ALARM_NUMBER ALARM_PRIORITY ALARM_TIM
------------ -------------- ---------
          57              7 01-JAN-99
         145              6 01-JAN-99
         149              1 01-JAN-99
         150              9 01-JAN-99
         262              4 01-JAN-99
         329              1 01-JAN-99
         423              4 01-JAN-99
         580              5 01-JAN-99
         615              2 01-JAN-99
         796              5 01-JAN-99
         858             10 01-JAN-99
         993              3 01-JAN-99

12 rows selected.
```

2. Now turn the alarm 993 off

```
SQL> UPDATE SYSTEM.ALARM_MASTER SET ALARM_STATE = 'off' WHERE
ALARM_NUMBER=993;

SQL> UPDATE SYSTEM.ALARM_MASTER SET ALARM_STATE =
'on',alarm_time=sysdate WHERE
ALARM_NUMBER=423;

1 row updated.

SQL> select TO_CHAR( alarm_time,'MM/DD/YYYY HH24:MI:SS'), alarm_number,
   alarm_priority from alarm_view4
   order by alarm_priority, alarm_time;

TO_CHAR(ALARM_TIME, ALARM_NUMBER ALARM_PRIORITY
------------------- ------------ --------------
01/01/1999 00:00:00      149          1
01/01/1999 00:00:00      329          1
01/01/1999 00:00:00      615          2
01/01/1999 00:00:00      262          4
01/01/1999 00:00:00      423          4
01/01/1999 00:00:00      580          5
01/01/1999 00:00:00      796          5
01/01/1999 00:00:00      145          6
01/01/1999 00:00:00       57         7
01/01/1999 00:00:00      150          9
01/01/1999 00:00:00      858         10
11 rows selected.
```

Note alarm 993 has been removed

3. Now update an existing alarm and note the change in the time.

The time can also be updated using a trigger.

```
SQL> UPDATE SYSTEM.ALARM_MASTER SET ALARM_STATE =
'on',alarm_time=sysdate
      WHERE
      ALARM_NUMBER=423;

1 row updated.

Commit complete.

SQL> select TO_CHAR( alarm_time,'MM/DD/YYYY HH24:MI:SS'), alarm_number,
      alarm_priority from alarm_view4
  order by alarm_priority, alarm_time;

TO_CHAR(ALARM_TIME, ALARM_NUMBER ALARM_PRIORITY
------------------- ------------ --------------
01/01/1999 00:00:00      149          1
01/01/1999 00:00:00      329          1
01/01/1999 00:00:00      615          2
01/01/1999 00:00:00      262          4
06/10/1999 02:41:57      423          4
01/01/1999 00:00:00      580          5
01/01/1999 00:00:00      796          5
01/01/1999 00:00:00      145          6
01/01/1999 00:00:00       57          7
01/01/1999 00:00:00      150          9
01/01/1999 00:00:00      858         10

11 rows selected.
```

4. Now turn the alarm_view4 'off' for alarm 796 and observe it is removed from the view

```
SQL> UPDATE SYSTEM.ALARM_MASTER SET alarm_view4 = 'off',
alarm_time=sysdate WHERE
ALARM_NUMBER=796;

1 row updated.
Commit complete.

SQL> select TO_CHAR( alarm_time,'MM/DD/YYYY HH24:MI:SS'), alarm_number,
alarm_pr
iority from alarm_view4
  2     order by alarm_priority, alarm_time;

TO_CHAR(ALARM_TIME, ALARM_NUMBER ALARM_PRIORITY
------------------- ------------ --------------
01/01/1999 00:00:00        149          1
01/01/1999 00:00:00        329          1
01/01/1999 00:00:00        615          2
01/01/1999 00:00:00        262          4
06/10/1999 02:41:57        423          4
01/01/1999 00:00:00        580          5
01/01/1999 00:00:00        145          6
01/01/1999 00:00:00         57          7
01/01/1999 00:00:00        150          9
01/01/1999 00:00:00        858         10

10 rows selected.
```

User Acknowledgement of alarms

The users view the alarm screen in an applet running in a browser. The applet interfaces with the relational database tables using the JDBC API. The alarm views are displayed in the browser window and the user gets to select and acknowledge individual alarms on the browser screen. The acknowledged alarm is then updated in the AlarmMaster table using another JDBC call which in turn updates the alarm views.

3.1.6 Statistical functions , averaging, integration, moving average, control charts

Real-time databases have configurable facilities for performing statistical functions on the gathered process data. No code needs to be written to implement these features. In IP21 real–time database, the process data is stored in trend records as timestamp, data value pairs. To generate 10 minute or hourly averages for a process data an averaging record has to be configured and pointed to the original trend record. Counting functions can be provided to count the number of occurrences of digital inputs in specified intervals ie. per hour. Integrating records can be configured to integrate instantaneous values ie. To derive the volume of liquid from the instantaneous flow rate. Again the integrating record has to be configured and pointed towards the trend record storing the flow rate. Moving averages can be performed to smooth noise by configuring moving average records and pointing them to the original trend record.

A generic Java trigger can be provided to handle each of the different statistical functions described above. The results would be collected in another table. The trigger can be added as a row trigger to the table where the timestamped process data is being stored. The implementation of the statistical functions is relatively simple using Java.

3.2 Implementing Compression - Problem 2

3.2 Synchronous online compression - why do we need compression ?

To improve product quality and to be able to quickly detect and arrest any aberrations in quality we need to do the following

1. Continuously measure and store process data for 5 years or more to allow comparison across the same product group when analyzing quality problems. Typically the same product may be made in batches which are months apart.
2. Sampling data at a rate fast enough to catch significant events and the cause and effect associated with them.
3. To retrieve easily stored data for detailed statistical analysis to be able to explore the cause and effect relations.

While the above requirements seem simple they soon run into severe implementation problems when the storage requirements are sized. All current real-time databases recognize this problem and have provided a configurable per point compression ability that operates on the data before storing it. Any attempt to use a mainstream relational database for real-time applications will therefore need to build data compression into the solution.

We will attempt to show how a compression algorithm coupled with an understanding of the nature of the process data can provide a solution.

Some of the criteria the compression mechanism must satisfy are:

- Integrity of the data must be preserved during compression to ensure recording of ALL variations.
- Timestamp must be recorded with sub millisecond accuracy to allow comparison of related process parameters.

- The Compressed data should be selectable across a multi year timespan. Shortcut solutions like zipping of datafiles are not acceptable.
- The compression technique must be fast and simple executing in sub microsecond times per input to allow deployment at the source of the data. Potential deployment targets are the low level devices like the PLC or DCS.
- The data must be compressed by 10 times or better to provide meaningful reduction in storage requirements
- Must allow merging of older data into history to allow lab results insertion.

In the rest of this chapter we will attempt to implement such a historical data system which synchronously compresses the data before writing it into the database meeting the criteria outlined above.

3.2.1 High throughput data storage estimate

We will study the volume of data generated in a real life automation application and observe the need to compress the gathered data. In manufacturing applications similar to the strip mill it is often a statutory requirement to store the complete log of the manufacturing data for a defined period of 7 – 10 years. This is driven by the legal need to be able to certify the quality of the manufactured product in any legal issues arising from any failures of any downstream product manufactured using the strip. Where the manufacturing process is a fast moving operation like strip rolling, the sheer volume of date being generated can be staggering. Also important is the requirement that users should be able to get at the stored data with simple tools like SQL preferably without any manual intervention by a database administrator. We will now attempt to estimate the volume of data in the strip rolling operation.

The data is gathered at a periodicity of 20 observations per second from 200 measured inputs. Each data point is stored as a 8 byte long number. Since this is a continuous operation the amount of space needed to store the equivalent of one days data is

200 * 8 * 20 * 60 * 60 * 24 = 27648000000 bytes = 2.765 Giga Bytes

1 months data = 30 * 2.765 = 82.95Gb

1 years data = 365 * 2.765 = 1009.225 Gb

Each data value store must be stored with its 8byte timestamp. However savings can be achieved by storing a common timestamp for a number of different inputs. This is not hard because low level io devices usually scan all their inputs at the same time.

Even with common timestamps at this rate of generation of data it may be very difficult to store data for more than a month using the currently available RAID arrays of 18Gb SCSI disks.

Once the database grows to such a huge size normal DBA(database administration) activities like backing up data files becomes a technical feat since the backup spans multiple tapes and takes several hours to perform. The latter can be a problem in the 7x24 hour operation environment of plants like a strip mill.

The actual cost of the hardware rises significantly once the typical expandability of the platform is exceeded. Additional cabinets power supplies and cabling and RAID controller cards have to be catered for all this adds cost increases complexity and lowers reliability.

Compression of the data before writing it to disk seems to offer a wonderful opportunity to be able to achieve the desired data storage functionality using a commercially used and much cheaper and more manageable computer and RAID configuration

Keeping in mind the large volume of data, the following are desirable features to make the system meet user expectations.

1. A fast compression algorithm needs to be devised to compress the data by rejecting those observations, which do not represent any significant deviation from the previous trend.
2. To be able to partition the stored data based on time to allow partitions to be taken off line and restored online easily when required.
3. It should be possible to load the stored data into an offline database to allow users to query at will without affecting the production system.

3.2.1.1 Strip mill data compression results

The compression algorithm described in 3.2.2 was deployed on the SOFTLOGIX SoftPLC, which is a NT4.0 COM application. RTP data consists of 128 inputs sampled at 50ms. The ESP GEM Data is obtained from the 7 GEM PLC's which are polled by a Java application at a rate of 32 16 bit words every 250 ms. All the data was put through the compression algorithm. The RTP data was compressed dramatically to 1.15% of the total samples. Whereas the ESP GEM data was compresses only to 31% of the total samples. The compressibility was not as much in case of the ESP GEMS because some of the data consisted of digital inputs for which the passband had to be kept to 0. It is possible to set the passband on an input by input basis so adjusting the pass band for the analog inputs from the GEMs will realize further compression.

The compression results overall are very encouraging because the most of the data volume was resulting from the 128 inputs sampled at 50 ms. And these were compressing beautifully. Based on these results we concluded that the estimated raw data of 5GB / day would compress to ~500 MB/day without losing data integrity.

Further compression is possible by identifying the signals which generate the most data and critically looking at these signals to see if the passbands could be increased to reduce the sensitivity of the compression. In other cases there could be a need to increase the sensitivity for some other parameters.

1. **RTP – Analog Data**

   The following results were taken overnight (5/11/99). Each result represents one hour of data, with the data being sampled every 50msecs. The compression pass-band was setup at 0.5% FSD and each value stored as a float (32 bits).

RTP Data

Hour Intervals	Uncompressed Signals	Compressed Signals	% of Compression Signals
1	9216128	1013134	10.99
2	9213824	853476	9.26
3	9215232	996755	10.82
4	9216000	1103892	11.98
5	9214976	932043	10.11
6	9216256	867110	9.41
7	9215360	1069575	11.61
8	9216256	917267	9.95
9	9216256	1013742	11.00
10	9216256	1012508	10.99
Total	92156544	9779502	**10.612**

Table 3.1

As a result of further tuning subsequent results show we are getting around a compression figure of ~6%.

2. ESP – GEM Data

The following results were taken overnight (5/11/99). Each result represents one hour of data, with the data being sampled every 1sec (Existing system). The compression pass-band was setup at 0.0% FSD and each value stored as a integer (16 bits). Note the passband was set to 0% to record all changes. The reason for this was the fact that some of the data was digital inputs saved as an integer and every change had to be recorded. We may be able to get more by altering the pass band for non digital inputs.

GEM Data

Total Number Of Samples	Total Number After Compression	Compression percentage
691200	223785	32.38
691200	218575	31.62
691200	216531	31.33
691200	217319	31.44
691232	211100	30.54
690528	201089	29.12
691264	212573	30.75
691200	229260	33.17
691200	230415	33.34

Table 3.2

3.2.2 Synchronous compression technique

Compressing the data at the source is very attractive because it reduces the network traffic and the processor load in marshalling and unmarshalling the data at the data source and at the Server end before saving to the database. By profiling the Java code on the data source and receiving end we noted that almost half the cpu time was spent in marshalling and unmarshalling.

The following is a low overhead algorithm to compress and timestamp multiple streams of synchronous data while preserving the fidelity of the trend. The algorithm can be deployed wherever it is convenient. It simplicity and resulting speed allows it to be deployed without any concern about performance.

The algorithm derives its simplicity from the fact that for synchronous data, the delta time is constant so the slope is equal to the difference between the current and last data value multiplied by a constant.

The algorithm coded in java2 and run on a 300Mhz Pentium II executed in 16 microseconds for a block of 200 data points, this translates to 80 nanoseconds per point. Each point was executing a sine wave of amplitude +-4000. This means we will be able to deploy the algorithm on a variety of platforms perhaps even inside PLC's

Here is textual description of the compression algorithm

1. The passband is computed by projecting the slope from the previous 2 points and adding and subtracting a nominated passband.
2. If the current value is within the computed passband it is rejected.
3. If the current value breaks the passband we log the previous value and the current value. Logging of the previous value is skipped where it had already been logged

4. If the time span in milliseconds since the last recorded value exceeds a configured maximum then the value is logged anyway.
5. reject the measured value anyway if it is less than a low limit or above a high limit. This is to take care of misleading observations i.e. temperatures measured when there is no metal in mill.

Java Implementation of the compression Algorithm

presented here is the java routine which implements the compression algorithm described above. The complete source is also available in the appendix

```
  //===========================================================================
  // Function
  // processData
  //
  // = DESCRIPTION
  // This routine recieves an raw data array containing a measured value for each input point in the .
  // system. The routine iterates through the array testing the measured value against the trend established
  // by the previous measured values. If the measured value differs significantly it is selected for
  // storage. The selected value is saved in a the output array called values. The index of the measured
  // value in the raw data array is saved in another output array. The output arrays contain now a subset
  // of the original values  the extent of compression depends on the per point configuration parameters.
  // The routine also updates the trend data in its static arrays for each point.
  //
  // params:
  //  rawData[]   is an array of the input scan data for all the values.
  //  timeStamp   is the long number representing the ms since 1970 associated with this scan. All values
  //              share the common timestamp since the data was gathered by the same input device.
  //  value[]     is the array of  values which were not rejected after processing the rawData[]
  //   tags[]     is the input tag id associated with the corresponding entry in the values[] array
  //======================================================================
//=============================================================================
// Function
// processData
//
// = DESCRIPTION
// run through the array of data and processes one point for compression in each array
//
//=============================================================================
   public short  processData( int[] rawData, int[] values,
                              int[] tags, long timeStamp )
   {
      if( myProcessingBusy == true )
      {
         return -1;
      }
         myProcessingBusy = true;
```

```
Trace.level5("in processData");

int tagCount=0;
  // index of non rejected data entries
for( int i=0; i<myIoArraySize; i++ )
{
  myDeltaY[i]=myYi[i]-myYi_1[i];
    // delta y from last time
  myYi_1[i]=myYi[i];
        // set the value prior to last equal to the last value
  myYi[i]=rawData[i];
    // read the new value


  int c = myYi_1[i] + myDeltaY[i] + myPassBand[i];
  int d = myYi_1[i] + myDeltaY[i] - myPassBand[i];
    // calculate pass band limit

  if( i == 10)
  {
        System.out.println("entering loop in processData. myDeltaY[i]="+
        myDeltaY[i]+" myYi[i]"+myYi[i]+" myYi_1[i]"+myYi_1[i]+" c="+c+"
        d="+d+" mypb[i]="+myPassBand[i] );
  }

  myI=i;
  boolean skipFlag = false;

  // record the event that a value is about to be logged because the
  // number of allowed skips has been exceeded
  if( ( timeStamp - myLastLoggedTime[i] ) > myMaxSkip[i] )
  {
    skipFlag = true;
  }

  if( ( myYi[i] > c ) || ( myYi[i] < d ) || ( skipFlag == true ) )
    // test if the new data is outside the allowable pass band
  {

   if( myFlagYi_1Logged[i] == false  && skipFlag == false )
      // log the last value too if it was a point skipped by the compression logic
      // earlier. The  myFlagYi_1Logged[i] is set true when a new point is logged
    {
      values[tagCount]=myYi_1[i];
        // save value
        tags[tagCount]=i+1;
          //save index of non rejected tag. This will point to the history record
          // i.e.History54 where i = 53

        if( i==10 )
          {
                System.out.println( "i="+i+" Retaining Yi-1
                values["+tagCount+"]="+values[tagCount]+" sf="+skipFlag );
          }

          tagCount++;
          // increment count of non rejected tags
      }
          values[tagCount]=myYi[i];
        // save value
      if( i==10 )
      {
          System.out.println( "i="+i+" Retaining Y i
          values["+tagCount+"]="+values[tagCount]+" sf="+skipFlag );
      }
      tags[tagCount]=i+1;
        //save index
      tagCount++;
        // increment count

      myFlagYi_1Logged[i] = true;
        // flag the logging into history of the current point
     myLastLoggedTime[i]=timeStamp;
        // save the time when last reading was logged
    }
    else
      // skip the point for now it is within the pass band
    {
      mySkipCount[i]++;
      myFlagYi_1Logged[i] = false;
    }
    int k = Math.max(0,tagCount-1);
    myTotalCount[i]++;
      // count total vlues for this point
    }
    myProcessingBusy = false;
    return  (short)tagCount;
}
```

The following is an extract from a configuration file which is dynamically monitored by the compression algorithm to allow the compression criteria to be tuned while it is operating

Configuration file

```
#==========================================================================
# FILE NAME: HSM.config
# DESCRIPTION: IO compression parameters for 200 points
# AUTHOR: Saugato Mukerji
# CREATED: 08 Mar 1999
# LIBRARY:
#
# REVISION HISTORY:
# Initial revision
#
#
#==========================================================================
#
===========================================================================
# the following defines the max interval for which data may be skipped
#
===========================================================================
HSM.skip1 = 300
HSM.skip2 = 300
HSM.skip3 = 300
. .
. .
HSM.skip199 = 1000
HSM.skip200 = 1000

# ======================================================================
# the following define the deadband
# ======================================================================

HSM.passBand1 = 150
HSM.passBand2 = 110
HSM.passBand3 = 10
. .
. .
HSM.passBand198 = 110
HSM.passBand199 = 110
HSM.passBand200 = 110
```

This figure shows graphically, how the compression algorithm decides to reject or save a datapoint based on the computed passband. Typically only the points falling outside the passband are saved. The point preceding the point being saved is saved too, if it not already saved. The examples of this are at the top and bottom of the sine wave. This allows the algorithm to record faithfully the points of discontinuity.

A graph showing the pass band based operation of the compression algorithm

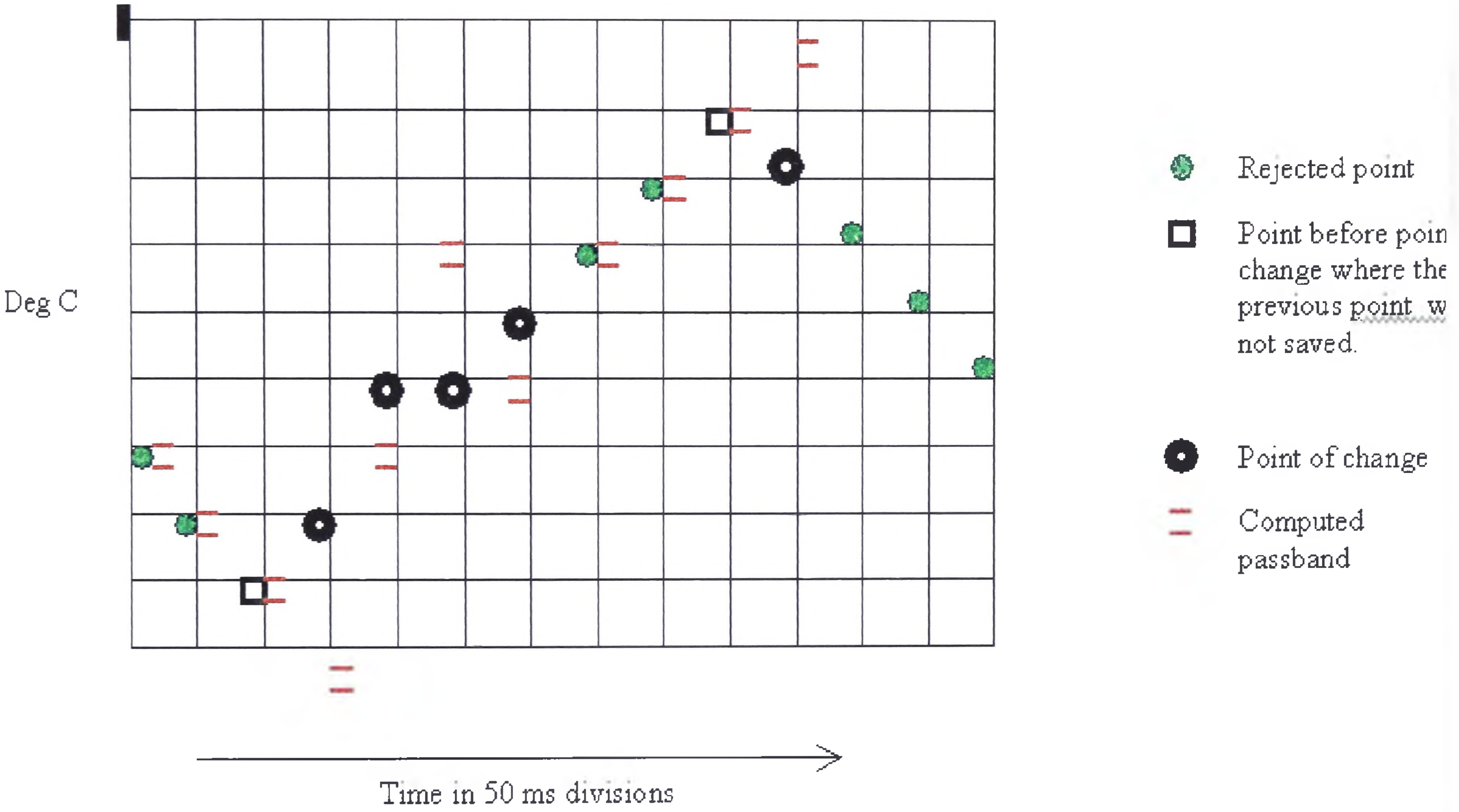
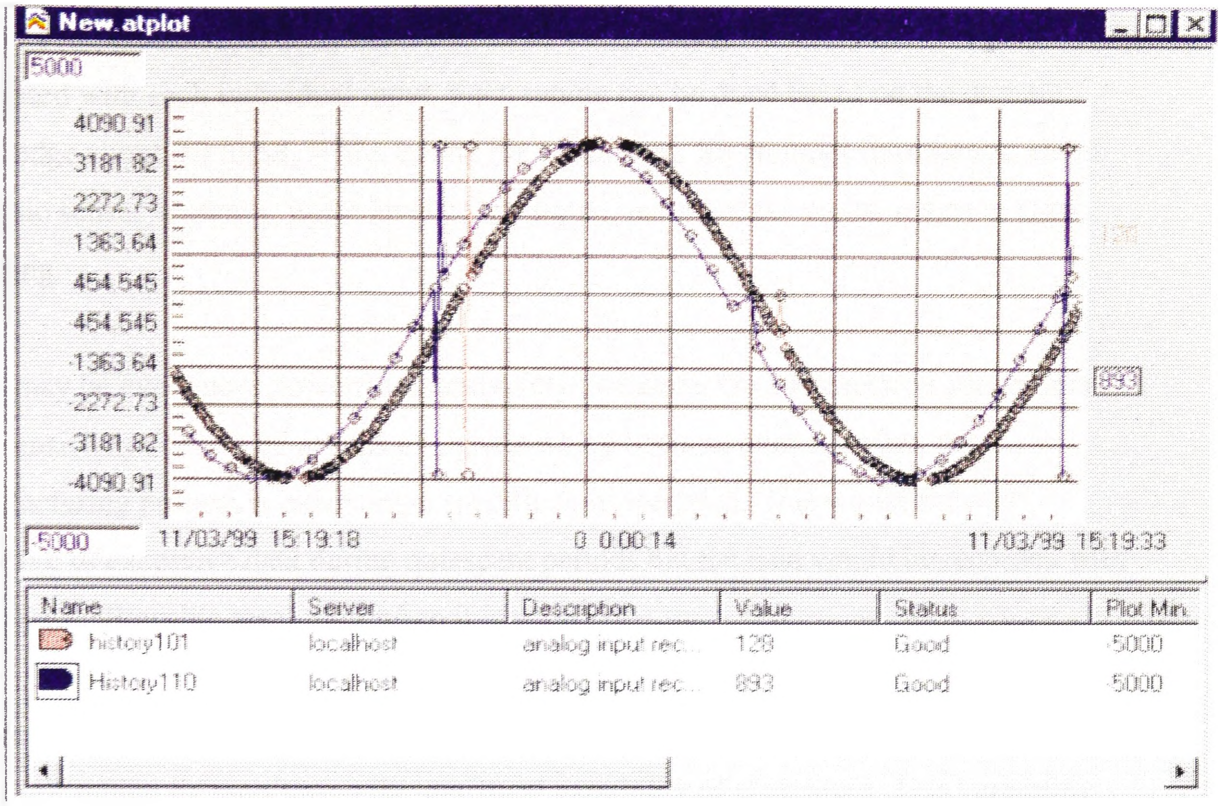


Fig 3.2

Fig 3.3



Name	Server	Description	Value	Status	Plot Min
history101	localhost	analog input rec...	128	Good	-5000
History110	localhost	analog input rec...	893	Good	-5000

A graph showing the max skipped time-span based operation of the compression algorithm

The above figure shows two variables one of which is compressed and the other uncompressed. The two variables were given the same simulated data with a small delay. The simulated data was a sine wave with disturbances. The compression algoritm was set up in this case to have a very high passband so that only when the maximun interval was exceeded was the value recorded. The compressed variable shows that the shape of the graph was preserved and the disturbances were recorded faithfully. As expected the number of datapoints saved was much less in the compressed variable.

3.2.3 Algorithm design with adaptive tuning

Further compression may be achieved by selectively tuning the compression parameters associated with each individual input. Such tuning can be done based on the domain knowledge about the input. When tuning the parameters for an input information about the basic sensor accuracy, noise level of the signal, nature of the signal ie fast or slow changing.

What may be even more powerful is adaptively switching the compression parameters to high sensitivity for related groups of signals which become relevant when the manufacturing process is executing a specific unit operation. This would prevent gathering of excessive data during quiescent periods but the data would be recorded with the highest allowable resolution when it is valid.

Where such domain knowledge is not available the compression algorithm itself can be made self tuning by monitoring the statistical variability of each input. This variability can be used to dynamically specify the compression parameters. This may be a highly relevant approach where the number of inputs are in the order of 10000 to 20000 as is the case in thermal or nuclear power plants.

Further sophistication can be built to allow compression parameters to switch automatically to high sensitivity when a bump or a large variation is observed. The sensitivity can be gradually relaxed to a relatively less sensitive quiescent setting(which yields better compression) once the variation in the data has declined.

3.2.4 Using tables with many columns to improve the data rate

When benchmarking the object relational database for inserts it was soon realised that the number of separate inserts which could be performed per second was limited. Increasing the number of columns in each row did reduce the insertion rate but the decline was not in proportion with the increase in the number of columns. The bench mark figures in the earlier section on update/insert rate limitations show that the insert rate fell from 150 to 80 inserts per second when the number of columns in the table increased from 10 to 289. So a 30 fold increase in the number of columns led to the insert rate falling by a little less than 2 times. The CPU loading was never more than 30% since the task seemed to be disk bound.

Obviously the object relational database is incurring a certain fixed overhead when we insert a row into a table irrespective of the number of columns inserted. The number of columns can best be treated as a variable overhead.

So where we need to insert large amounts of data at a high scan rate (ie the 200 points per second at 50ms per scan) it is probably best to keep the data in a single table if possible. If that is not possible then the number of tables should be kept as low as possible.

Since the task is disk bound it would be effective to locate the tables on separate spindles. Further the object relational databases are designed for symmetric multiprocessor architectures so the data rate would improve if a 4 cpu or 8 cpu box was used.

Further benefits are realised when additional instances of data inserting programs are used. While the individual insertion rate per insertor may drop the cumulative insertion rate increases. A bench mark on a 300 column table using a single insertor inserted 80 rows/second whereas two instances of the insertor running together inserted 60 rows each taking the total to 120 rows/second.

3.2.5 Effect of sparseness on amount of space used per row

Time series data from sensors lends itself very well to compression. Compression in this case means not storing those measurements, which do not deviate significantly from the trend, defined by the stream of measured values.

Where the time series data is being gathered synchronously at predefined intervals by the data acquisition system a complete set of measurements and the absolute time is stored as a row in the relational table. It is typical to store the time in terms of the number of milliseconds since midnight of 01-01-1970. Most operating systems provide a call, which returns this.

However when the compression algorithm rejects the data for a particular measurement it still has to be stored in the table. It is possible to store a null in place of the rejected data. Where the data trends are slowly varying the compression algorithm will replace many of the observations by nulls leading to the rows becoming highly sparse.

We did a practical experiment by inserting in turn 10000 rows each of 0%, 90% and 100% sparseness into a table called NUMBERTABLE. NUMBERTABLE has 289 columns of numbers. We observed the free space used space, blocks and extents used by the number table using the DBMS_DDL.ANALYZE_OBJECT (...) function after inserting each lot of 1000 rows.

0% sparseness

Let us first consider the results for 0% sparseness i.e. there are no null values in any row. Predictable the space used increased and the free space declined by an identical amount of 2000kb after inserting each lot of 1000 rows. The blocks used increased by 1000 for each 1000 rows. Considering the block size is 2kb this matches perfectly the changes in free space and used space.

Considering each row has 289 columns the space taken to store a single number is = (2000x1024)/ (289*1000) = 7.986 bytes/ value. This may go up by a small amount when we add an index.

90% sparseness

Next let us consider the scenario most likely to occur when storing process trend data after rejecting the values not significantly differing from the trend. Typically an online compression algorithm is used to reject the values not significantly differing from the trend by replacing such values by nulls. Such data may end up having a sparseness of around 90% i.e. only every 10th value is non null in each column or in other words 10% of the measured values are sufficient to represent the trend accurately. The Compression algorithm described in earlier chapters achieved the 90% rejection rate on sample data.

As before the space used and free space, changed in opposite directions by identical amounts. The space was measured after inserting each lot of 1000 rows. In this case amount of change in space was 600kb for 1000 rows. The change in the number of blocks used by the NUMBERTABLE for each 1000 rows was 300 blocks of 2kb each.

Since the original pre compression data was of the same size the effective storage used per value with 90% sparseness is
=(600x1024)/ (289*1000) = 2.126 bytes/ value.

That's a saving of 100*(7.986 – 2.125)/7.986 = 73.37%

Thus the effective space needed per day to store 200 points of data measured 20 times a second after compression =200* 24*60*60*20*2.125/ 1000000 = 734.4Mb/day

100% sparseness i.e. down day

To simulate the condition when the plant is down where no new measured values are being generated - we tested with 100% sparseness.
10000 rows needed 200kb. So the effective space, required to store each null value, was =(200x1024)/(289*10000) = 0.0708 bytes/value
That's a saving of 100*(7.986 – 0.0708)/7.986 = 99.11%
Thus the effective space needed per day to store the 200 points of on a plant down day =200* 24*60*60*20*0.0708/ 1000000 = 24.5Mb/day

If the compressed data were to be stored in a real-time database with a compression ratio of 1:10 the volume of data per day would only be a little lower. The real-time stored 15 bytes for the timestamp and value, however there was no overhead associated with the rejected values
Ie 200* 24*60*60*20*0.1*15/ 1000000 = 518.4Mb/day

Conclusions

There is a 70% reduction in volume of disk space used if a slow moving time-series data are compressed with a trend based rejection algorithm and then stored into a relational table columns formatted with a variable length data type. This technique achieves the benefits of compression while retaining the ability to perform SQL queries on the entire or any sub part of the entire chronological dataset.

A related observation was that the insert rate improved as the sparseness of the table increased. We observed that the time taken to insert 1000 rows was 16.5, 13 and 9.5 second for sparseness values of 0%, 90% and 100%. This is explainable by the fact that as data becomes sparser less bytes have to be written to rollback and redo log segments.

Storing the compressed data in relational tables with nulls representing data took 734.4 MB/day whereas storing the same data at the same 1:10 ratio in the real-time database took 518.4MB. So while the relational data took a little more space it may still be a better choice from the scalability and robustness point of view.

3.2.6 Effect of sparseness on table with varray columns

The Oracle VARRAY datatype allows a predefined array structures to be used to define a column in a table just like any conventional primitive datatype like number, char, varchar etc. The VARRAY construct is extremely useful in mapping the data structure of real world objects into the object relational tables in the RDBMS.

In this section we have examined the space occupied by a row of such a object relational table which has half its columns defined as VARRAYS. We will also examine the reduction in space occupied when the row becomes highly sparse. Reduction in space occupied with increase in sparseness is a highly useful attribute when gathering data synchronously at a high scan rate. Real world data often alternates between relatively non sparse and very sparse. A typical example is data resulting from process measurements on in a batch process. Typically data is generated in different process stages by different input sensors. As a result the data from a particular sensor is sparse except when the batch process is operative in the section of plant where the sensor is located.

To examine the effect of sparseness on the amount of space saved when inserting a row we tested a table called TESTCLASS with NON-NULL data and then with highly sparse data where every column an VARRAY element was NULL except one column. We inserted data in blocks of 10000 rows at a time and analysed the table after each block. The results are tabulated below and also presented as a graph.

The enclosed graph and table clearly shows that significant saving could be obtained by ensuring that the data null when the values are insignificant.

The Physical significance of sparseness in a real life

To illustrate the point let us consider the process of rolling of a coil. The process starts at the roughing mill which we designate as zone 1 and proceeds through zone 2 and zone 3 to end up in the delivery area designated as zone 4. In each zone there are 75 sensors making a total of 300 sensors, which gather process data. Typical process parameters are strip width, thickness, length, tension, roll motor amps, strip temperatures across the width of the strip from top and bottom metal detectors for position sensing etc. The sensors are mounted at several locations in each zone to monitor the progress of the slab.

The data from each sensor is scanned at a high speed of 20 scans /sec. This data is written into the database immediately. It is now obvious that the data from each sensor is relevant for only a small fraction of the rolling period of the coil. The rest of the time the sensor data in many cases is irrelevant. It is relatively easy to detect the relevance or irrelevance of the data by comparing it with predefined HiHi and LoLo limits.

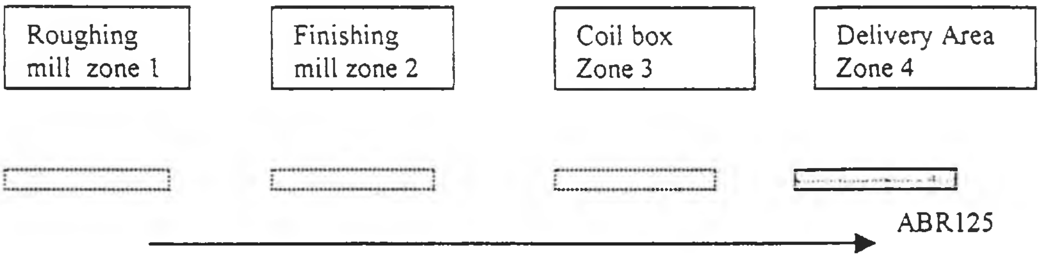


Fig 3.4

In real life the situation is a little more complicated as there is usually several bars in the mill at the same time. The number of zones too is further broken up into many more.

A very important requirement is the ability to be able to quickly retrieve in real time the data from an adhoc subset of the 300 data points for an arbitrary time span in history. Typically such data is used to perform fault analysis, prepare response to customer queries about product quality etc.

Arguably this is easy if you had a table with 300 columns. Oracle can handle the data rate of inserting 20 rows per second with no trouble. Elsewhere in this thesis there are benchmarks of upto 60 rows of 300 columns on a pentium 300Mhz running NT4.0 with a single ide 8GB disk. The bottleneck is disk io. The cpu usage was only 30%.

The problem however is in the volume of the actual stored data. Benchmark data elsewhere in the thesis shows it takes 2MB to store 1000 rows of 300 columns of NON NULL data.

Considering we have 20 rows to insert per second and there are 86400 seconds/ we are looking at inserting of NON NULL data, we are looking at

(2.048/1000)*20*86400 = 3539.9MB /day

Note even if a row has all zero values it is still non sparse because Oracle has no way of distinguishing a 0 from 234.56 hence it allocates the same amount of space to store both.

As we have noted in section 3.2.5 replacing 90% of the column values with nulls the storage requirement reduced by 73.37% to 942.67 MB/day.

The table below shows that this benefit of saving in space extends to tables with Varrays as well. In the example below each row in the table has 66 effective colums after counting all the Varray elements. We stored only one non null value per row to get a spareseness of 1 by 66 or 98.5% spareseness.

Space needed to store 50000 rows on NON Null data = 21053440 bytes

Space needed to store 50000 rows on 98.5% Null data = 1597440 bytes

% reduction when 98.5% sparse data used

= 100*(1 – (1597440 / 21053440))

= 92.41%

Fig 3.5

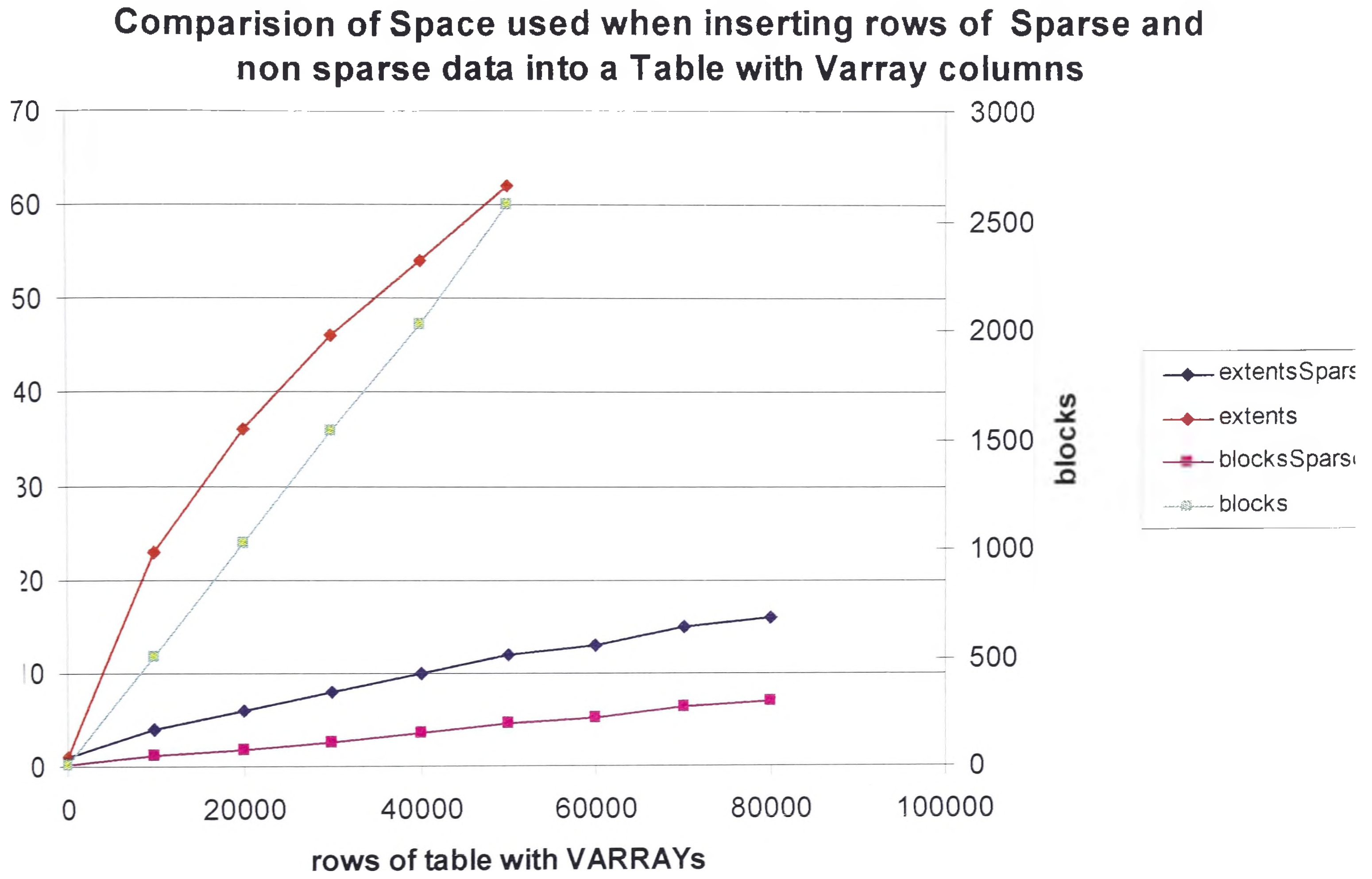

Comparision of Space used when inserting rows of Sparse and non sparse data into a Table with Varray columns
70
60
50
40
30
20
10
0
3000
2500
2000
1500
1000
500
0
blocks
0
20000
40000
60000
80000
100000
rows of table with VARRAYs
extentsSpars
extents
blocksSpars
blocks

Table 3.3

Table with 7 primitive and 7 VARRAY datatypes with non-Null data in every cell

sparse%	rows	bytes	Extents	blocks	free	used	delta_time
0	0	81920	1	10	178344	188592	0
0	10000	4136960	23	505	175176	192872	48
0	20000	8396800	36	1025	170496	197712	61
0	30000	12615680	46	1540	165896	202152	65
0	40000	16588800	54	2025	161456	206592	67
0	50000	21053440	62	2570	156496	211552	65

Table with 7 primitive and 7 VARRAY datatypes with all Null except two columns

sparse%	rows	bytes	Extents	blocks	free	used	delta_time
0	0	81920	1	10	178344	188592	0
98.5	10000	409600	4	50	177704	189232	48
98.5	20000	655360	6	80	177448	189712	50
98.5	30000	942080	8	115	176696	190592	55
98.5	40000	1269760	10	155	176112	191432	55
98.5	50000	**1597440**	12	195	175528	192152	55
98.5	60000	1802240	13	220	174864	192992	56
98.5	70000	2211840	15	270	174400	193632	55
98.5	80000	2416640	16	295	173976	194072	50
	blocksize	8192		initial extent	57344	Max extent	121

SQL> DESCRIBE SCOTT.TESTCLASS

```
Name                                   Type
INTVALUE                               NUMBER
FLOATVALUE                             NUMBER
SHORTVALUE                             NUMBER
DOUBLEVALUE                            NUMBER
STRINGVALUE                            VARCHAR2(10)
DATEVALUE                              DATE
STRINGVALUEARRAY1DIM                   VARCHAR2(40)
INTVALUEARRAY1DIM                      SCOTT.NUMBER_ARR_5
FLOATVALUEARRAY1DIM                    SCOTT.NUMBER_ARR_6
SHORTVALUEARRAY1DIM                    SCOTT.NUMBER_ARR_7
DOUBLEVALUEARRAY1DIM                   SCOTT.NUMBER_ARR_8
INTVALUEARRAY2DIM                      SCOTT.NUMBER_ARR_6
INTVALUEARRAY3DIM                      SCOTT.NUMBER_ARR_24
DATEVALUEARRAY1DIM                     SCOTT.DATE_ARR_4
```

Table 3.4

3.2.7 Partitioning a large high throughput database for administrative ease

Even with data compression the high throughput table with 300 columns will grow at 0.5 GB /day when data is scanned 20 times a second. When the database stores 5 years of data its size is likely to be more than 800GB.

Partitioning the table can offer great benefits when performing administrative tasks like backups. If partitioning was not available we would have to backup the entire table. Current fast tape speeds are at best 4MB/sec on a 70GB DLT tape. At this rate a backup would span 12 tapes and would take over 55 hours. Needing to swap tapes 12 times makes backing up impossible to automate. Also such a long backup duration makes online backups unwieldy. In many cases a 55 hour down time for a closed full backup is impossible.

It is now possible to partition the table into many smaller manageable partitions. The benefits are

- We get complete control over exactly where each portion of the table gets created. Each partition can have its individual storage parameters.
- The backup process is greatly simplified because each partition can be backed up separately.
- The older partitions, which contain historical data that does not change can be made read only. Read only partitions do not need to be backed up.
- Users can query more efficiently by focussing on rows within the partition of interest.
- Block corruption in one partition of the table can be isolated and fixed while users continue to work on the rest of the table.

- Preferably partitions of a large table should be in their own tablespaces. This will allow the DBA to take individual partitions offline without disturbing the rest of the table.

3.3 Implementing Software Engineering approach - Problem 3

3.3.1 Database configuration tools and API available in a typical real-time databases for application development

Typical real-time database provide out of the box tools for creating record structures or templates with a GUI driven point and click interface. The application developer is able to define the database structure without having to write a single line of code.

The tools allow complex record structures to be constructed which include a mixture of fixed or array fields. The arrays can consist of a single or a collection of primary fields. The sizes of the arrays are resizable in real time by altering the sizing field in the record. The tools allow record structures to be duplicated and altered easily.

To complement the flexibility in the record structure there is a suite of API functions to access and manipulate the database records. The application programmer can manipulate the Real time database fields from the program in the language of his choice using the API calls.

The real-time database administrative tools are now network wise and are able to talk to and administer the different database instances running on the LAN.

The challenge here is to provide software engineering tools to ease the development when we attempt to use the object relational database for developing real time applications.

3.3.2 Auto-generating table definition from .java class definition

The challenge here is to provide software engineering tools to provide similar ease of development when we attempt to use the object relational database for developing real time applications.

Object modeling tools like Rational Rose or Software Through Pictures are now commonly used to design applications in modeling languages like UML. The Java or C++ skeletal code is generated automatically from the case tools as a deliverable output.

It is very useful to be able to forward generate the object relational database structure based on the skeletal code produced by the case tool. In this section we will describe the implementation of such forward generation scripts. The actual script and the skeletal classes and the table creation sql files produced are contained in the appendix.

The main benefit of forward generating the table definition lies in the fact that there is no manual typing or editing involved in creating the table structure given the .java or .cpp source files. This leads to great ease in creating the database tables where the number of such tables is large. Also there is a great reduction in the risk of introducing errors when making changes and improvements during the rest of the software life cycle.

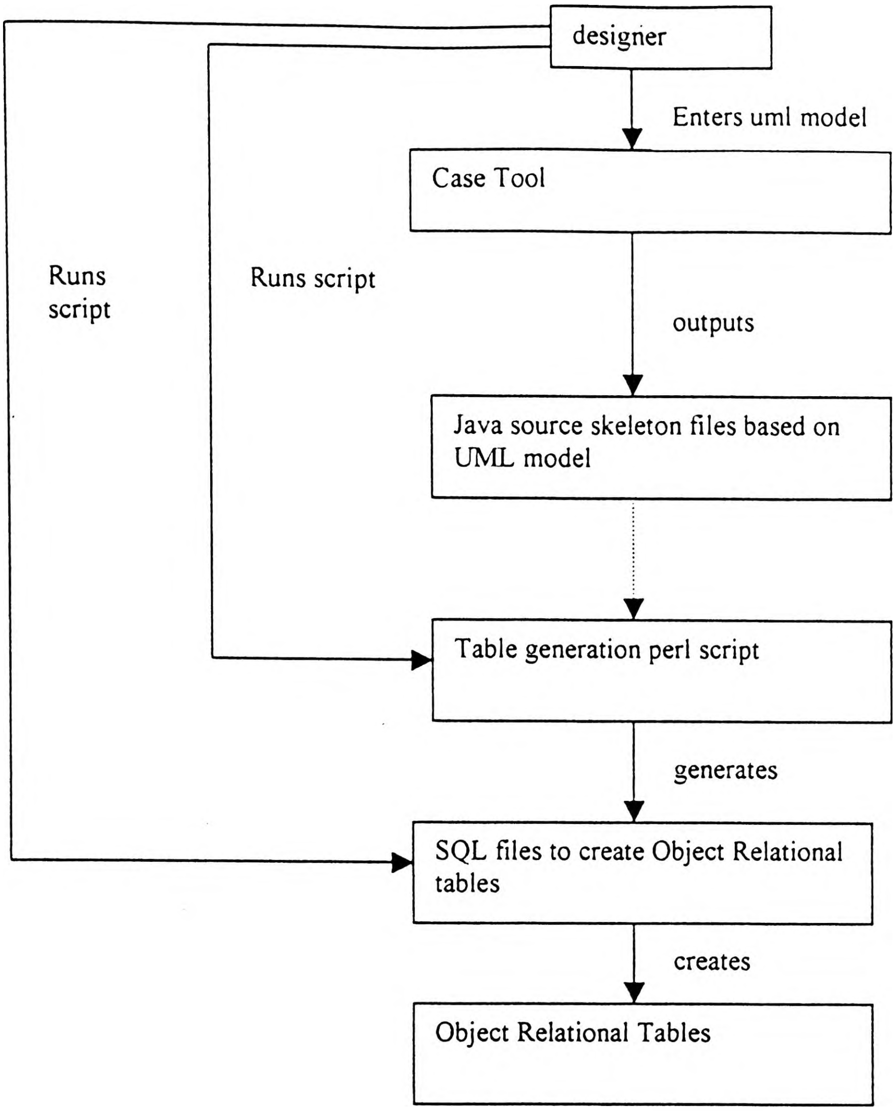

designer
Enters uml model
Case Tool
Runs script
Runs script
outputs
Java source skeleton files based on UML model
Table generation perl script
generates
SQL files to create Object Relational tables
creates
Object Relational Tables

Fig 3.6

3.3.3 Auto-generating Java wrapper library to access object relational tables

To make the task of accessing the application data stored in the object relational tables we need to have a wrapper library which encapsulates the entire activities involved in opening a connection to a database, performing a select, update, insert as applicable and exchanging the query output with the appropriate data members in the Java class instance.

The task involved here is to parse the skeletal Java class and identify the data members. Then special access functions have to be created to exchange data between class instances and the object relational table.

The wrapper functions are incorporated into the skeleton .java file and the augmented source file is created. The augmented file is placed in the appropriate directory structure as defined by the package statement. The application developers can use the assessor functions in the wrapper to manipulate the object relational tables without having to know much about the database and its location. Since the wrapper is auto-generated the incremental effort involved in adding more tables is small. The auto-generation method provides great help in maintenance during the life cycle of the installation.

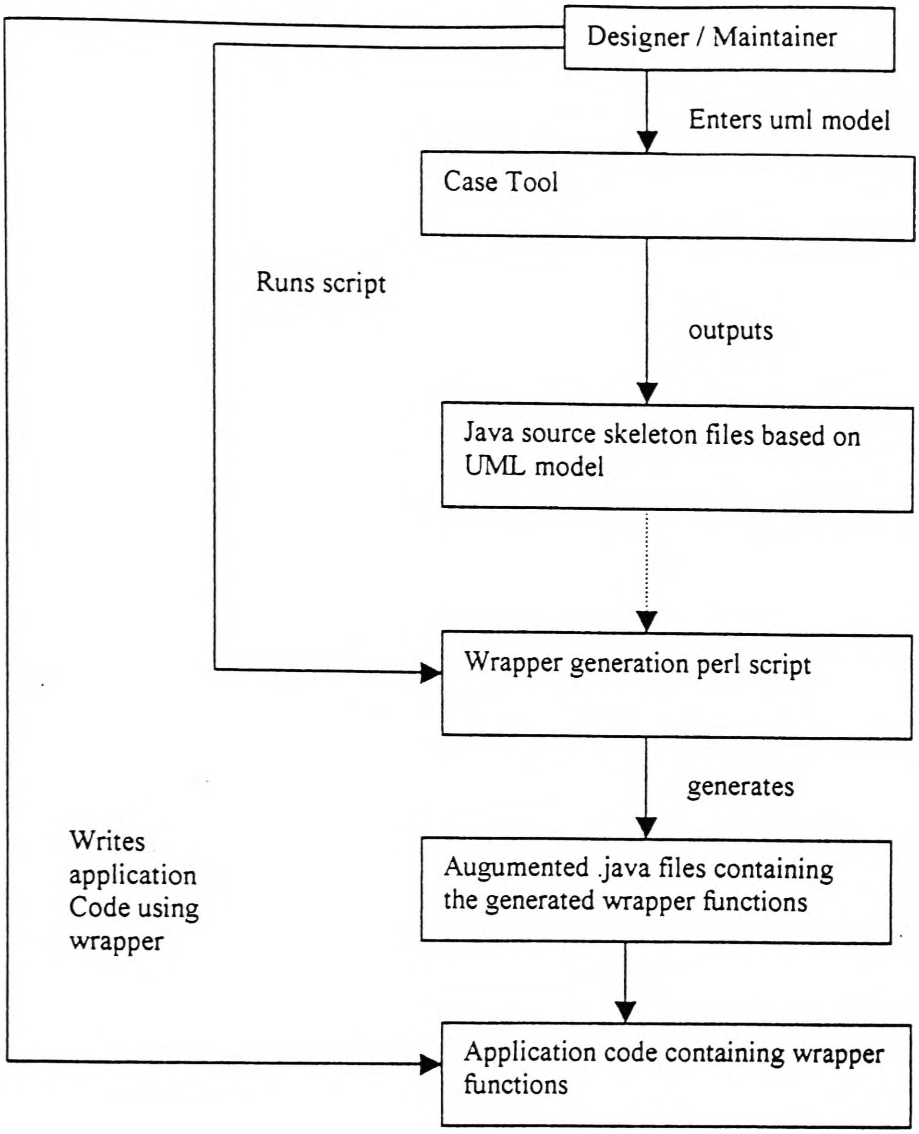

Designer / Maintainer
Enters uml model
Case Tool
Runs script
outputs
Java source skeleton files based on UML model
Wrapper generation perl script
generates
Writes application Code using wrapper
Augumented .java files containing the generated wrapper functions
Application code containing wrapper functions

Fig 3.7

3.4 Performance Issues - Problem 4

3.4.1 IP21 real-time database benchmarks with multiple processes on a multiprocessor system.

3.1.4.1 Test Conditions

Read and write tests were performed on Dual Pentium II 400 MHz, 256 MB, VM limit 857752 KB. We are testing IP21 in a dual processor SMP(symmetric multiprocessing) environment to see if it is able to benefit from the additional processor. To test this we ran several instances of the Java program TestClassTestHarness. The test consisted of repeated reads and / or writes to IP/21 running on the same computer. The IP21 C API library was accessed using JNI from the Java program.

Memory usage before tests was around 310000 KB, usage with all Java programs running was 335000 KB

3.1.4.2 Results

Test results for reading / writing single integer values to / from IP/21, as fast as possible. Number of reads / writes in each test = 10,000,000.

Test conditions	Number of processes	Approx CPU	Average # calls / second / process	Total # calls / second
Reading	1	50%	140597.55	140597.55
Writing	1	50%	135107.75	135107.75
Reading same data point	2	60%	41540.92	83081.84
Reading different data points	2	60%	40607.7875	81215.575
Reading / writing to the same data point	2	60%	41972.719	83945.438
Writing to the same data point	2	60%	40191.0235	80382.047
Reading various data points	3	65%	17810.2103	53430.631

Table 3.5

3.1.4.3 Explaination

The benchmark results using the IP21 real-time database did not improve as multiple instances of the benchmark applications were used. It seems the IP21 real-time database is not able to exploit the additional processor, which is available. Additionally it incurs significant overhead in context switching, locking and unlocking in the SMP environment which degrades performance as additional instances are used. However this degradation may not be experienced in real life situations where the database access will be sporadic unlike the benchmark setup.

3.4.1.1 Relational database benchmarks

We tested the insert performance using JDBC and PLSQL on Oracle 8.1.5 database running on a Pentium II 300 Mhz single processor using tables with 69 and 300 columns The insertion task appeared to be IO bound with significant disk activity and the cpu running at 25%. We observed the following.

Performance on tables with 300 columns on a single cpu system

Table with 300 columns inserted 75 rows/second on a continuous basis using JDBC with all columns having data. This amounts to writing 300 x 75 = 22500 data values/second. The cpu loading was observed to be 28%.

When a second instance of the same program was run the insertion rate on both the programs dropped to 50 rows/second. This amounts to a rate of 300 * 50 = 15000 data values ./ second per program, which is a drop of 33.33%. However the two programs together inserted 30000 data values per second which is an overall increase of 33.33%. The Cpu loading with the two programs running was around 38%. The disk activity was significantly higher.

Relational databases are designed to exploit parallel operation opportunities in multi cpu SMP systems and they also react well to high performance disk controllers and RAID arrays as we will see in the 4 processor benchmark.

Performance on table with 69 columns on a single 300MHz cpu PC

We inserted using a PLSQL program upto 50000 rows of data in blocks of 10000 rows and measured the insertion time for each block. The results from Section 3.2.6 on page 75 show the insert times for the table with 69 columns were 181 rows/second (12545 values/second) where the data was highly sparse and 154 rows per second (10615 values/second) where all the data were non null. As expected the effective throughput declined as the number of columns in each transaction declined. The number of transactions per second did increase from 75 to 153 transactions but failed to match the decline in the column numbers from 300 to 69.

Benchmark using a single under powered single processor (150 MHz AMD K-6 CPU) PC

Table 3.6

No. of instances	Loop count	Elapsed seconds	Calls per second	Combined calls/sec	Cpu %	Comment
Select Benchmark						
1	10000	6	1666.7	1666.7	100	single instance
2	10000	11, 12	869.6	1739.2	100	each selecting same data
3	10000	16,16,17	612.4	1837.2	100	each selecting same data
5	10000	26,27,29 ,29,26	364.96	1824.8	100	each selecting same data
1	10000	6	1666.7	1666.7	100	single instance
2	10000	11,11	909.1	1819.2	100	each selecting different row data
3	10000	17,17,17	588.23	1764.69	100	each selecting different row data
5	10000	27,28,28 ,28,28	359.7	1798.5	100	each selecting different row data
Update Benchmark						
1	10000	14	714.28	714.28	100	single instance
2	10000	28 ,28	357.14	714.28	100	each updating a different row
3	10000	43,43,43	232.56	697.68	100	each updating a different row
5	10000	79,77,76 ,74,78	130.21	651.05	100	each updating a different row

The under-powered 150 MHz CPU makes the task CPU bound which in turn causes the available CPU to be rationed among the concurrent processes. The calls per process decline but the combined calls to the database remains the same. The results here are in contrast with the results on the 300MHz Pentium II where the task was IO bound and running concurrent processes actually enhanced the combined performance by 33.3%.

Multiprocessor benchmarks

Select benchmark on 4 processor 550 MHz Xeon Netfinity 7010

Table 3.7

# of instance	Loop count	elapsed sec	avg. elapsed sec	Avg. calls/sec	combined calls/sec	cpu % used	comments
1	100000	9	9	11111.11	11111.11	28	single instance
2	100000	11, 11	11.5	8695.652	17391.3	53	each selecting same data
3	100000	11,11,10	10.66	9380.863	28142.59	78	each selecting same data
5	100000	13,12,12, 14,14	13	7692.308	38461.54	100	each selecting same data
1	100000	9	9	11111.11	11111.11	28	single instance
2	100000	10,10	10	10000	20000	53	each selecting different row data
3	100000	10,10,10	10	10000	30000	77.5	each selecting different row data
5	100000	13,13,12, 15,14	13.4	7462.687	37313.43	100	each selecting different row data

update benchmark on 4 processor 550 MHz Xeon Netfinity 7010

Table 3.8

# of instance	Loop count	elapsed sec	avg. elapsed	Avg. calls /sec	combined	cpu % used	comments
1	100000	21	21	4761.905	4761.905	29.5	single instance
2	100000	26 ,26	26	3846.154	7692.308	47.5	each updating a different row
3	100000	36,36,39	37	2702.703	8108.108	58	each updating a different row
5	100000	53,45,53, 44,49	48.4	2066.116	10330.58	68	each updating a different row

Analysis of performance

It is obvious from the experimental results that the select and insert rate are directly dependent on the CPU rating. The select rate went from 1667/sec for 150MHz K6 to 4545/sec for 300MHz Pentium II to 11111/sec for a 550MHz Xeon. The cpu loading was 100% for the under powered 150 MHz K6. In the under powered cpu the select rate declined proportionally as more instances were started since the cpu was already saturated. The combined call rate increased marginally as the saturated cpu was rationed between the concurrent processes. In the 550MHz Xeon the select rate did not decline when the 2nd or 3rd concurrent processes were used since the cpu climbed from 28 to 53 to 78%. The net effect of maintaining the select rate as additional concurrent processes were run was to lift the combined select rate over the test period from 11111 to 20000 to 30000 as the 2nd and 3rd concurrent process were used. On starting the 4rth and 5th however the cpu hit 100%, the individual call rates fell sharply by 25% causing the combined rate to increase at a much slower rate to 37313. This tapering off is directly caused by the cpu saturating to 100%.

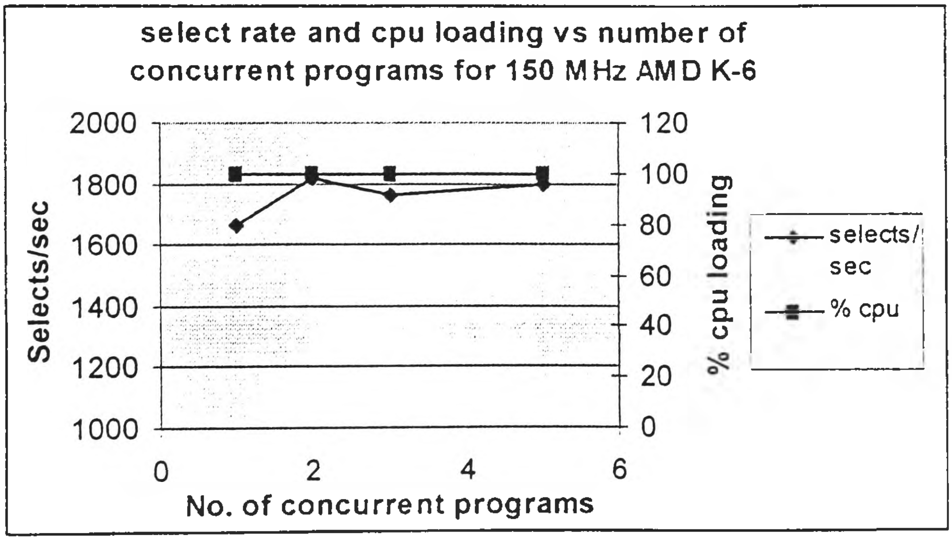


Fig 3.8

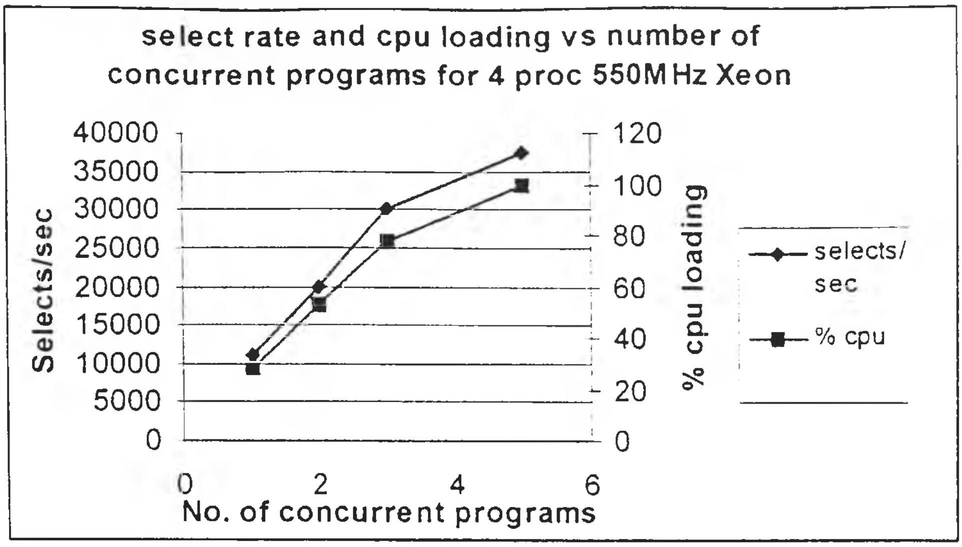


Fig 3.9

Update rate performance as expected is lower than the select rate. The update rates for a single process varied from 714/sec for 150MHz K6 to 3333/sec for 300MHz Pentium II to 4761/sec for a 550MHz Xeon.

In the under powered 150 MHz cpu the update call rate declined proportionally as more instances were started since the cpu was already saturated. As before the combined call rate however only declined marginally as the saturated cpu was rationed between the concurrent processes In the 550MHz Xeon the update rate declined from 4761 to 3846 top 2702 to 2066 updates/sec when the 2nd ,3rd and 4rth-5th concurrent processes were used. The cpu climbed from 28 to 48% to 58 to 68%. The update rate decline as more

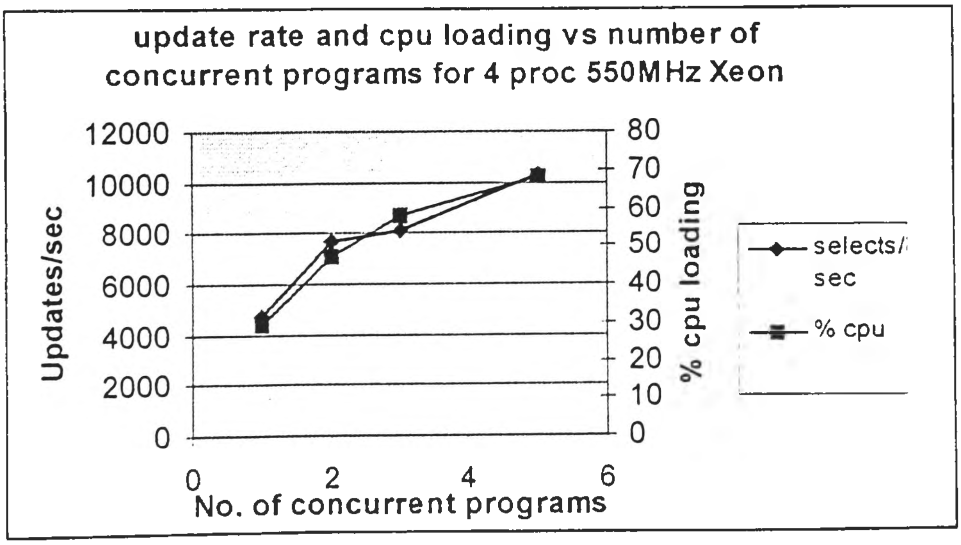


Fig 3.10

instances are started is expected as the system becomes more IO bound. However the disk system though decreasing in response has obviously not saturated, because as the additional available cpu's are pressed into service and the combined call rate lifts steadily from 4761 to 7692 to 8108 to 10330.

3.4.2 Update / insert rate limitations in relational databases

One of the main drawbacks of the mainstream relational databases is the low number of transactions per second, which can be achieved. The reason for this lies in the higher overhead involved in ensuring integrity of transactions by the commit and rollback mechanism. We noted that the overhead applied to each transaction irrespective of the number of columns updated or inserted in the transaction. Our observations show the update rate was similar whether all columns or only one column of a table with many columns was updated in a transaction. Inserts to behaved similarly, we noted this as we timed insertion of a row with non-null data and insertion of a row with almost 90% sparse data.

Lower performance of JDBC as compared to PLSQL and roadmap for the future.

We measured equivalent update and select benchmarks using JDBC. Java database connectivity (JDBC) is a standard SQL database access interface, providing uniform access to a wide variety of databases. However in the rush to launch object relational databases with in built java virtual machines which supported the JDBC standard, vendors were unable to optimize performance to the levels of native procedural SQL languages native to their database.

JDBC Prepared statement update Benchmark on 4 processor 550MHz Xeon Netfinity 7010

# of instance	Loop count	updates/sec	combined updates/ sec	cpu %	Comment
1	100000	663	663	28	single instance
2	100000	494,496	990	50	each updates same row and column same
3	100000	379,378,374	1131	71	each updates same row and column same
5	100000	261,258,258, 249,259	1285	100	each updates same row and column same

Table 3.9

We found the performance of JDBC thin driver from Oracle more than 10 times slower than PLSQL for tables with around 10 columns, this difference is lower for tables with large number of columns. In the next release of the product the vendors expect to close this performance gap by using compiled Java instead of the current interpreted Java. The claim is not implausible because the performance of the C and C++ interfaces to the RDBMSs are comparable with their procedural SQL equivalents.

To measure the maximum update rate we updated a single row in the emp table distributed as part of the demo. This table has only 14 rows and 10 columns. To test the update rate we ran a JDBC application on a client PC which updated the server using a PreparedStatement. The source is in the appendix. The PreparedStatement is a JDBC Interface which allows the prepared SQL statement to be parsed in advance. When the prepared statement is executed, it executes directly on the server without incurring the parsing overhead for each call in real time. This is something like the execution of a SQL statement inside a PLSQL loop.

We ran this update test on a Oracle 8.14 , Pentium II 400 MHz NT4.0 platform with a 8GB SCSI disk, the measured rate for only the execution, was timed over 1000 loops. The results varied from 141 to 153 updates/ second. The server loading was between 10 – 20% as measured on Task manager. On a 4 processor 550MHz Xeon the same test ran on one processor and returned 663 updates/sec this out of proportion increase may be due to the superior performance of the RAID subsystem as opposed to a single SCSI disk.

A similar test was run on a table called vchartable, which contains 289 columns each of which is formatted with VARCHAR2. This test when run from the client inserted 80.4 rows per/second on the 400MHz Pentium II server. The point to note is the insert rate dropped from around 150 to 80 transactions per second though the number of columns increased almost 30 times from 10 to 289.

Overhead is not linear where transaction results have multiple rows. Some test data using JDBC on the 400MHz Pentium II showed it took 41ms to return all values from each of the 14 rows and 30ms to return a single value from each of the 14 rows of the emp table.

The current performance is already viable

With a powerful CPU or preferably multiple CPU mainstream RDBMS we may already be able to implement a non-demanding real-time database application. As was shown in the last section where with a 4 processor 550MHz server we were able to achieve 37000 selects/sec and 10330 updates/sec. Such rates are more than adequate for many applications especially if the vendors deliver on their promise of lifting the JDBC performance to PLSQL levels. Here is the explanation supporting the above assertion.

We observed a 2 processor 550MHz Xeon Server was running at less than 10% when performing its rated load. The tasks it was doing included communication with a IO gateway using CORBA, tracking, driving gui trends and logging over 350 points of strip mill data, out of which 150 were at a fast rate of 50ms scan and computing quality data and verdicts. The real time database benchmark in section 3.4.1 showed around 80000 read/write calls/sec. If we assume that half of the cpu usage occurred in making calls to the real-time database, then the required maximum read or maximum write rate is 80000*0.5*0.10=4000 calls/sec.

The viability of using mainstream RDBMS instead of a real– time database can be further improved by reading and writing the entire class instead of doing single accesses of a column from some row. This is borne out by the observation that the performance only declines by around 30% when all 10 columns from a table with 14 rows are returned instead of just one column from each row.

Finally more power is under way 1000MHz processors on 8 cpu SMP representing a 8 fold increase in computing power over the 4processor 550 MHz 7000M10 server benchmarked has been announced. If historic trends are any guide the price for this is expected to fall to the current 4 processor server prices within an year. So the price performance will make up soon for any shortfall in transaction rate grunt when a pure object relational option is considered for more demanding real time applications.

3.4.3 Solid state disks as shared memory

Solid state disks are a viable and under used option at present. The solid state disks implement a regular external SCSI interface. Internally they consist of arrays of battery-backed DRAMs. They offer access times of around 20microseconds and offer sustained throughputs 5MB/sec, 10MB/sec or 20MB/sec depending on the SCSI bus variant i.e. normal , fast or ultra SCSI. The 50000 read/writes per second is available through this technology using normal file access mechanisms. With the recent declines in memory costs per unit storage a solid state disk of 130MB was quoted as USD15000. Higher capacity upto several GB are available at higher costs.

Where the application processes are all on the same server the solid state disk is actually superior both in terms of access rate as well as data throughput when compared with CORBA or other middleware technologies. Where however the application processes are distributed the solid state disk does not outperform the current middleware technologies. The performance when using solid state disks as shared memory between distributed applications depends on the speed and performance of the network connection the distributed nodes.

We benchmarked CORBA and achieved a top rate of 2500 calls per second between applications on two different hosts over a LAN. The rate is dependent on the rate at which the TCPIP stack performs on the distributed nodes. The CORBA performance actually declined to 1000 calls /sec when both the communicating applications were on the same node. One exception to this is the case where the different applications are actually run under different threads inside a common JVM or executable, for this case the CORBA communications between different objects on different threads can be a blazing 1 millon calls/sec since the communication bypasses the protocol stack. Middlewares like CORBA incur the additional overhead of marshalling and unmarshalling the data at the source and destination. This is a significantly CPU intensive activity as was observed by profiling Java code that was used to benchmark the performance. The access of a solid state disk over a network does not incur any such overhead since the file is read from or written to directly at a lower level by the application.

In our opinion in applications where high speed and throughput both are critical solid state disks are a very effective option. Only dedicated memory resident real-time databases can approach the read/write performance offered by solid state disks. In mission critical failsafe application it may actually be more effective and efficient to store the shared data in solid state disks as compared to a memory resident dedicated real-time database. This is due to the fact that solid state disks are battery backed and the data is safe even if the server crashes on account of an operating system or hardware failure. Whereas the memory resident data would be lost if it were held in the server ram using a real-time database. Here are some of the urls for vendors of solid state disks

3.4.4 Shared memory using memory based file systems

Using a memory based file system it is possible to implement shared memory by storing the shared data structures in different files in the memory based file system. Typically such a disk based filing system emulates a hardware disk drive in memory. Application programs see the memory based file system as just another disk. With the recent decline in the cost of memory and the ever-increasing memory density, such memory based file systems are becoming an viable option. This can be an attractive low-cost alternative to a real-time database, in applications where large amounts of data have to be stored in high-speed bursts and the stored data has to be accessed by multiple processes at fast access rates.

A typical application which requires such features is a geo physical prospecting system where a blast is triggered by an explosive and the geo-phone data from a large number of sensors is acquired and stored at millisecond or faster scan rates for a few seconds. The analysis programs then access such data to compute results. Tens or even hundreds of MB of data may be generated in a few seconds and stored in the memory based file system. Then the analysis programs crunch this data by reading it at a fast rate to produce computed results on the spot.

Another typical application could be to act as an buffering cache to smooth out peaks in the data flow between a high speed data acquisition system and a conventional relational database.

Third party products are available which simulate a file system on disk. We tested a product called SuperDisk.

The following characteristics were noted

a) Application programs can access the data by normal file read write calls from application code in any language.

b) The effective rate of data transfer is high since the media is not a magnetic disk but actual memory. We achieved upto 25MB/sec

c) The relatively low number of file read and writes /second was limited by the number of IO related system calls the OS could handle. This meant

d) There was degradation in data throughput, as the data size became smaller. A 50-byte write gave 400 writes/sec while the rate fell to 91 writes/sec for a 250kb write.

The last of the above restricts the possibility of using a memory based file system in many applications. However in some applications it is possible to implement the shared memory as large data structures. These can be read into the applications local memory and manipulated and only the final results need to be written back to the memory based filing system.

Fig 3.11

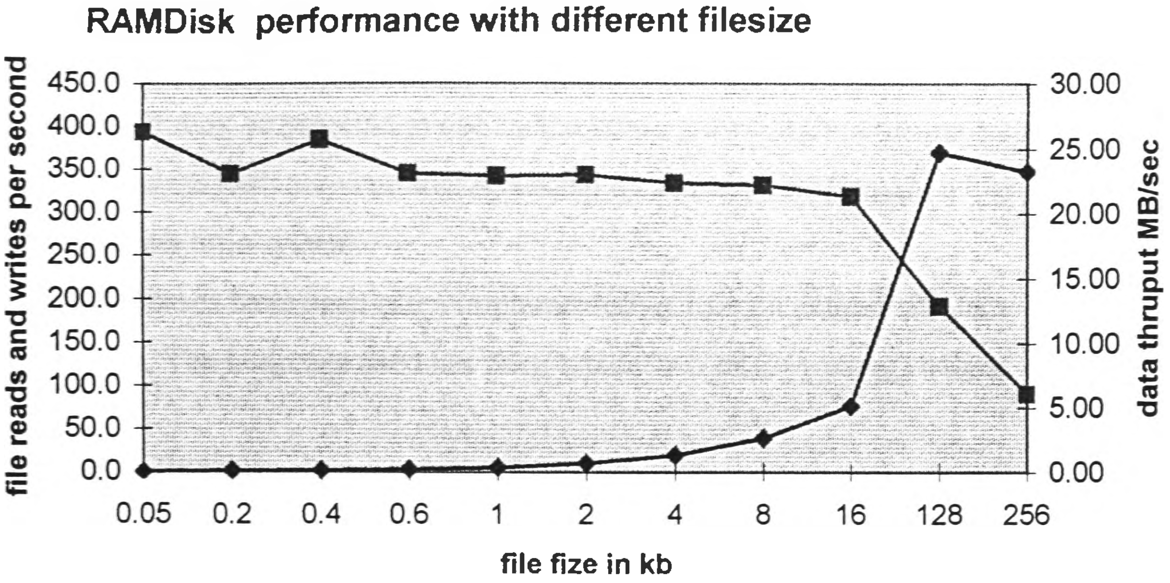


Table 3.10

File Size	Throughput	read and writes
Kb	Mb/sec	Rd/Wr /sec
0.05	0.02	393.7
0.2	0.07	344.8
0.4	0.15	384.6
0.6	0.21	346.0
1	0.34	343.6
2	0.69	344.8
4	1.34	335.6
8	2.67	333.3
16	5.13	320.5
128	24.71	193.1
256	23.27	90.9

3.4.5 Proposed design of a shared memory data server.

This section is a bit of the diversion from the main theme of this thesis. It is nevertheless presented here since it contains ideas how high performance shared memory can be implemented. It may be possible to use such a home grown high performance shared memory in tandem with conventional relational databases to implement a high performance real time system.
Real time databases have been often implemented by using a pattern similar to the one described here. The language used has been typically C or C++ using sockets.

A proposed design

With the advent of languages like Java which allow multithreading and socket communications to be done with relative ease, it is not difficult to implement a shared memory. Convenient building blocks including JNI, the java.net classes and threads are available to achieve an acceptable performance..

A common shared memory server process instantiates all the shared memory class instances using autogenerated initialise() function. The footprint of the shared data server process is fixed and does not grow since the class instances are static datamembers of the shared data server. A data file containing the stored data for the class instance is used to initialise the data members to the last saved values.

At startup the shared memory server instantiates the class instances as its static data members and loads the saved data and waits for application processes to connect.

The application processes start up and open a socket to the shared memory server. Which in turn spawns a new thread to service the incoming socket connects. This Thread will remain alive until the application process dies and communications can not be maintained.

If we autogenerate the SDSV class it is possible to have 5 combined read and 5 combined write functions. Each of which will return or accept as a parameter, a specific datatype. Each combined function will contain large case statements of several thousand cases. Each case will refer to a specific accessor method in a specific instance.

The SharedDataServer class may look like this:

```
Public class SDSV{


static TestClass1 testClass1 = new TestClass1();
static TestClass2 testClass2 = new TestClass2();
.
.
static TestClassN testClassN = new TestClassN();




//==============
// static methods
//==============

static float readFloats(int classMethodId )
{;}

static double readDoubles(int classMethodId )
{;}

static long readLongs(int classMethodId )
{;}

static short readShorts(int classMethodId )
{;}

static Date readDates(int classMethodId )
{;}
static String readStrings(int classMethodId )
{;}

static int readInts( int classMethodId )
{       int returnInt = 0;
        switch(classMethodId )
        {
```

```
        case 1:
            returnInt = testClass1.intValue;
            break;
        case 2:
            returnInt = testClass1.anotherIntValue;
            break;
        case 3:
            returnInt = testClass2.intValue;
            break;
        case 4:
            returnInt = testClass2.anotherIntValue;
            break;


        .

        .
        case 2N-1:
            returnInt = testClassN.intValue;
            break;
        case 2N:
            returnInt = testClassN. anotherIntValue;
            break;
        default
            throw new DataNotFoundException();
    }
    return returnInt;
}


static void writeFloats( int classMethodId, float value )
{
}

static void writeLongs( int classMethodId, long value )
{
}
static void writeShorts( int classMethodId, short value )
{
}

static void writeDoubles( int classMethodId, double value )
{
}

static void writeDates( int classMethodId, Date value )
{
}
```

```
static void writeStrings( int classMethodId, String value )
{
}



static void writeInts( int classMethodId, int value )
{
        switch(classMethodId )
        {
                case 1:
                        testClass1.intValue=value;
                        break;
                case 2:
                        testClass1.anotherIntValue=value;
                        break;
                case 3:
                        testClass2.intValue=value;
                        break;
                case 4:
                        testClass2.anotherIntValue=value;
                        break;


                .
                .
                case 2N-1:
                        testClassN.intValue=value;
                        break;
                case 2N:
                        testClassN. anotherIntValue=value;
                        break;
                default
                        throw new DataNotFoundException();
        }
}

}
// ------------------------------------------------------------
```

The application class Application1 may look like

```
Import  SDSV;
Public class Application1{
        Static void main( String args[] ) {
        }

        void someMethod()
        {
                int intValue =SDSV.readInts(
                            LU.testClass1_intValue );
                int intValue1 =SDSV.readInts(
                            LU.testClass1_anotherIntValue );
                float  floatValue = SDSV.Floats(
                            LU.testClass1_floatValue );
                SDSV.writeDates(
                            LU.testClass24_intValue, new Date() );
                SDSV.writeShorts(
                            LU.testClass15_shortValue, (short)100 );

        }

        void anotherMethod()
        {
                Date dateValue =SDSV.readDates(
                            LU.testClass51_dateValue );
                double doubleValue1 =SDSV.readDoubles(
                            LU.testClass12_anotherDoubleValue );
                SDSV.writeInts(
                            LU.testClass24_intValue, (int)100 );
                SDSV.writeFloats(
                            LU.testClass15_anotherIntValue, 100f );
        }
}

// ---------------------------------------------
```

The autogenerated lookup class may look like this

```
public class LU{

        public static int testClass1_intValue = 1;
        public static static int testClass1_anotherIntValue = 2;
        .
        public static int testClassN_intValue = 2N-1;
        public static int testClassN_anotherIntValue = 2N;

        public static int testClass1_floatValue = 1;
        public static int testClass1_anotherFloatValue = 2;
        .
        public static int testClassN_floatValue = 2N-1;
        public static int testClassN_anotherfloatValue = 2N;

        public static int testClass1_dateValue = 1;
        public static int testClass1_anotherDateValue = 2;
        .
        public static int testClassN_dateValue = 2N-1;
        public static int testClassN_anotherDateValue = 2N;

}
```

In this technique we are able to access the specific data member in a shared memory instance record using only 4 basic operations which are

1. Reading a data member from a static Look up class
2. Invoking a static method on SharedMemoryServer class
3. Do a case statement inside the static method
4. Read/assign values to or from a member of the shared memory class instance. Which in turn is a static data member of SDSV.

While the 4 operation are likely to be very fast we need to pass the parameters from the application to the SDSV process between steps 1 and 2 and after step 4 as well in case of a read. This can be done over Sockets or JNI. It is obvious JNI is going to be the fastest of these methods though sockets are easier to implement.

Multi threaded in one JVM

Alternatively if we run the shared memory server and all the applications on different threads of the same process then the execution will be really fast because there is no need to use an external transport between steps 1 and 2

Using RMI

Another alternative is to use RMI. Have the shared memory server process instantiate all the shared data class instances. Each class is to implement remotely invokable accessor methods. The application processes will invoke the accessor methods to read/write the shared data.

While the RMI approach will work the performance may need to be benchmarked to determine its viability where high access rates are required.

4.0 Conclusions

Section 4.0 Conclusions

The central conclusion of the research is – we may be able to use an object relational database to entirely replace a dedicated real-time database in a significant proportion of the real-time industrial automation applications.

Real-time applications which can be implemented using an object relational database (green indicate suitable **)**

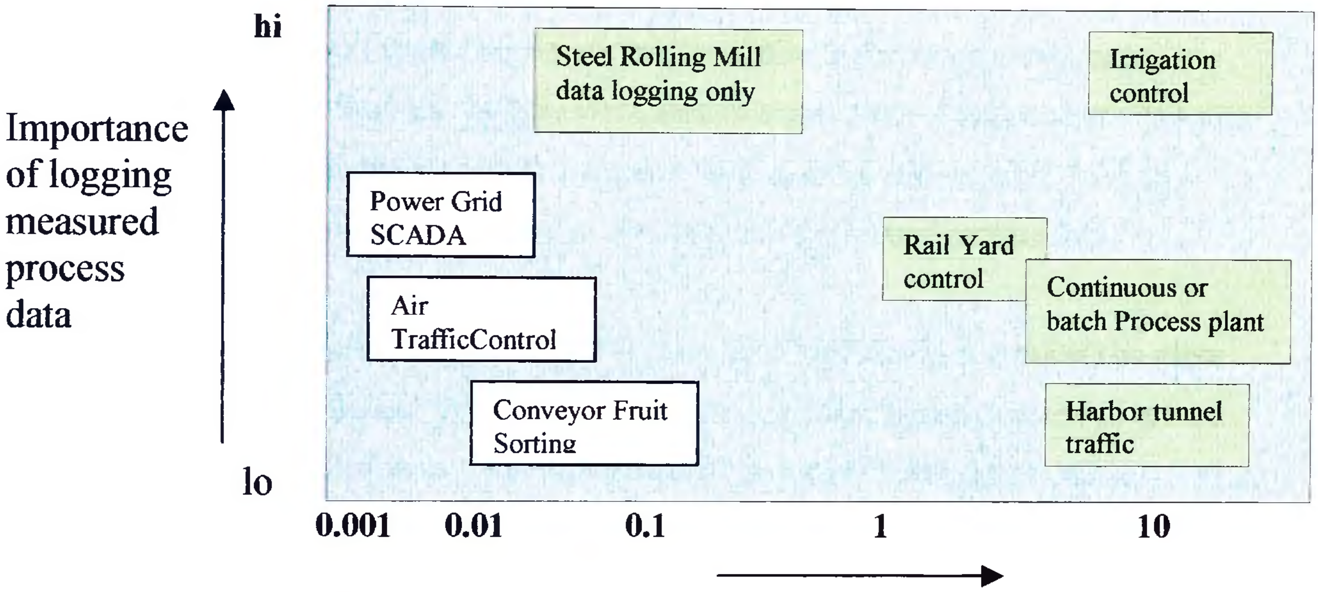


Fig 4.1

Many applications impose such stringent deterministic response requirements that they can only be met by a dedicated real-time database. However with the increasing processing power of fast multi CPU SMP servers these applications are getting fewer every day. Increasingly more and more applications which were earlier considered too demanding will be addressed using a mainstream object relational database.

The ever growing requirement of more accurate and higher resolution process data storage is being driven by the quality standards being followed by the manufacturers, who in turn are responding to the quality assurance demanded by their customers. Such historical data warehousing with user friendly desktop browser driven retrieval, is a another important requirement well addressed by mainstream object relational databases. We have accordingly presented the suitability matrix to test applications using their need for temporal determinism and importance of datalogging to help decide if they are suitable to be implemented using a mainstream object relational database.

Based on our research and analysis we found that a real-time database typically provides a more than 10 distinct features which are very useful in real time applications in automation. All these features could be replicated in a object relational database using its native constructs and features. When compared with a actual real-time database all features except "shared memory between applications" performed comparably.

The shared memory performance was slower than an actual real-time database by more than 10 times for single read writes. This gap was closed significantly when full record block reads and writes of large objects with several thousand fields. The relative performance of the object relational database was better for block reads and writes because the overhead associated with the read or write did not increase in proportion with the number of fields in the record. Whereas in case of a record with several thousand individual fields a block read or write would bring down the call rate to ~ 100000/number of fields in the record calls /second. This when calculated for a record with 4000 individual fields translates to 25 block read/writes /second which is worse than what can be achieved in an object relational database. So by structuring the applications to do most of the shared memory access as block reads we may be able to achieve the required shared memory performance. Naturally this can only be done where the application allows the use of this pattern. We were able to research the performance of third party hardware and software disk-caching tools which are aimed to speed object-relational databases by attacking their current disk i/o bound nature. The benchmark figures gathered by us indicate improvements in the order of 2 to 20 times are possible. In fact similar 10 times faster results are already offered by newer and non mainstream memory resident relational databases like TimesTen. We have largely ignored them because they do not qualify as mainstream and are considered too risky to be used in industrial automation where the software lifecycles are typically 7-10 years after commissioning.

As expected the historical data storage and archival features required in current real-time applications are easier and more scalable robust and easier to implement in the object relational database. Due to their commercial origin the object relational database offered much better error recovery options and fault tolerance by supporting multiplexing of control files and journalized transactions to different target disks. Thus a object relational database can be configured to continue operating through a disk crash which would in most cases bring down or cripple a real-time database which is not normally designed for this.

As a small nevertheless very important diversion from the main investigation we looked at a mechanism for the online compression of the synchronously scanned measured plant data. The simple but effective technique developed here was able to preserve the integrity of the data yet achieve a compression ratio of 1:10 when storing the time-stamped data to the object relational tables. We were able to use the fact that object relational databases are optimized to save storage when null values are written. Using the compression algorithm we were able to replace all the non trend significant data readings with nulls prior to storage. We saved additional storage by storing all the data for the scan against a common time-stamp, by noting the fact that the data gathering device scanned all its inputs in the same scan. The compression of data is vital in large industrial systems with hundreds and thousands of sensors. If such large volume data were not compressed well prior to storage, the capacity requirements of the historical data storage systems would soon exceed the current commercially available - competitively priced storage solutions. While significantly larger storage solutions are available the cost of acquiring, maintaining and backing up these rises astronomically making them almost impossible.

5.0 References

5.0 References

1. K. Ramamritham, Real-Time Databases, (invited paper) *International Journal of Distributed and Parallel Databases* 1 (1993), pp. 199-226, 1993.
2. TimesTen - http://www.timesten.com/
3. Angara Database Svstems - http://www.angara.com/
4. http://www.bitmicro.com/ProductLine.htm
5. http://www.soliddata.com/technology/white.html
6. M. Xiong, R. Sivasankaran, K. Ramamritham, J.A. Stankovic and D. Towsley, Scheduling Access to Temporal Data in Real-Time Databases, *Real-Time Database Systems: Issues and Applications*, Sang H. Son, Kwei-Jay Lin and Azer Bestavros ed., Kluwer Academic Publishers, pp. 167-191, 1997.
7. R.F.M. Aranha, V. Ganti, S. Narayanan, C.R. Muthukrishnan, S.T.S. Prasad and K. Ramamritham, Implementation of a Real-Time Database Svstem, *Information Systems*, Volume 21, Number 1, March 1996.

5.1 Bibiliography

1. K. Ramamritham, Real-Time Databases, (invited paper) *International Journal of Distributed and Parallel Databases* 1 (1993), pp. 199-226, 1993.
2. A. Burns, D. Prasad, A. Bondavalli, F. Di. Giandomenico, K. Ramamritham,, J. A. Stankovic, and L. Strigini, The Meaning and Role of Value in Scheduling Flexible Real-time Systems (to appear in) *Journal of Systems Architecture*
3. W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J.A. Stankovic, G. Wallace and C. Weems: The Spring Scheduling Co-Processor: A Scheduling Accelerator, (to appear in) *IEEE Trans. on VLSI Systems.*
4. J. M. Adan, M. F. Magalhaes and K. Ramamritham : Developing Predictable and Flexible Distributed Real-time systems, *Control Engineering Practice*, Vol 6, 1998, pp. 67-81.
5. G. Manimaran, S. R. Murthy and K. Ramamritham, A New Algorithm for Dynamic Scheduling of Parallelizable Tasks in Real-Time Multiprocessor Systems, *Real-Time Systems Journal* Vol. 15, 1998, pp.39-60.
6. M. Xiong, R. Sivasankaran, K. Ramamritham, J.A. Stankovic and D. Towsley, Scheduling Access to Temporal Data in Real-Time Databases, *Real-Time Database Systems: Issues and Applications*, Sang H. Son, Kwei-Jay Lin and Azer Bestavros ed., Kluwer Academic Publishers, pp. 167-191, 1997.
7. G. Manimaran, S. R. Murthy and K. Ramamritham : New Algorithms for Resource Reclaiming from Precedence Constrained Tasks in Multiprocessor Real-time Systems, *Journal of Parallel and Distributed Computing*, vol.44, no.2, Aug. 1997, pp.123-132.
8. K. Ramamritham: Predictability: Demonstrating Timing Requirements, *ACM Computing Surveys*, 28A(4), December 1996.
9. K. Ramamritham: Where Do Deadlines Come from and Where Do They Go?, invited paper, *Journal Of Database Management*, Vol 7, No. 2, pp. 4-10, Spring 1996.
10. K. Ramamritham: Allocation and Scheduling of Precedence-Related Periodic Tasks, *IEEE Transactions on Parallel and Distributed Systems*, Vol 6, No 4, April 1995, pp. 412-420.
11. R.F.M. Aranha, V. Ganti, S. Narayanan, C.R. Muthukrishnan, S.T.S. Prasad and K. Ramamritham, Implementation of a Real-Time Database System, *Information Systems*, Volume 21, Number 1, March 1996.
12. F.Wang, K. Ramamritham and J. A. Stankovic: Determining Redundancy Levels for Fault Tolerant Real-Time Systems, Special Issue of *IEEE Transactions on Computers* on Fault Tolerant Computing, Vol. 44, No. 2, February 1995, pp. 292-301.
13. K.Ramamritham and J.A. Stankovic: Scheduling Algorithms and Operating Systems Support for Real-Time Systems, invited paper, *Proceedings of the IEEE*, Jan 1994, pp. 55-67.

14. Chia Shen, Oscar Gonzalez, Krithi Ramamritham and Ichiro Mizunuma: User Level Scheduling of Communicating Real-Time Tasks, in Proceedings of the *Fifth IEEE Real-Time Technology and Applications*, Vancouver, Canada, June 1999.
15. Jesus Fernandez, Krithi Ramamritham, Adaptive Dissemination of Data in Real-Time Asymmetric Communication Environments, to appear in *EuroMicro Conference on Real-Time Systems*, June 1998.
16. Krithi Ramamritham, Chia Shen, Oscar Gonzalez, Shubo Sen, and Shreedhar B Shirgurkar: Using Windows NT for Real-Time Applications: Experimental Observations and

Recommendations, in Proceedings of the *Fourth IEEE Real-Time Technology and Applications*, Denver, CO, June 1998.

17. Oscar Gonzalez, H. Shrikumar, John A. Stankovic and Krithi Ramamritham: Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-time Scheduling, Proceedings of *the 18th IEEE Real-Time Systems Symposium*, San Francisco, California, December 1997.
18. Oscar Gonzalez, Chia Shen, Ichiro Mizunuma and Morikazu Takegaki: Implementation and Performance of MidART, Proceedings of *IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, San Francisco, California, December 1997.
19. Ping Xuan, Subhabrata Sen, Oscar Gonzalez, Jesus Fernandez and Krithi Ramamritham: Broadcast on Demand: Efficient and Timely Dissemination of Data in Mobile Environments, Proceedings of the *Third IEEE Real-Time Technology and Applications*, Montreal, Canada, June 1997.
20. Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham: Integrated Scheduling of Multimedia and Hard Real-Time Tasks, Proceedings of the *17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

21. Marty Humphrey and John A. Stankovic: CAISARTS: A Tool for Real-Time Scheduling Assistance, Proceedings of the *Second IEEE Real-Time Technology and Applications*, Brookline, MA, June, 1996.

22. M. Xiong, K. Ramamritham, Specification and Analysis of Transactions in Real-Time Active Databases, *Real-Time Database and Information Systems: Research Advances*, Azer Bestavros and Victor Fay-Wolfe ed., Kluwer Academic Publishers, pp. 327-354, 1997.

23. S. Sen, O. Gonzalez, K. Ramamritham, J.A. Stankovic, C. Shen, and M. Takegaki, Multimedia Capabilities in Distributed Real-Time Applications, in *Real-Time Database Systems: Issues and Applications*, Sang H. Son, Kwei-Jay Lin and Azer Bestavros ed., Kluwer Academic Publishers, 1997.
24. J. Shanmugasundaram, A. Nithrakashyap, J. Padhye, R. Sivasankaran, M. Xiong and K. Ramamritham, Transaction Processing in Broadcast Disk Environments, in *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg ed., Kluwer Academic Publishers, 1997, pp. 321-338.
25. P. O'Neil, K. Ramamritham, C. Pu, A Two-Phase Approach to Predictably Scheduling Real-Time Transactions, Chapter 18, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, V. Kumar (ed.), Prentice-Hall, Sep 1995, pp. 494-522.
26. B. Purimetla, R. Sivasankaran, K. Ramamritham and J. A. Stankovic, Real-Time Databases: Issues and Applications, in *Principles of Real-Time Systems*, Sang Son, Ed. Prentice-Hall, 1994, pp. 487-507.
27. J.A. Stankovic, K. Ramamritham, D. Towsley, Scheduling in Real-Time Transaction Systems, *Chapter in Foundations of Real-Time Computing: Scheduling and Resource Allocation*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 157-84, 1991.

28. K. Ramamritham, Where do Time Constraints Come From and Where Do They Go? International Journal of Database Management (invited paper), Vol. 7, No. 2, Spring 1996, pp. 4-10.
29. R. Sivasankaran, J.A. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham, Priority Assignment in Real-Time Active Databases, *VLDB Journal*, Vol. 5, No. 1, pp. 19-34, 1996.
30. B. Hamidzadeh, Y. Atif, K. Ramamritham, To Schedule or to Execute: Decision Support and Performance Implications, (to appear in) *Real-Time Systems*
31. J. Huang, J.A. Stankovic, K. Ramamritham, D. Towsley, B. Purimetla, On Using Priority Inheritance in Real-Time Databases, *Special Issue of Real-Time Systems Journal on Real-Time Databases*, Vol. 4, No. 3, September, 1992, pp. 243-68.
32. Chen, J.A. Stankovic, J. Kurose, D. Towsley, Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems, *Real-Time Systems Journal*, Vol. 3, #3, 1991, pp. 307-336.
33. Ming Xiong, Krithi Ramamritham, Deriving Deadlines and Periods for Update Transactions in Real-Time Databases, to appear in *the 20th IEEE Real-Time Systems Symposium (RTSS'99)* , Phoenix, Arizona, December, 1999.
34. Ming Xiong, Krithi Ramamritham, Jayant Haritsa, John A. Stankovic, MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases, to appear in *the 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, Vancouver, Canada, 1999.
35. R. Gupta, J. Haritsa, K. Ramamritham, More Optimism about Real-Time Distributed Commit Processing, *18th IEEE Real-Time Systems Symposium*, 1997, San Francisco, California.
36. M. Xiong, R. Sivasankaran, K. Ramamritham, J.A. Stankovic and D. Towsley, Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics, *17th IEEE Real-Time Systems Symposium*, 1996, Washington, DC.
37. R. Gupta, J. Haritsa, K. Ramamritham and Seshadri, Commit Processing in Distributed Real-Time Database Systems, *17th IEEE Real-Time Systems Symposium*, 1996, Washington, DC.
38. M. Xiong, K. Ramamritham, J.A. Stankovic D. Towsley and R. Sivasankaran, Maintaining Temporal Consistency: Issues and Algorithms, First International Workshop on Real-Time Databases, March 1996, Newport Beach, California.
39. R. Sivasankaran, K. Ramamritham, J.A. Stankovic and D. Towsley, Data Placement, Logging and Recovery in Real-Time Active Databases, *ARTDB-95*, Skovde, Sweden, 1995, Workshops in Computing, Mikael Berndtsson and Jorden Hansoon (Eds), pp. 226-41.
40. M. Kamath, K. Ramamritham and D. Towsley, Continuous Media Sharing in Multimedia Database Systems, *Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)*, Singapore, pp. 79-86.
41. K. Ramamritham, Time for Real-Time Temporal Databases? *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, June 1993,
42. R. Sivasankaran, Purimetla, J.A. Stankovic, K. Ramamritham, Network Services Database - A Distributed Active Real-Time Database(DARTDB) Application, *Proceedings First IEEE Workshop on Real-Time Applications*, May 1993.
43. Purimetla, R. Sivasankaran, J.A. Stankovic, K. Ramamritham, A Study of Distributed Real-Time Active Database Applications, *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1993.

6.0 Appendix.

6.1 Solid State Disk Performance Data

References from url 3.4.2

SolidData

-
- Provides ultra high-speed access to I/O-intensive Unix and NT data files
- Turbocharges existing RAID and JBOD storage systems
- Scales from 536 MB to 2.7 GB capacity per system
- Incorporates a non-volatile system architecture

Excellerator 600 is an affordable, entry-level solid state storage system that enables customers to achieve dramatic performance improvements in I/O intensive applications such as electronic commerce, Internet email and news, billing, messaging, and customer care. Excellerator 600 uses DRAM technology to turbocharge existing RAID and JBOD storage systems. It runs on all major Unix and NT platforms and incorporates an industry-standard SCSI interface.

Ultra High-Speed Performance for I/O-Intensive Applications

Excellerator 600 eliminates bottlenecks that are created by high-usage data files known as "hot files." These hot files usually comprise only 5% to 10% of an application's data, yet they constitute over 50% of the I/O activity. When the hot files are moved onto Excellerator 600, overall system performance is dramatically increased because DRAM technology eliminates the mechanical delays associated with magnetic disk drives. As a result, Excellerator 600 has a data transfer rate that is over 90% faster than traditional disk storage.

To further accelerate performance, Excellerator 600 incorporates a patented Direct Addressing™ technology. Direct Addressing translates SCSI addresses received from the host directly into DRAM array addresses. The translation uses high-speed dedicated circuits, eliminating intermediate microprocessor-to-insert latencies, overhead, and time delays. Accordingly, Excellerator 600 routinely outperforms the competition by a factor of two to three times in raw data access speed.

Solid, Non-Volatile System Architecture

Recognizing the inherent volatile nature of DRAM components, Excellerator 600 was designed with a patented Data Retention System™ including: integral battery backup, an on-board disk drive with a separate data path that is independent of the host CPU, automatic backup control logic, and RAIC (Redundant Array of Independent Chips). In the event of a power failure, the Data Retention System provides uninterrupted power to the DRAM array. After a prescribed time period (typically 8 minutes), Excellerator 600 will back up all data onto the internal disk drive, providing full data integrity. The DRAM array has been architected with built-in parity protection so that it can tolerate the failure of an entire memory chip without experiencing loss of data.

Plug-and-Play Installation

Because Excellerator 600 incorporates an industry-standard SCSI interface, it can be readily integrated into any Unix or NT server, just like another disk drive. Simply connect Excellerator 600 to an available SCSI bus, move the selected data files to the system, and it is ready to go. The process typically takes less than an hour and requires no special configuration of the operating system or database engine. And if you don't know which files to move, Solid Data has the applications expertise to assist you.

Field-Proven Reliability

Excellerator 600 uses high-reliability components and undergoes extensive burn-in and testing before shipment. Demonstrated field MTBF is greater than two million hours, translating to an annual system availability in excess of 99.999%. All Solid Data products include a full one-year warranty supported by Hewlett-Packard's On-Site Next-Day Service Program.

Excellerator 600 Technical Specifications

High Performance

- Delivers 50% to 200% improvement to target applications
- 18 microsecond data access time

Compatibility

Plug-and-play installation

- SCSI-2 Fast and Wide interface
- Compatible across all Unix and NT server platforms (with no device driver installation), including:
  Hewlett-Packard
  Silicon Graphics
  Sun
  IBM
  Compaq
  Dell

Supported relational databases:

Sybase SQL Server
Oracle
Informix
Most proprietary RDBMS

Scalability

- 536 MB to 2.7 GB, based on number and capacity of array boards
- System self-configures to number of array boards installed
- Up to 15 systems can be configured on a single SCSI Wide bus

Reliability

- Field-measured mean-time-between-failure (MTBF) is over 2 million hours

Battery backup specifications:

- Dual internal lead acid gel cell batteries
- Internal, full-time battery charging with automatic charge monitoring
- Automatic sensing of power outage conditions
- Battery operation of 1 to 2.5 hours
- Backup and restore data rate of 536 MB/minute

Internal backup disk drive

- Disk is powered down during normal system operation
- Automatic backup to disk during extended power failures
- Automatic restore when power resumes

Full-syndrome error correction coded (ECC) integrated circuits

- 8 bits per 72 dedicated for ECC
- Two bit error detection and single bit error correction

Regulatory approvals

- Underwriters Laboratory and C-UL listed
- FCC certified
- VDE/TUV certified to EN60950

Field diagnostics

- Host-resident I/O Test software provides detailed performance, reliability, and maintenance information
- Uses RS-232 output port
- Provides detailed diagnostic and repair information

Excellerator 600 Solid State Storage System	
System Capacity	
Minimum and Maximum capacity	536 MB 2.68 GB

bitmicro

E-Disk^{TM} SCSI Narrow Product Matrix

E-DISK MODEL	SCSI NARROW 8-bit INTERFACE	FORM FACTO R	BURST R/W RATES	SUSTAINED R/W RATES	ACCESS TIMES	STORAGE CAPACITY
SNX25	Normal SCSI SE	2.5"	5 MB/sec	>4.5MB/sec	<0.25 ms	128MB-2GB
SFX25	Fast SCSI SE	2.5"	10 MB/sec	>9 MB/sec	<0.2 ms	128MB-2GB
SNX35	Normal SCSI SE	3.5"	5 MB/sec	>4.5MB/sec	<0.25 ms	128MB-18.5GB
SFX35	Fast SCSI SE	3.5"	10 MB/sec	>9 MB/sec	<0.2 ms	128MB-18.5GB
SUX35	Ultra SCSI SE	3.5"	20 MB/sec	>18 MB/sec	<0.1 ms	256MB-18.5GB

E-Disk^{TM} SCSI Wide Product Matrix

E-DISK MODEL	SCSI WIDE 16-bit INTERFACE	FORM FACTO R	BURST R/W RATES	SUSTAINED R/W RATES	ACCESS TIMES	STORAGE CAPACITY
SFW35	Fast Wide SCSI SE	3.5"	20 MB/sec	>18 MB/sec	<0.1 ms	256MB-18.5GB
SUW35	Ultra Wide SCSI SE	3.5"	40 MB/sec	>34 MB/sec	<0.049 ms	256MB-18.5GB
SUD35	Ultra Wide SCSI LVD	3.5"	40 MB/sec	>34 MB/sec	<0.049 ms	256MB-18.5GB
SCW35	SCA Ultra Wide SCSI SE	3.5"	40 MB/sec	>34 MB/sec	<0.049 ms	256MB-18.5GB
SCD35	SCA Ultra Wide SCSI LVD	3.5"	40 MB/sec	>34 MB/sec	<0.049 ms	256MB-18.5GB

E-Disk^{TM} ATA/IDE Product Matrix

E-DISK MODEL	IDE INTERFACE	FORM FACTO	BURST R/W RATES	SUSTAINED R/W RATES	ACCESS TIMES	STORAGE CAPACITY

		R				
ATI25	IDE	2.5"	5 MB/sec	>4.5MB/sec	<0.25 ms	128MB-2GB
ATE25	EIDE	2.5"	10 MB/sec	>9 MB/sec	<0.2 ms	128MB-2GB
ATI35	IDE	3.5"	5 MB/sec	>4.5MB/sec	<0.25 ms	128MB-18.5GB
ATE35	EIDE	3.5"	10 MB/sec	>9 MB/sec	<0.2 ms	128MB-18.5GB
ATX35	Ultra DMA	3.5"	16.6 MB/sec	>14 MB/sec	<0.1 ms	256MB-18.5GB
ATU35	Ultra DMA/33	3.5"	33.3 MB/sec	>30 MB/sec	<0.05 ms	256MB-18.5GB

6.2 Source for Java Triggers that generate automatic updates to a GUI in a browser when a table gets updated.

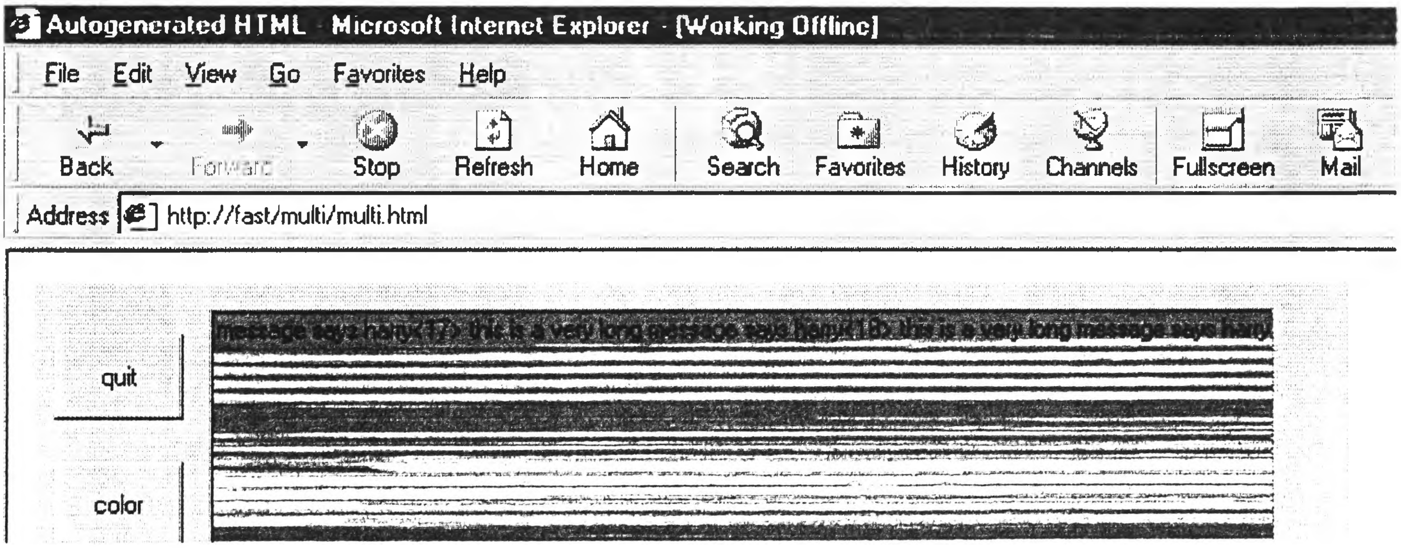
Java Source for Stored Procedure:

The J developer offers a built menu option to publish the class to a stored procedure though this can be done even from the command line using the loadjava utility.

```
//==========================================================================
//Title: COSNotifierStoredProc.java
//Version:
//Copyright:  Copyright (c) 1999
//Author:     Saugato Mukerji
//Company:    Saugato Mukerji
//Description: The source for the java stored procedure that provides a method that
//              accepts a String as a parameter and multicasts the string on the
//              Address 224.5.6.7
//==========================================================================
package COSUPDATE;

import java.net.*;
import java.io.IOException;

public class COSNotifierStoredProc {
  // static data members for cennect information
  static MulticastSocket multicastSocket;
  static InetAddress group;
  static long messageCount=0;

  public COSNotifierStoredProc() {
  }

  public static void main(String[] args) {
   new COSNotifierStoredProc();
  }

  public static void send( String messageString )
  {
        try {
   if( multicastSocket == null )
   {
                group = InetAddress.getByName("224.5.6.7");
                multicastSocket = new MulticastSocket(6789);
                multicastSocket.joinGroup(group);
   }
   messageCount++;
                                // increment message count
  messageString = "<"+messageCount+"> "+messageString;
            int messageLength = messageString.length();
  byte [] msgBuffer = new byte[messageLength+10];

            for( int i=0; i<messageLength; i++)
            {
                  msgBuffer[i] = ( byte )messageString.charAt( i );
            }


  DatagramPacket datagramPacket =
   new DatagramPacket( msgBuffer, messageLength,
             group, 6789);
              multicastSocket.send(datagramPacket);

      } catch (UnknownHostException e) {
        System.err.println("Host not found: " + e);
      }  catch (IOException e) {
            System.err.println("IO exception: " + e);
 }
}
}
```

Testing the COSUPDATE.send() stored procedure from SQLPLUS prompt to update GUI Applet

```
D:\>Sqlplus scott/tiger

SQL> call COSUPDATE.send('this is a very long message says harry');

Call completed.
SQL> /

Call completed.
SQL> /

Call completed.
SQL>
.
.
.
SQL> /

Call completed.
SQL> /

Call completed.
```



Autogenerated HTML - Microsoft Internet Explorer - [Working Offline]

Address: http://fast/multi/multi.html

message says harry<17> this is a very long message says harry<18> this is a very long message says harry

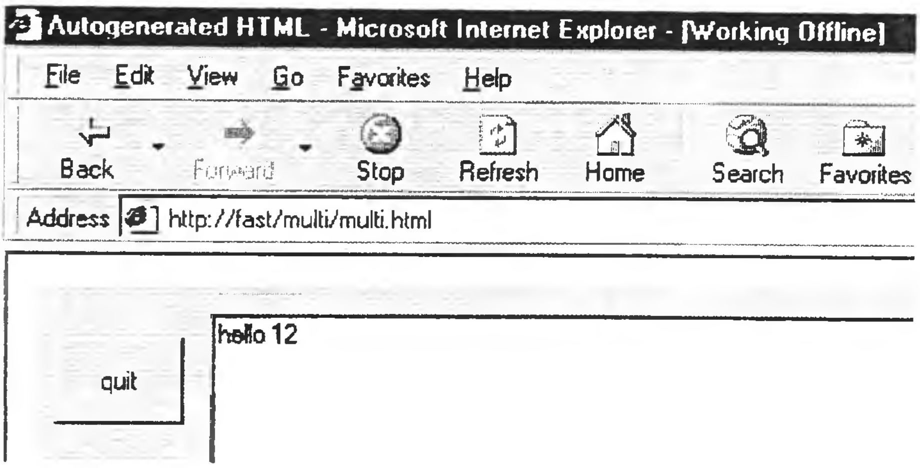
Fig 2.6.1

Causing Trigger Action to invoke the stored procedure by updating a field in the emp table with SQLPLUS. The changed table value is transmitted to the GUI Applet by the java stored procedure

```
SQL> CREATE OR REPLACE TRIGGER gui_regresh_trigger
  2  AFTER UPDATE OF stringValue ON TestClass
  3  FOR EACH ROW
  4  WHEN (new.stringValue <> old.stringValue)
  5  CALL COSUPDATE.send(:new.stringValue)
  6  /

Trigger created.
```

Inserting a row, look at fig 2.6.2 to verify the data reached the Applet .

```
SQL> update testclass set stringValue='hello 12';

1 row updated.

SQL>
```



http://fast/multi/multi.html

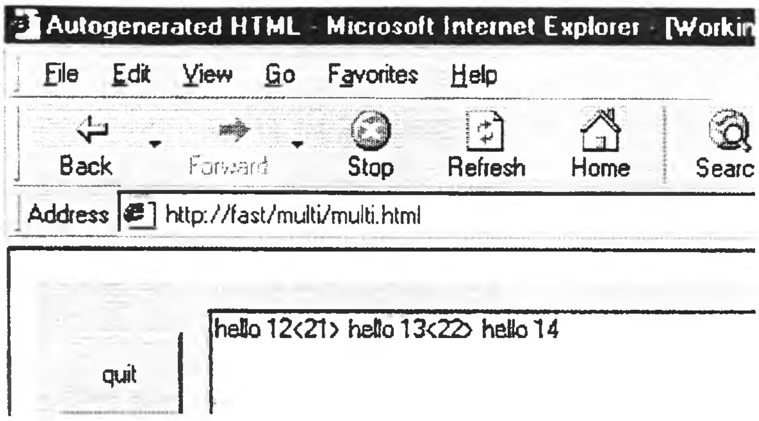
hello 12

Fig 2.6.2

```
SQL> update testclass set stringValue='hello 13';

1 row updated
```

```
SQL> update testclass set stringValue='hello 14';

1 row updated.
```

Updated the row twice, look at fig 2.6.3 to verify the data reached the Applet .


Autogenerated HTML - Microsoft Internet Explorer - [Workin
File Edit View Go Favorites Help
Back
Forward
Stop
Refresh
Home
Searc
Address
quit

http://fast/multi/multi.html

hello 12<21> hello 13<22> hello 14

Fig 2.6.3

```
SQL> /
1 row updated.
```

Note trigger did nor activate since same value was updated in the table

```
SQL> /
1 row updated.
```

Note trigger did nor activate since same value was updated in the table

```
SQL> /
1 row updated.
```

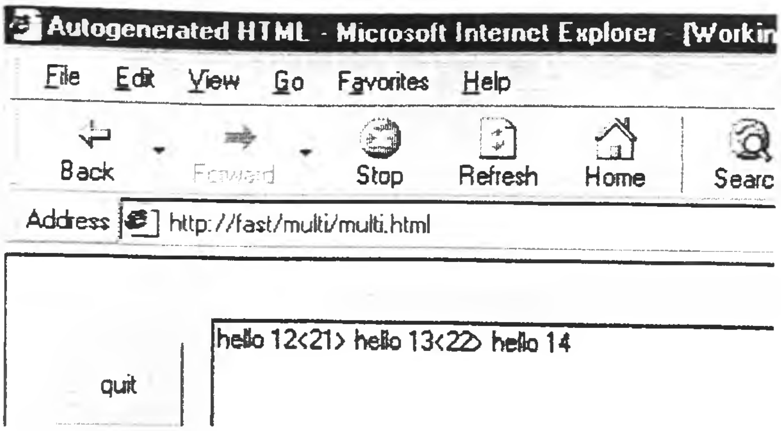
Note trigger did nor activate since same value was updated in the table

```
SQL> /
1 row updated.
```

Note trigger did nor activate since same value was updated in the table

```
SQL> /
1 row updated.
```

Updated the row 5 times but value remained same so no Change of State caused, look at fig 2.6.4 to verify that data was not sent to the Applet .



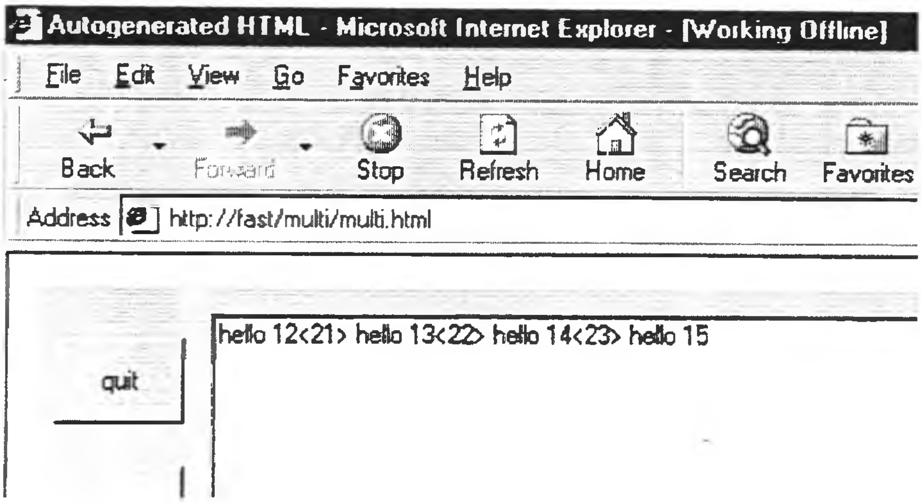
Address http://fast/multi/multi.html

```
hello 12<21> hello 13<22> hello 14
```

Fig 2.6.4

Updated the row, look at fig 2.6.5 to verify the changed data “hello 15” reached the Applet .

```
SQL> update testclass set stringValue='hello 15';

1 row updated.

SQL>
```



Address http://fast/multi/multi.html

```
hello 12<21> hello 13<22> hello 14<23> hello 15
```

6.3 Object Relational Table read write benchmark code in plsql.

```
SQL> @nlsnolog
SP2-0310: unable to open file "nlsnolog.sql"
SQL> @nlunolog

Table altered.

start_time=42210 end_time=42218 loopcount=10000 aSal=10800

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlunolog

Table altered.

start_time=42227 end_time=42233 loopcount=10000 aSal=20800

========================================================
update emp nologging on troys
========================================================

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlunolog

Table altered.

start_time=42241 end_time=42247 loopcount=10000 aSal=30800

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlunolog

Table altered.

start_time=42255 end_time=42260 loopcount=10000 aSal=40800

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlunolog

Table altered.

start_time=42269 end_time=42275 loopcount=10000 aSal=50800

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlunolog

Table altered.

start_time=42286 end_time=42291 loopcount=10000 aSal=60800

PL/SQL procedure successfully completed.

Commit complete.
```

```
SQL>


========================================================
update emp logging enabled on troys
========================================================

SQL> @nlulog

Table altered.

Logging on start_time=42599 end_time=42606 loopcount=10000 aSal=70800

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlulog

Table altered.

Logging on start_time=42613 end_time=42619 loopcount=10000 aSal=80800

PL/SQL procedure successfully completed.

Commit complete.
SQL> @nlulog

Table altered.

Logging on start_time=42625 end_time=42630 loopcount=10000 aSal=90800

PL/SQL procedure successfully completed.



========================================================
select emp on troys
========================================================

Connected to:
Oracle8 Enterprise Edition Release 8.1.4.0.0 - Beta
With the Partitioning and Objects options
PL/SQL Release 8.1.4.0.0 - Production

start_time=43049 end_time=43076 loopcount=100000 aname=SMITH

PL/SQL procedure successfully completed.

SQL> @nls
start_time=43110 end_time=43137 loopcount=100000 aname=SMITH

PL/SQL procedure successfully completed.

SQL> /
start_time=43143 end_time=43169 loopcount=100000 aname=SMITH

PL/SQL procedure successfully completed.

SQL> /
start_time=43173 end_time=43199 loopcount=100000 aname=SMITH
```

File: D:\masters\RC1\Section 6.0 Appendix\plsql benchmark code\nlu.sql 7/28/99, 7:15:01PM

```
--============================================================
-- Name nlu.sql
-- Description a test to determine the update performance
--
--============================================================

set serveroutput on
alter table emp NOLOGGING;
set autocommit on

DECLARE
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname   CHAR(30);
    aSal    NUMBER;
BEGIN

    ITEM_COUNT := 10000;
    select sal into aSal from emp WHERE EMPNO = 7369;
    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
        aSal := aSal + 1;
        UPDATE EMP SET sal = aSal WHERE EMPNO = 7369;
    END LOOP;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    select sal into aSal from emp WHERE EMPNO = 7369;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                          ' loopcount='||ITEM_COUNT||' aSal='||aSal );
END;

/
```

**File: D:\masters\RC1\Section 6.0 Appendix\plsql benchmark code\nls.sql 7/28/99, 7:14:01P
M**

```
--===============================================================
-- Name nls.sql
-- Description a test to determine the select performance
--
--===============================================================

set serveroutput on


DECLARE
    STATUS NUMBER(10);
    MESSAGE VARCHAR2(80);
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname   CHAR(30);
    X   NUMBER;
BEGIN

    ITEM_COUNT := 10000;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
            SELECT ENAME INTO aname FROM EMP WHERE EMPNO = 7369;
    END LOOP;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                          ' loopcount='||ITEM_COUNT||' aname='||aname );
END;

/
```

```
--==========================================================================
-- Copyright Saugato Mukerji
--==========================================================================
-- SYSTEM:        Thesis
-- SUBSYSTEM:     Sparse Data
-- FILE NAME:     NumberTable.sql
-- DESCRIPTION:   generates the table with 300 numbers
--                used to test with sparse data resulting from compression
-- AUTHOR:        Saugato Mukerji
-- CREATED:       10-05-99
-- USAGE:
-- INPUTS:
-- OUTPUTS:
-- NOTES:
--
-- REVISION HISTORY:
-- $Author$
-- $Id$
--
-- $Log$
--
--==========================================================================

--PROMPT ******************************************
--PROMPT * executing NUMBERTABLE.sql script
--PROMPT ******************************************

DROP TABLE NUMBERTABLE;

CREATE TABLE numbertable
(
    TIME_STAMP  DATE,
    PATTERN_ID VARCHAR2(6),
    SPARE_B_1 NUMBER,
    SPARE_B_2 NUMBER,
    SPARE_B_3 NUMBER,
    SPARE_B_4 NUMBER,
    SPARE_B_5 NUMBER,
    SPARE_B_6 NUMBER,
    SPARE_B_7 NUMBER,
    SPARE_B_8 NUMBER,
    SPARE_B_9 NUMBER,
    SPARE_B_10 NUMBER,
    SPARE_B_11 NUMBER,
    SPARE_B_12 NUMBER,
    SPARE_B_13 NUMBER,
    SPARE_B_14 NUMBER,
    SPARE_B_15 NUMBER,
    SPARE_B_16 NUMBER,
    SPARE_B_17 NUMBER,
    SPARE_B_18 NUMBER,
    SPARE_B_19 NUMBER,
    SPARE_B_20 NUMBER,
    SPARE_B_21 NUMBER,
    SPARE_B_22 NUMBER,
    SPARE_B_23 NUMBER,
    SPARE_B_24 NUMBER,
    SPARE_A_1 NUMBER,
    SPARE_A_2 NUMBER,
        SPARE_A_3 NUMBER,
        SPARE_A_4 NUMBER,
        SPARE_A_5 NUMBER,
        SPARE_A_6 NUMBER,
        SPARE_A_7 NUMBER,
        SPARE_A_8 NUMBER,
        SPARE_A_9 NUMBER,
        SPARE_A_10 NUMBER,
        SPARE_A_11 NUMBER,
        SPARE_A_12 NUMBER,
        SPARE_A_13 NUMBER,
        SPARE_A_14 NUMBER,
        SPARE_A_15 NUMBER,
        SPARE_A_16 NUMBER,
        SPARE_A_17 NUMBER,
        SPARE_A_18 NUMBER,
        SPARE_A_19 NUMBER,
        SPARE_A_20 NUMBER,
        SPARE_A_21 NUMBER,
        SPARE_A_22 NUMBER,
        SPARE_A_23 NUMBER,
        SPARE_A_24 NUMBER,
        SPARE_c_1 NUMBER,
        SPARE_c_2 NUMBER,
        SPARE_c_3 NUMBER,
        SPARE_c_4 NUMBER,
        SPARE_c_5 NUMBER,
        SPARE_c_6 NUMBER,
        SPARE_c_7 NUMBER,
        SPARE_c_8 NUMBER,
        SPARE_c_9 NUMBER,
        SPARE_c_10 NUMBER,
        SPARE_c_11 NUMBER,
        SPARE_c_12 NUMBER,
        SPARE_c_13 NUMBER,
        SPARE_c_14 NUMBER,
        SPARE_c_15 NUMBER,
        SPARE_c_16 NUMBER,
        SPARE_c_17 NUMBER,
        SPARE_c_18 NUMBER,
        SPARE_c_19 NUMBER,
        SPARE_c_20 NUMBER,
        SPARE_c_21 NUMBER,
        SPARE_c_22 NUMBER,
        SPARE_c_23 NUMBER,
        SPARE_c_24 NUMBER,
        SPARE_D_1 NUMBER,
        SPARE_D_2 NUMBER,
        SPARE_D_3 NUMBER,
        SPARE_D_4 NUMBER,
        SPARE_D_5 NUMBER,
        SPARE_D_6 NUMBER,
        SPARE_D_7 NUMBER,
        SPARE_D_8 NUMBER,
        SPARE_D_9 NUMBER,
        SPARE_D_10 NUMBER,
        SPARE_D_11 NUMBER,
        SPARE_D_12 NUMBER,
        SPARE_D_13 NUMBER,
    SPARE_D_14 NUMBER,
    SPARE_D_15 NUMBER,
    SPARE_D_16 NUMBER,
    SPARE_D_17 NUMBER,
    SPARE_D_18 NUMBER,
    SPARE_D_19 NUMBER,
    SPARE_D_20 NUMBER,
    SPARE_D_21 NUMBER,
    SPARE_D_22 NUMBER,
    SPARE_D_23 NUMBER,
    SPARE_D_24 NUMBER,
    SPARE_E_1 NUMBER,
    SPARE_E_2 NUMBER,
    SPARE_E_3 NUMBER,
    SPARE_E_4 NUMBER,
    SPARE_E_5 NUMBER,
    SPARE_E_6 NUMBER,
    SPARE_E_7 NUMBER,
    SPARE_E_8 NUMBER,
    SPARE_E_9 NUMBER,
    SPARE_E_10 NUMBER,
    SPARE_E_11 NUMBER,
    SPARE_E_12 NUMBER,
    SPARE_E_13 NUMBER,
    SPARE_E_14 NUMBER,
    SPARE_E_15 NUMBER,
    SPARE_E_16 NUMBER,
    SPARE_E_17 NUMBER,
    SPARE_E_18 NUMBER,
    SPARE_E_19 NUMBER,
    SPARE_E_20 NUMBER,
    SPARE_E_21 NUMBER,
    SPARE_E_22 NUMBER,
    SPARE_E_23 NUMBER,
    SPARE_E_24 NUMBER,
    SPARE_F_1 NUMBER,
    SPARE_F_2 NUMBER,
    SPARE_F_3 NUMBER,
    SPARE_F_4 NUMBER,
    SPARE_F_5 NUMBER,
    SPARE_F_6 NUMBER,
    SPARE_F_7 NUMBER,
    SPARE_F_8 NUMBER,
    SPARE_F_9 NUMBER,
    SPARE_F_10 NUMBER,
    SPARE_F_11 NUMBER,
    SPARE_F_12 NUMBER,
    SPARE_F_13 NUMBER,
    SPARE_F_14 NUMBER,
    SPARE_F_15 NUMBER,
    SPARE_F_16 NUMBER,
    SPARE_F_17 NUMBER,
    SPARE_F_18 NUMBER,
    SPARE_F_19 NUMBER,
    SPARE_F_20 NUMBER,
    SPARE_F_21 NUMBER,
    SPARE_F_22 NUMBER,
    SPARE_F_23 NUMBER,
    SPARE_F_24 NUMBER,
    SPARE_G_1 NUMBER,
    SPARE_G_2 NUMBER,
    SPARE_G_3 NUMBER,
    SPARE_G_4 NUMBER,
    SPARE_G_5 NUMBER,
    SPARE_G_6 NUMBER,
    SPARE_G_7 NUMBER,
    SPARE_G_8 NUMBER,
    SPARE_G_9 NUMBER,
    SPARE_G_10 NUMBER,
    SPARE_G_11 NUMBER,
    SPARE_G_12 NUMBER,
    SPARE_G_13 NUMBER,
    SPARE_G_14 NUMBER,
    SPARE_G_15 NUMBER,
    SPARE_G_16 NUMBER,
    SPARE_G_17 NUMBER,
    SPARE_G_18 NUMBER,
    SPARE_G_19 NUMBER,
    SPARE_G_20 NUMBER,
    SPARE_G_21 NUMBER,
    SPARE_G_22 NUMBER,
    SPARE_G_23 NUMBER,
    SPARE_G_24 NUMBER,
    SPARE_H_1 NUMBER,
    SPARE_H_2 NUMBER,
    SPARE_H_3 NUMBER,
    SPARE_H_4 NUMBER,
    SPARE_H_5 NUMBER,
    SPARE_H_6 NUMBER,
    SPARE_H_7 NUMBER,
    SPARE_H_8 NUMBER,
    SPARE_H_9 NUMBER,
    SPARE_H_10 NUMBER,
    SPARE_H_11 NUMBER,
    SPARE_H_12 NUMBER,
    SPARE_H_13 NUMBER,
    SPARE_H_14 NUMBER,
    SPARE_H_15 NUMBER,
    SPARE_H_16 NUMBER,
    SPARE_H_17 NUMBER,
    SPARE_H_18 NUMBER,
    SPARE_H_19 NUMBER,
    SPARE_H_20 NUMBER,
    SPARE_H_21 NUMBER,
    SPARE_H_22 NUMBER,
    SPARE_H_23 NUMBER,
    SPARE_H_24 NUMBER,
    SPARE_I_1 NUMBER,
    SPARE_I_2 NUMBER,
    SPARE_I_3 NUMBER,
    SPARE_I_4 NUMBER,
    SPARE_I_5 NUMBER,
    SPARE_I_6 NUMBER,
    SPARE_I_7 NUMBER,
    SPARE_I_8 NUMBER,
    SPARE_I_9 NUMBER,
    SPARE_I_10 NUMBER,
    SPARE_I_11 NUMBER,
    SPARE_I_12 NUMBER,
    SPARE_I_13 NUMBER,
    SPARE_I_14 NUMBER,
    SPARE_I_15 NUMBER,
    SPARE_I_16 NUMBER,
    SPARE_I_17 NUMBER,
    SPARE_I_18 NUMBER,
    SPARE_I_19 NUMBER,
    SPARE_I_20 NUMBER,
    SPARE_I_21 NUMBER,
    SPARE_I_22 NUMBER,
    SPARE_I_23 NUMBER,
    SPARE_I_24 NUMBER,
    SPARE_J_1 NUMBER,
    SPARE_J_2 NUMBER,
    SPARE_J_3 NUMBER,
    SPARE_J_4 NUMBER,
    SPARE_J_5 NUMBER,
    SPARE_J_6 NUMBER,
    SPARE_J_7 NUMBER,
    SPARE_J_8 NUMBER,
    SPARE_J_9 NUMBER,
    SPARE_J_10 NUMBER,
    SPARE_J_11 NUMBER,
    SPARE_J_12 NUMBER,
    SPARE_J_13 NUMBER,
    SPARE_J_14 NUMBER,
    SPARE_J_15 NUMBER,
    SPARE_J_16 NUMBER,
    SPARE_J_17 NUMBER,
    SPARE_J_18 NUMBER,
    SPARE_J_19 NUMBER,
    SPARE_J_20 NUMBER,
    SPARE_J_21 NUMBER,
    SPARE_J_22 NUMBER,
    SPARE_J_23 NUMBER,
    SPARE_J_24 NUMBER,
    SPARE_K_1 NUMBER,
    SPARE_K_2 NUMBER,
    SPARE_K_3 NUMBER,
    SPARE_K_4 NUMBER,
    SPARE_K_5 NUMBER,
    SPARE_K_6 NUMBER,
    SPARE_K_7 NUMBER,
    SPARE_K_8 NUMBER,
    SPARE_K_9 NUMBER,
    SPARE_K_10 NUMBER,
    SPARE_K_11 NUMBER,
    SPARE_K_12 NUMBER,
    SPARE_K_13 NUMBER,
    SPARE_K_14 NUMBER,
    SPARE_K_15 NUMBER,
    SPARE_K_16 NUMBER,
    SPARE_K_17 NUMBER,
    SPARE_K_18 NUMBER,
    SPARE_K_19 NUMBER,
    SPARE_K_20 NUMBER,
    SPARE_K_21 NUMBER,
    SPARE_K_22 NUMBER,
    SPARE_K_23 NUMBER,
    SPARE_K_24 NUMBER,
    SPARE_L_1 NUMBER,
    SPARE_L_2 NUMBER,
    SPARE_L_3 NUMBER,
    SPARE_L_4 NUMBER,
    SPARE_L_5 NUMBER,
    SPARE_L_6 NUMBER,
    SPARE_L_7 NUMBER,
    SPARE_L_8 NUMBER,
    SPARE_L_9 NUMBER,
    SPARE_L_10 NUMBER,
    SPARE_L_11 NUMBER,
    SPARE_L_12 NUMBER,
    SPARE_L_13 NUMBER,
    SPARE_L_14 NUMBER,
    SPARE_L_15 NUMBER,
    SPARE_L_16 NUMBER,
    SPARE_L_17 NUMBER,
    SPARE_L_18 NUMBER,
    SPARE_L_19 NUMBER,
    SPARE_L_20 NUMBER,
    SPARE_L_21 NUMBER,
    SPARE_L_22 NUMBER,
    SPARE_L_23 NUMBER,
    SPARE_L_24 NUMBER
)
pctfree 30 pctused 60
STORAGE( INITIAL 200K NEXT 200K
PCTINCREASE 0 MAXEXTENTS 500 )
TABLESPACE USERS;


grant all on numbertable to public;
```

```
--==========================================================================
-- Copyright Saugato Mukerji
--==========================================================================
-- SYSTEM:         Thesis
-- SUBSYSTEM:      Sparse Data
-- FILE NAME:      RunNumberTable.sql
-- DESCRIPTION:    runs a sql script to insert rows of 300 numbers into the
--                 numbertable and measures space used in datafiles.
--
-- AUTHOR:         Saugato Mukerji
-- CREATED:        10-05-99
-- USAGE:
-- INPUTS:
-- OUTPUTS:
-- NOTES:
--
-- REVISION HISTORY:
-- $Author$
-- $Id$
--
-- $Log$
--
--==========================================================================
set serveroutput on


DECLARE
    start_time  CHAR(5);
    end_time    CHAR(5);
    free_Space NUMBER;
    used_Space NUMBER;
    table_extents NUMBER;
    table_blocks NUMBER;
    segment_name varchar2(30);
    segment_owner varchar2(30);
    segment_initial_extent number;
    segment_max_extents number;
    aname   CHAR(30);
    X   NUMBER;
    Y NUMBER;
    numRows NUMBER;
BEGIN
    X := 10;
    Y:= 1000;
    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;

    DBMS_OUTPUT.PUT_LINE( 'owner'
                          ||'    '||'name'
                          ||'    '||'extents'
                          ||'    '||'blocks'
                          ||'    '||'initial_extent'
                          ||'    '||'max_extents'
                          ||'    '||'start_time'
                          ||'    '||'end_time'
                          ||'    '||'free space in kb '
                          ||'    '||'used space'
                          ||'    '||'numRows' );
    FOR J IN 1..X LOOP
```

```
        SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;

        FOR K IN 1..Y LOOP

            insert into NUMBERTABLE
            values( sysdate, ' ax122',

0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,
3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,
6,7,8,9,0,1,2,3,4,5,6,7,8,9,

0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,
3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,
6,7,8,9,0,1,2,3,4,5,6,7,8,9,

0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,
3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,
6,7
            );

            COMMIT;

        END LOOP;

        SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
        -- time the end of the 1000 inserts

        DBMS_DDL.ANALYZE_OBJECT (
            'TABLE',
            NULL,
            'NUMBERTABLE',
            'COMPUTE',
            NULL,
            NULL,
            NULL,
            NULL);

        SELECT OWNER , SEGMENT_NAME, EXTENTS , BLOCKS, INITIAL_EXTENT, MAX_EXTENTS
                INTO
                segment_owner, segment_name, table_extents, table_blocks,
                segment_initial_extent, segment_max_extents
                FROM DBA_SEGMENTS
                WHERE SEGMENT_NAME='NUMBERTABLE';

        select sum(bytes)/1024 INTO free_space from dba_free_space;
        select sum(bytes)/1024 INTO used_space from dba_segments;
        select count(*) INTO numRows from NUMBERTABLE;

        DBMS_OUTPUT.PUT_LINE( segment_owner
                              ||'   '||segment_name
                              ||'   '||table_extents
                              ||'   '||table_blocks
                              ||'   '||segment_initial_extent
                              ||'   '||segment_max_extents
                              ||'   '||start_time
                              ||'   '||end_time
                              ||'   '||free_space
                              ||'   '||used_space
                              ||'   '||numRows );
```

```
    END LOOP;
END;
```

```
--===============================================================================
-- Copyright Saugato Mukerji
--===============================================================================
-- SYSTEM:       Thesis
-- SUBSYSTEM:    Sparse Data
-- FILE NAME:    RunNumberTable90pcNull.sql
-- DESCRIPTION:  runs a sql script to insert rows of 300 numbers into the
--               numbertable and measures space used in datafiles. 90 % of
--               values inserted in each row were null creating a sparse table.
--
-- AUTHOR:       Saugato Mukerji
-- CREATED:      10-05-99
-- USAGE:
-- INPUTS:
-- OUTPUTS:
-- NOTES:
--
-- REVISION HISTORY:
-- $Author$
-- $Id$
--
-- $Log$
--
--===============================================================================
set serveroutput on


DECLARE
    start_time  CHAR(5);
    end_time    CHAR(5);
    free_Space NUMBER;
    used_Space NUMBER;
    table_extents NUMBER;
    table_blocks NUMBER;
    segment_name varchar2(30);
    segment_owner varchar2(30);
    segment_initial_extent number;
    segment_max_extents number;
    aname   CHAR(30);
    X   NUMBER;
    Y NUMBER;
    numRows NUMBER;
BEGIN
    X := 10;
    Y:= 1000;
    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;

    DBMS_OUTPUT.PUT_LINE( 'owner'
                          ||'   '||'name'
                          ||'            '||'extents'
                          ||'   '||'blocks'
                          ||'   '||'ini_ext'
                          ||'   '||'max_ext'
                          ||'   '||'start'
                          ||'   '||'end'
                          ||'   '||'free kb '
                          ||'   '||'used kb'
                          ||'   '||'numRows' );
    FOR J IN 1..X LOOP
```

```
        SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;

        FOR K IN 1..Y LOOP

             insert into NUMBERTABLE
             values( sysdate, ' ax122',

0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,null,null,null,nu
ll,null,0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,null,null
,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,n
ull,null,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,null,nul
l,null,null,null,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,
null,null,null,null,null,null,null,null,

0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,null,null,null,nu
ll,null,0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,null,null
,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,n
ull,null,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,null,nul
l,null,null,null,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,
null,null,null,null,null,null,null,null,

0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,null,null,null,nu
ll,null,0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,null,null
,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,null,null,null,n
ull,null,null,null,null,0,null,null,null,null,null,null,null,null,null,0,null,null,nul
l,null,null,null,null,null,null,0,null,null,null,null,null,null,null
             );

             COMMIT;

        END LOOP;

        SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
        -- time the end of the 1000 inserts

        DBMS_DDL.ANALYZE_OBJECT (
             'TABLE',
             NULL,
             'NUMBERTABLE',
             'COMPUTE',
             NULL,
             NULL,
             NULL,
             NULL);

        SELECT OWNER , LTRIM( RTRIM( SEGMENT_NAME ) ), EXTENTS , BLOCKS,
INITIAL_EXTENT, MAX_EXTENTS
                INTO
                segment_owner, segment_name, table_extents, table_blocks,
                segment_initial_extent, segment_max_extents
                FROM DBA_SEGMENTS
                WHERE SEGMENT_NAME='NUMBERTABLE';

        select sum(bytes)/1024 INTO free_space from dba_free_space;
        select sum(bytes)/1024 INTO used_space from dba_segments;
        select count(*) INTO numRows from NUMBERTABLE;

        DBMS_OUTPUT.PUT_LINE( segment_owner
                                        ||'    '||segment_name
                                        ||'    '||table_extents
                                        ||'    '||table_blocks
                                        ||'    '||segment_initial_extent
                                        ||'    '||segment_max_extents
                                        ||'    '||start_time
                                        ||'    '||end_time
                                        ||'    '||free_space
                                        ||'    '||used_space
                                        ||'    '||numRows );
    END LOOP;
END;
```

6.4 Source code for synchronous compression algorithm prototype

```
//==============================================================
================
// Copyright 1996 BHP IT All Rights Limited
//==============================================================
================
// SYSTEM:         Plate Rolling Mill Control System
// SUBSYSTEM:      IO
// FILE NAME:      CompressArray.java
// DESCRIPTION:    a java io compression for 200 points
// AUTHOR:         Saugato Mukerji
// CREATED:        25 Jan 1998
// LIBRARY:
//
// REVISION HISTORY:
// Initial revision
//
//
//==============================================================
================

// Package name.
package hsm.Components;


// Java imports.
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import java.io.*;
import java.net.*;
import hsm.Client.*;


//==============================================================
================
// = CLASS NAME
// CompressArray
//
// = DESCRIPTION
// This class is the main class of the ESPProtocolHandler. The
class recieves
// data for transmission to and from the terminal server port
inerfacing the
// external ESP devices.
//
//==============================================================
================
public class CompressArray
{
    static final int PASS_BAND = 120;   // had 55% comp with
100;
        // default allowed pass band

    static final int MAX_SKIP = 300;
        // default max number of skipped time in millisecond

    int myIn[];
    boolean myProcessingBusy = false;
        //

    int myI=0;
        // message count. We are getting messages synchronously
in 50ms intervals

    long myLastLoggedTime[];
        // last recorded message count for each.

    int myMaxSkip[];
        // max number of skipped values.

    int myYi_1[];
        // last value in local memory

    int myYi[];
        // current value in local memory

    int myPassBand[]; // 19% with 3;
        // allowed passband calculated in absolute values full
scale. i.e. 0.5%FS

    int myDeltaY[];
        // the delta value computed as       myDeltaY = myYi -
myYi-1     .

    boolean myFlagYi_1Logged[];   // = false;
        // flag to indicate if the last value in memory had
broken the criteria and
        // been logged as a result.

    long mySkipCount[];
        // count of skipped values

    long myTotalCount[];
        // count of all values


    int myIoArraySize = 200;
        // variable to aloow cofiguration of the size of the io
list


    public void init()
    {
        System.out.println(" in init() ==================");

        myIoArraySize = 200;
            // set array size

        myLastLoggedTime = new long[myIoArraySize];
        myYi_1 = new int[myIoArraySize];
        myYi  = new int[myIoArraySize];
        myPassBand = new int[myIoArraySize];
        myDeltaY = new int[myIoArraySize];
        myFlagYi_1Logged = new boolean[myIoArraySize];
        mySkipCount = new long[myIoArraySize];
        myTotalCount = new long[myIoArraySize];
        myMaxSkip = new int[myIoArraySize];

            // allocate memory for arrays to hold intermediate
values during compression

        for(int i=0; i<myIoArraySize; i++)
        {
            myLastLoggedTime[i]=0;
            myYi_1[i]=0;
            myYi[i]=0;
            myPassBand[i]=PASS_BAND;
            myMaxSkip[i]=MAX_SKIP;
            myDeltaY[i]=0;
            myFlagYi_1Logged[i]=true; // false; done to skip
logging first value
            mySkipCount[i]=0;
        }

        loadTheProps();
            // load the user specified properties

    }



//=============================================================
================
    // Function
    // setIoArraySize
    //
    // = DESCRIPTION
    // Constructor
    //

//=============================================================
================
    public void setIoArraySize( int aIoArraySize )
    {
        myIoArraySize = aIoArraySize;
    }


//=============================================================
================
    // Function
    // CompressArray
    //
    // = DESCRIPTION
    // Constructor
    //

//=============================================================
================

    void CompressArray()
    {
        System.out.println(" Constucting CompressArray ==");
    }
```

```
//=============================================================
================
    // Function
    // store
    //
    // = DESCRIPTION
    // log the value to history
    //

//=============================================================
================
    void store( int value, int count )
    {
        // System.out.println("writing to history i="+count+"
value="+value);
        // System.out.println(""+count+","+value);
    }

/*


//=============================================================
================
    // Function
    // setDataArray
    //
    // = DESCRIPTION
    // log the value to history
    //

//=============================================================
================
    void setDataArray( String rawData )
    {
        String token;
        int i=0;
        StringTokenizer st = new StringTokenizer(rawData);
        while (st.hasMoreTokens())
        {
            token = st.nextToken();
            try
            {
                dataArray[i] = Integer.parseInt( token );
            }
            catch( NumberFormatException e )
            {
                dataArray[i] = 0xFFFF;
                System.out.println
                ("\n setting dataArray["+i+"]="+dataArray[i]);

            }
        }
    }
*/


//==============================================================
================
    // Function
    // processData
    //
    // = DESCRIPTION
    // run through the array of data and processes one point for
compression in each array
    //

//============================================================
================
    public short  processData( int[] rawData, int[] values,
                                            int[] tags, long
timeStamp )
    {
        if( myProcessingBusy == true )
        {
            return -1;
        }

        myProcessingBusy = true;


        Trace.level5("in processData");

        int tagCount=0;
            // index of non rejected data entries
        for( int i=0; i<myIoArraySize; i++ )
        {


            myDeltaY[i]=myYi[i]-myYi_1[i];
                // delta y from last time

            // 27/05/00
            myYi_1[i]=myYi[i];
            //27/05/00



            // myYi[i]=dataArray[i];
            myYi[i]=rawData[i];
                // read the new value




            int c = myYi_1[i] + myDeltaY[i] + myPassBand[i];
            int d = myYi_1[i] + myDeltaY[i] - myPassBand[i];
                // calculate pass band limit

            if( i == 10)
            {
                System.out.println("entering loop in
processData, myDeltaY[i]="+
                myDeltaY[i]+" myYi[i]"+myYi[i]+"
myYi_1[i]"+myYi_1[i]+" c="+c+" d="+d+" mypb[i]="+myPassBand[i]
);
            }


            myI=i;
            boolean skipFlag = false;

            // record the event that a value is about to be
logged because the
            // number of allowed skips has been exceeded
            if( ( timeStamp - myLastLoggedTime[i] ) >
myMaxSkip[i] )
            {
                skipFlag = true;
            }


            if( ( myYi[i] > c ) || ( myYi[i] < d ) || ( skipFlag
== true ) )
                // test if the new data is outside the allowable
pass band
            {



               if( myYi[i] > 4000 || myYi[i] < -4000 )
               {
                   ;// System.out.println( "============ c="+c+"
d="+d+" myYi["+i+"]="+myYi[i] );
               }



                if( myFlagYi_1Logged[i] == false  && skipFlag ==
false )
                    // log the last value too if it was a point
skipped by the compression logic
                    // earlier. The  myFlagYi_1Logged[i] is set
true when a new point is logged
                {
                    values[tagCount]=myYi_1[i];
                        // save value
                    tags[tagCount]=i+1;
                        //save index of non rejected tag. This
will point to the history record
                        // i.e.History54 where i = 53

                    if( i==10 )
                    {
                        System.out.println( "i="+i+" Retaining
Yi-1 values["+tagCount+"]="+values[tagCount]+" sf="+skipFlag );
                    }



                    tagCount++;
                        // increment count of non rejected tags

                    //Trace.level4(" A tagCount="+tagCount+"
i="+i );


                        // store( myYi_1[i], Math.max(0,myI-1 )
);
                        // make necessary calls to log the point
before the current to history
                }

                    //System.out.print("logging current: ");
                    // CompressedString+=" "+myI;
                    //store( myYi[i], myI );
                    // make necessary calls to log the current
point to history



                values[tagCount]=myYi[i];
                    // save value
                if( i==10 )
                {
                    System.out.println( "i="+i+" Retaining Y i
values["+tagCount+"]="+values[tagCount]+" sf="+skipFlag );
                }

                tags[tagCount]=i+1;
                    //save index
                tagCount++;
                    // increment count



                myFlagYi_1Logged[i] = true;
                   // flag the logging into history of the
current point

                //myLastLoggedI[i]=myI;
                    // save last index

               myLastLoggedTime[i]=timeStamp;
                   // save the time when last reading was logged

                //Trace.level4(" B tagCount="+tagCount+" i="+i
);


                //System.out.println( i+","+myYi[i] );


            }
            else
                // skip the point for now it is within the pass
band
            {
                mySkipCount[i]++;
                    // count skips for this point

                //System.out.println("....for now skipping
i="+myI+" myYi=<"+myYi+"> myYi_1[i]="+myYi_1[i] );

                //System.out.print( " skip "+(timeStamp -
myLastLoggedTime[i])+","+myI+" Yi="+myYi[i]+"i="+i );

                myFlagYi_1Logged[i] = false;

            //}
```

```
                //myYi_1[i]=myYi[i];
            }

            int k = Math.max(0,tagCount-1);
            //Trace.level4( "loopend
values[tagCount-1]="+values[k]+
            //                      " tags[tagCount-1]="+tags[k]+
            //                      " tagCount-1="+k);


           myTotalCount[i]++;
                // count total vlues for this point


        }

        myProcessingBusy = false;

        return  (short)tagCount;
    }





//=============================================================
================
    // Function
    // loadTheProps
    //
    // = DESCRIPTION
    // loads the configurable properties

//=============================================================
================
    public void loadTheProps()
    {
        String skipString = "";

        skipString = System.getProperty( "user.dir" );
        System.out.println( "skipString="+skipString );
            // test using default system property

        // System.setProperty("HSM.skip","a.b.cd.efgh.g");
        // this forces a new System.property entry HSM.skip


        // This code loads property strings from a file in the
classpath
        Properties prop = new Properties();
        try
        {
            prop.load( new FileInputStream( "HSM.config" ) );
        }
        catch (Exception e)
        {
           System.out.println("Error loading CORVUS or
Securities property file: "+ e.toString());
```

```
        }

        System.setProperties(prop);
            // force the properties loaded from the file into
the System Property

        Enumeration propertyEnum =
System.getProperties().propertyNames();
        Object propertyKey;
        int i=0;
        int j=0;

        while (propertyEnum.hasMoreElements())
        {
             propertyKey = propertyEnum.nextElement();
             String propString = (String)propertyKey;

             if( propString.indexOf( "HSM.skip" ) != -1 )
             {
                String indxString =
                propString.substring( "HSM.skip".length(),
propString.length() );

//System.out.println("indxString=["+indxString+"]");


                String p =  System.getProperty( propString );
                //System.out.println("propString="+propString+"
p="+p);

                try{
                    i=Integer.parseInt( indxString );
                    i = i - 1;
                    myMaxSkip[i] = Integer.parseInt( p );
                }
                catch( Exception e)
                {
                    System.out.println("skip=["+propString+"]
i="+i+" Error in parsing skip value: "+ e.toString());
                }

             }

             if( propString.indexOf( "HSM.passBand" ) != -1 )
             {
                String indxString =
                propString.substring( "HSM.passBand".length(),
propString.length() );

                String p =  System.getProperty( propString );
                //System.out.println("propString="+propString+"
p="+p);

                try{
                    i=Integer.parseInt( indxString );
                    i = i - 1;
                    myPassBand[i] = Integer.parseInt( p );
                }
                catch( Exception e)
                {
```

```
System.out.println("myPassBand=["+propString+"] i="+i+" Error in
parsing myPassBand value: "+ e.toString());
                }

            }

        }


        for( i=0; i<myIoArraySize; i++)
        {

System.out.println("myPassBand["+i+"]="+myPassBand[i]+"
myMaxSkip["+i+"]="+myMaxSkip[i]);

        }


    }



//*************************************************************
****************
    // Function
    // printSkipStats
    //
    // - DESCRIPTION
    // run through the array of data and processes one point for
compression in each array
    //

//*************************************************************
****************
    void printSkipStats()
    {
        for(int i=0; i<myIoArraySize; i++)
        {
            System.out.println( " Total count ="+
myTotalCount[i]+
                                " skip count for "+i+"th input
="+mySkipCount[i]+
                                " compression ratio="+
                                (100 * ( myTotalCount[i] -
mySkipCount[i] ) / myTotalCount[i] ) );

        }
    }

}
```

```
// Package name.
package hsm.Client;



import java.lang.*;
import java.util.*;
import java.io.*;
// You need to import the java.sql package to use JDBC
import java.sql.*;
// System imports
import hsm.Components.*;


public class MessageClient implements Runnable
{
    // The connect string
    //static final String connect_string =
    //            "jdbc:oracle:thin:scott/tiger@home:1521:ORCL";
    static final String connect_string =
                "jdbc:oracle:thin:scott/tiger@ISDWOL-PC5952:1521:ORCL";
    static ResultSet rset;
    PreparedStatement anInsertStatement;
    static final String NULL_MARKER = "'.'";

    static String nullAndComma = ""+NULL_MARKER+",";

    static int count = 0;
    static final int numColumnsInTable = 288;

    static final long FULLSCALE = 4096;
        // base address for cisco terminal servers
    static int myOffset = 0;
        // offset for generating varying data from fixed array
    static long startTime=System.currentTimeMillis();

    int myIoArraySize = 200;
            // set array size
    short myDeviceId = 0;
        // short myNumberOfPoints = 0;
    String argString[] = new String[10];
        // storage for command line arguments

    CompressArray myCompressArray;
        // declare the compressor object
    int [] myValues;
        // array of not rejected data values
    // int [] myRawDataArray ;
    int myRawDataArray[] =
{-2409,-2329,-3273,-3189,461,-3056,455,-3079,448,-3099,429,-3139,418,-3164,401,-3
190,384,-3227,369,-2248,-2533,-2608,-2656,-2687,-2667,-2733,-2725,-2757,-2761,-27
63,-2767,-2763,-2765,-2751,-2737,-2671,-2651,-2623,-2603,-2596,-2592,-2584,-2580,
-2579,-2577,-2573,-2569,-2570,-2572,-2159,-2138,-1958,-1940,-1923,-1909,-1891,-18
80,-1863,-1846,-1834,-1812,-1790,-1774,-1751,-1729,-1713,-1691,-1673,-1650,-1627,
-1613,-1584,-1560,-1531,-1500,-1471,-1433,-1384,-1353,-1305,-1274,-1236,-1208,-11
87,-1161,-1133,-1116,-1083,-1062,-1031,-1002,-978,-944,-916,-898,-875,-859,-835,-
813,-796,-774,-755,-742,-725,-711,-695,-678,-670,-656,-643,-635,-623,-613,-603,-5
92,-582,-572,-565,-556,-548,-544,-542,-536,-531,-526,-521,-514,-499,-486,-469,-44
7,-435,-420,-409,-398,-385,-378,-368,-358,-352,-343,-336,-325,-319,-310,-301,-294
,-286,-277,-271,-262,-252,-246,-238,-232,-226,-217,-211,-203,-197,-190,-183,-177,
-171,-161,-156,-148,-139,-136,-127,-121,-114,-106,-101,-95,-88,-82,-76,-70,-63,-5
6,-51,-44,-40,-32,-25,-22,-15,-7,-2,4,11,17,25,29,36,44,47,52,56,59,62,65,67,68,7
2,72,76,77,79,81,85,86,87,90,89,91,94,94,96,97,101,102,105,107,110,111,114,116,11
7,119,122,125,127,129,133,134,134,137,138,137,138,138,138,136,137,136,136,135,133
,133,133,132,131,130,131,131,129,130,129,129,128,128,128,127,127,126,126,127,127,
127,126,126,126,126,125,126,126,126,126,125,125,125,125,124,125,124,123,118,115,1
13,111,112,109,107,108,104,104,102,98,96,91,87,83,81,78,74,72,69,67,67,66,64,62,6
0,58,55,52,48,44,45,41,39,37,36,28,30,31,26,22,23,22,20,19,15,15,13,11,7,8,5,3,1,
-2,1,0,-2,-3,-6,-6,-7,-7,-9,-10,-10,-10,-12,-13,-15,-14,-14,-14,-14,-15,-18,-38,-
19,-16,-17,-18,-19,-18,-22,-21,-24,-25,-26,-27,-26,-27,-30,-29,-32,-34,-34,-37,-3
6,-39,-37,-41,-40,-41,-42,-44,-44,-46,-45,-44,-44,-47,-47,-47,-47,-48,-46,-48,-51
```

```
,-49,-52,-49,-48,-49,-49,-51,-54,-49,-49,-49,-50,-49,-49,-50,-50,-50,-50,-50,-53,
-52,-53,-53,-54,-52,-52,-51,-52,-53,-51,-49,-48,-49,-47,-47,-47,-49,-50,-54,-49,-
50,-50,-50,-51,-53,-56,-56,-54,-55,-56,-55,-56,-55,-56,-55,-56,-53,-52,-51,-53,-5
3,-52,-52,-52,-50,-51,-51,-54,-49,-47,-46,-44,-45,-45,-39,-41,-41,-38,-36,-37,-36
,-35,-35,-36,-36,-36,-33,-34,-29,-31,-29,-27,-27,-26,-23,-23,-21,-18,-18,-20,-19,
-21,-15,-15,-16,-16,-14,-14,-14,-15,-12,-11,-10,-8,-6,-8,-4,-4,-4,-2,1,0,2,2,3,2,
7,6,9,8,11,11,13,12,16,16,15,18,18,17,17,17,19,20,21,20,22,24,24,7,25,25,28,28,31
,32,32,32,34,33,34,36,34,36,38,38,40,41,42,44,45,45,46,46,48,45,49,50,54,53,55,54
,54,54,56,53,56,57,55,55,51,52,50,48,50,50,47,49,47,44,44,45,42,41,39,39,38,36,34
,30,26,24,22,20,17,14,11,11,10,10,10,8,10,10,10,9,10,9,8,8,8,8,7,8,7,7,7,8,8,8,6,
5,5,6,4,5,4,6,6,4,5,6,5,6,5,5,3,4,5,4,4,4,3,3,3,3,4,4,2,1,1,0,2,2,1,0,-1,-1,1,0,0
,0,0,-2,-2,-2,-2,-2,-1,-1,-4,-4,-4,-4,-3,-3,-4,-4,-4,-4,-5,-3,-4,-5,-5,-5,-6,-6,-
5,-6,-7,-6,-6,-6,-6,-6,-6,-8,-7,-7,-9,-8,-8,-8,-8,-8,-9,-10,-11,-19,-14,-11,-9,-1
0,-11,-10,-10,-12,-13,-13,-12,-13,-12,-13,-13,-13,-13,-14,-14,-13,-13,-15,-15,-14
,-16,-14,-14,-15,-15,-15,-16,-18,-17,-16,-17,-17,-18,-18,-20,-19,-18,-20,-19,-21,
-21,-21,-21,-21,-21,-22,-22,-22,-22,-22,-23,-25,-24,-24,-25,-24,-23,-23,-22,-22,-
23,-23,-22,-20,-19,-19,-15,-16,-13,-11,-10,-7,-4,-2,-1,2,9,13,19,26,30,32,35,38,3
9,44,45,42,39,36,34,34,32,33,33,35,32,32,35,37,37,38,37,38,39,39,39,40,40,41,45,4
6,46,50,51,53,55,53,60,63,64,69,74,79,80,87,91,96,103,113,118,126,135,144,151,161
,168,178,188,195,200,209,221,233,241,250,258,276,298,319,348,364,394,403,446,471,
506,546,611,618,633,626,636,636,655,681,711,721,799,-3466,2840,5101,5024,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

        // array of raw data data values
    int [] myDataArray;
        // array of data values extrcted at a progressive offset to simulate
        // progress of the sine wave

    int [] myTags;
        // array of index values corresponding to the not rejected data
    long myTimestamp = 0;
        // time stamp of the observation in milliseconds
    short myNumberOfValues = 0;
        // count of the number of selected values


    public static void main(String args[])
    {
         MessageClient aMessageClient = new MessageClient();
            // instantiate a MessageClient

         short  portNumber=0;
         aMessageClient.setParams( args, portNumber );

         System.out.println(" here 1");


         MessageTimer aMessageTimer = new MessageTimer( aMessageClient, 50 );
            // setup for a 50 ms timer and pass reference to aMessageClient


         aMessageClient.initCompressor( args, portNumber );

         aMessageClient.start();
            // start  aMessageClient
         aMessageTimer.start();
            // start timer

        refreshConfig refresh = new refreshConfig( aMessageClient, "HSM.config"
);
        refresh.run();
            // allow dynamic change of per input compression parameters

    }






    public void setParams( String aString[], short aDeviceId )
        // save parameters
    {
        for( int i=0; i < aString.length; i++)
        {
            argString[i] = aString[i];
        }
    }


    public void initCompressor( String aString[], short aDeviceId )
    {

        System.out.println("entering initCompressor." );

        myCompressArray = new  CompressArray();
            // instantiate the Compressor object

        myCompressArray.init();

        System.out.println(" after creating CompressArray()");

        myDeviceId = aDeviceId;

        myRawDataArray = new int[myIoArraySize];
        myDataArray = new int[myIoArraySize];
        myValues = new int[2*myIoArraySize];
        myTags = new int[2*myIoArraySize];


            // set up sin values 180/200 degree apart.
        for( int i=0; i<200; i++)
        {
            myRawDataArray[i] = (int)( FULLSCALE * Math.sin( ( 2*3.14 / 200 ) * i
));
            // System.out.println( "myRawDataArray["+i+"]"+myRawDataArray[i] );
        }

        System.out.println("exitting initCompressor." );

    }





    void start() {
        System.out.println(" entering MessageClient.start()");
    }
    public void run() {
        System.out.println(" entering run");

        while(true)
        {
            try{
                Thread.sleep( 1000 );
            }
            catch( InterruptedException e )
            {
                System.out.println( ""+e );
            }
        }
    }




    void triggerMessage()
    {
        sendData();

    }


    void loadTheProps()
    {
        myCompressArray.loadTheProps();
            // cause re loading of congi parameters

    }


public void sendData()
{
    // call the Message server object and print results

    myTimestamp = System.currentTimeMillis();


    // System.out.println("MessageClient.sendData time="+myTimestamp);

    long tenPluscount = 0;
    long zerocount = 0;
    long tencount = 0;
    long result = 0;
    long total = 0;
    long delta = 0;
    long sleepPeriod = 0;
    long loopCount = 10000000;
    long cumCallTime = 0;
    long cumDataVolume = 0;
    String dataString = "";

    int j=0;

            // create a short lived disturbance by forcing a small spike
    myRawDataArray[5] = -4001;
    myRawDataArray[6] = 4001;

    myRawDataArray[105] = 400;
    myRawDataArray[106] =  -400;


    for(int i=0; i<myIoArraySize; i++)
        // copy with an offset from raw data, wrap around at end
    {
        j= ( myOffset + i ) % myIoArraySize;
        myDataArray[i] = myRawDataArray[j];
        myValues[i]=0;
        myTags[i]=0;

    }

    myOffset++;
        // incr offset

    myNumberOfValues =
    myCompressArray.processData( myDataArray, myValues, myTags, myTimestamp );
    //System.out.print( "-");


    sendMessage( myValues, myTags, myTimestamp, myNumberOfValues, myDeviceId );
}




// ============================================================
public void sendMessage( int[] values,
                             int[] tags,
                             long timestamp,
                             short numberOfValues,
                             short deviceId )
{
//        if( count%100 == 0 )
//        System.out.println( "\n.numberOfValues="+numberOfValues+" timestamp =
"+timestamp);

        int [] localValues = values;
        int [] localTags = tags;
        long localTimestamp = timestamp;

        short localNumOfValues = numberOfValues;
        short localDeviceId = deviceId;
            // copy the passed in values to local variables to release
            // the remote variables


    try
    {
        System.out.print(".");
        if( anInsertStatement == null )
        {
            System.out.println("Connecting ......");
            // Load the Oracle JDBC driver
            Class.forName ("oracle.jdbc.driver.OracleDriver");

            // Connect to the database
            // You must put a database name after the @ sign in the connection
URL.
            // You can use either the fully specified SQL*net syntax or a short
cut
            // syntax as <host>:<port>:<sid>.  The example uses the short cut
syntax.


            Connection conn =
            DriverManager.getConnection (connect_string);

            conn.setAutoCommit( true );

            System.out.println("got connection......");

            anInsertStatement = conn.prepareStatement
            ("insert into vchartable values( ?, "+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,"+
                "  ?, ?, ?, ?, ?, ?, ?, ? )"
            );

        }


/*
        System.out.println("after prepare statement ......");
        for( int k=0; k<289; k++)
        {
            anInsertStatement.setString(k+1, String.valueOf( k ) );
        }
*/

        anInsertStatement.setString(1, String.valueOf( timestamp ) );
            // write timestamp value into the first column

        // ==================================================
        // skip 1st column so go from col 2 to numColumnsInTable
        // ==================================================
        int j=0;
        int nextDataIndex = 0;
        int dataValue = 0;
        boolean foundDataFlag = false;
        for(int i = 0; i < numColumnsInTable ; i++)
        {

            if(  j < numberOfValues  )
            {
                nextDataIndex = tags[j];
                    // the values[j] has the next non null data
                if( nextDataIndex == i )
                {
                    dataValue = values[j];
                    j++;
                    foundDataFlag = true;
                }
            }


            if( foundDataFlag == true )
            {
                anInsertStatement.setString(i+2, String.valueOf( dataValue ) );
                    // place the non null data at (i+2)th column because
                    // i goes fom 0..n and the first data value column is 2
            }
            else
            {
                anInsertStatement.setString( i+2, NULL_MARKER );
                    // fill remaining columns with null marker
            }

        }

        anInsertStatement.execute();

    }
    catch( java.sql.SQLException e)
    {
        System.out.println( "Exception :"+e);
    }
    catch( ClassNotFoundException e)
    {
        System.out.println( "Exception :"+e);
    }
}

}
```

```
package hsm.Client;


import java.io.*;
//import java.util.*;


class refreshConfig implements Runnable {

    static final int REFRESH_CHECK_PERIOD = 2000;
        // interval in ms after which the file is checked fot config changes

    MessageClient myParentRef;
        // reference to the parent class which instantiated this

    String myFileName = "";
        // local copy of filename



    public refreshConfig( MessageClient aMessageClient, String fileName)
    {
        myParentRef = aMessageClient;
            // save reference to the parent.

        myFileName = fileName;
            // save passed file name.
    }



    public void run()
    {
        File fileRef = new File( myFileName );
            // create file handle

        long lastModifiedTime = 0;
            // last modification time on file in ms from 1-1-1970

        lastModifiedTime = fileRef.lastModified();
            // save the last file mod time


        // enter endless loop to look for file mod times
        while( true )
        {

            try{
                Thread.sleep( REFRESH_CHECK_PERIOD );
            }
            catch( InterruptedException e )
            {
                System.out.println( "Error "+e );
            }


            if( fileRef.lastModified() > lastModifiedTime  )
            {
                myParentRef.loadTheProps();
                    // force the reading of the configuration data

                lastModifiedTime = fileRef.lastModified();
                    // save the last file mod time

            }

        }

    }

}
```

```
rem *************************
rem Setup java home directory
rem *************************
SET JAVA_HOME=m:\jdk1.2

rem *************************
rem Setup java bin path
rem *************************
SET PATH=%JAVA_HOME%\bin;%PATH%

rem SET CLASSPATH=

rem *********************************
rem compile java code and place under
rem javaclasses directory
rem ********************************

javac -d .\javaclasses *.java

pause
```

```
            // add a sleep equal to the the timer interval less the
            // time taken by the last sendMessage call

            long callTime =
            System.currentTimeMillis() - myStartOfSleepMilliSeconds ;

            if( callTime > myInterval )
                // if call time exceeded 50 ms
            {
                myStartOfSleepMilliSeconds += callTime;
                System.out.print( " call="+callTime );
                continue;
            }

            long sleepTime =
            myInterval - Math.min( myInterval, callTime );


            try
            {
                Thread.sleep( sleepTime );
            }
            catch ( InterruptedException e )
            {
            }


            myStartOfSleepMilliSeconds = System.currentTimeMillis();
            myMessageClient.triggerMessage();
                // trigger the sending of the message


        }
    }

//=============================================================================
    // = FUNCTION NAME
    // start
    //
    // = DESCRIPTION
    // This function starts the timer thread.
    //==========================================================================
===
    public void start()
        // Overridden start method to start timer thread.
    {
        Thread thread = new Thread( this );
        thread.start();
    }



    // --------------------
    // Private Data Members
    // --------------------
    MessageClient   myMessageClient;
        // space for reference to passed MessageClient object.
        // Will be used to invoke the methods in the passed
        // MessageClient object ie myMessageClient.pollTimerExpired()

    long    myInterval;
        // space for passed timer interval

    long myStartOfSleepMilliSeconds = 0;
        // the starting time of the current period


}
```

```
//===============================================================================
// Copyright 1996 BHP IT All Rights Limited
//===============================================================================
// SYSTEM:        Plate Rolling Mill Control System
// SUBSYSTEM:     GUI
// FILE NAME:     MessageTimer.java
// DESCRIPTION:   Java implementation of BarTemperatureScanScreen.
// AUTHOR:        Saugato Mukerji
// CREATED:       12 November 1998 17:45:00
// LIBRARY:       hsm.Components
//
// REVISION HISTORY:
// $Author: macorv $
// $Id: MessageTimer.java,v 1.1 1999/02/16 08:08:20 macorv Exp macorv $
//
// $Log: MessageTimer.java,v $
// Revision 1.1  1999/02/16 08:08:20  macorv
// Initial revision
//
//===============================================================================

// Package name
package hsm.Client;

// Java imports
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.text.*;
import java.util.*;

import hsm.Client.*;

//===============================================================================
// = CLASS NAME
// MessageTimer
//
// = DESCRIPTION
// This class provides a DateTime tick. The TimerBean can be placed at a
// suitable location on any screen page.
//===============================================================================
public class MessageTimer implements Runnable
{

    //---------------
    // Public Methods
    //---------------

    public MessageTimer( MessageClient aMessageClient, long aPeriod )
        // Constructor
    {
        myMessageClient = aMessageClient;
            // save reference to caller.
        myInterval = aPeriod;
            // save timer interval
    }


//===============================================================================
    // = FUNCTION NAME
    // run
    //
    // = DESCRIPTION
    // This is the main loop of the timer thread.
    //===========================================================================
===
    public void run()
        // Overridden run method executed by timer thread.
    {
        while ( true )
        {
```

```
rem **************************
rem Setup java home directory
rem **************************
SET JAVA_HOME=m:\jdk1.2

rem ********************
rem Setup java bin path
rem ********************
PATH=m:\jdk1.2\bin;%PATH%
set CLASSPATH=m:\Orant\JDBC\lib\classes111.zip;.
java -cp .\javaclasses;%CLASSPATH%;.  hsm.Client.MessageClient
pause
```

6.5.1 Application code to test auto generated Relational Table accessor functions.

```
/*=======================================================================
 * = FUNCTION NAME
 * testSingleObject
 *
 * = DESCRIPTION
 */
/**
 * Tests ROMIS access functions that work on single objects.
 */
/*=======================================================================
 */
public void testSingleObject()
{
    Trace.startBlock(Trace.LEVEL3, "TestClassTestHarness::testSingleObject");
    boolean error = false;
    TestClass testClass = new TestClass();
    testClass.setTestClassData(TEST_CLASS_DATA_1);

    startScenario("Test ROMIS access functions for single objects");

    startCase("Insert / Select");
    {
        TestClass tc1 = new TestClass();
        TestClass tc2 = new TestClass();

        tc1.setTestClassData(TEST_CLASS_DATA_1);
        tc2.setIntValue(tc1.getIntValue());
        tc2.setDateValue(tc1.getDateValue());
        try
        {
            tc1.insert();
            tc2.select();
        }
        catch (DataAlreadyExistsException daee)
        {
            error = true;
            Trace.error("Insertion failed, record already exists", daee);
        }
        catch (IllegalDataAccessException idae)
        {
            error = true;
            Trace.error("Can not access data, access denied", idae);
        }
        catch (InvalidDataException ide)
        {
            error = true;
            Trace.error("Attempt to insert / retrieve invalid data", ide);
        }
        catch (DataNotFoundException dnfe)
        {
            error = true;
            Trace.error("Can not find record to retreive", dnfe);
        }
        catch (DataRetrievalException dre)
        {
            error = true;
            Trace.error("Unable to retrieve data", dre);
        }
        catch (DataStorageException dse)
        {
            error = true;
            Trace.error("Unable to insert data", dse);
        }
        catch (DataException de)
        {
            Trace.error("Can not perform insert / select", de);
            error = true;
        }

        if (error == false)
        {
            testEquality(tc1.getTestClassData(), tc2.getTestClassData());
    }
}

error = false;
startCase("Update / Select");
{
    TestClass tc1 = new TestClass();
    TestClass tc2 = new TestClass();

    // Change every attribute of tc1 except the promary key values.
        tc1.setTestClassData(TEST_CLASS_DATA_2);
        tc1.setIntValue(testClass.getIntValue());
        tc1.setDateValue(testClass.getDateValue());
        tc2.setIntValue(tc1.getIntValue());
        tc2.setDateValue(tc1.getDateValue());

        // Perform the update and select
        try
        {
            tc1.update();
            tc2.select();
        }
        catch (IllegalDataAccessException idae)
        {
            error = true;
            Trace.error("Can not access data, access denied", idae);
        }
        catch (InvalidDataException ide)
        {
            error = true;
            Trace.error("Attempt to update / retrieve invalid data", ide);
        }
        catch (DataNotFoundException dnfe)
        {
            error = true;
            Trace.error("Can not find record to retreive", dnfe);
        }
        catch (DataStorageException dse)
        {
            error = true;
            Trace.error("Unable to update data", dse);
        }
        catch (DataRetrievalException dre)
        {
            error = true;
            Trace.error("Unable to retrieve data", dre);
        }
        catch (DataException de)
        {
            error = true;
            Trace.error("Unkown data exception", de);
        }

        if (error == false)
        {
            testEquality(tc1.getTestClassData(), tc2.getTestClassData());
        }
}

startCase("Remove / Select");
{
        TestClass tc1 = new TestClass();
        TestClass tc2 = new TestClass();

        tc1.setTestClassData(testClass.getTestClassData());
        tc2.setTestClassData(testClass.getTestClassData());

        try
        {
            tc1.remove();
        }
        catch (IllegalDataAccessException idae)
        {
            Trace.error("Can not perform remove", idae);
        }
        catch (DataNotFoundException dnfe)
        {
            Trace.error("Can not find row to delete", dnfe);
        }
        catch (DataException de)
        {
            Trace.error("Unexpected data exception occurred", de);
        }

        boolean dataNotFoundFlag = false;
        try
        {
            tc2.select();
        }
        catch (IllegalDataAccessException idae)
        {
            Trace.error("Can not perform remove", idae);
        }
        catch (DataNotFoundException dnfe)
        {
            dataNotFoundFlag = true;
            Trace.debug(Trace.LEVEL2, "DataNotFoundException Occurred");
        }
        catch (DataRetrievalException dre)
        {
            Trace.error("Can not retrieve data", dre);
        }
        catch (DataException de)
        {
            Trace.error("Unexpected data exception occurred", de);
        }
        check( dataNotFoundFlag == true, "Remove");
}

Trace.endBlock(Trace.LEVEL3, "TestClassTestHarness::testSingleObject");
}
```

6.5.2 primary data class

```
/*==============================================================================
 *
 *==============================================================================
 * SYSTEM:      Hot Strip Mill Data Logger
 * SUBSYSTEM:   data_objects
 * FILE NAME:   TestClass.java
 * DESCRIPTION:
 * AUTHOR:      $Author$
 * CREATED:     9/28/99 9:44:09 AM
 *
 * REVISION HISTORY:
 * $Author: $
 * $Id: $
 *
 * $Log: $
 *
 *==============================================================================
 */

package bhpit.hsmics.database.data_objects;

import java.text.DateFormat;
import java.util.Date;

/*==============================================================================
 * = CLASS NAME
 * TestClass
 *
 * = DESCRIPTION
 */
/**
 * @author     $Author$
 * @version    $Revision$
 */
/*
 * = RAPID TABLE INSTANCES
 *
 * Format - RAPID==Def Rec Name==Inst Rec Name==No. Instances==Piece Dep (Y/N)
 *
 * RAPID==TestClassDef==TestClass==0==Y
 *
 *
 * - ROMIS LOGGING
 *
 * Format - ROMIS_LOGGING==(Y/N)
 *
 * ROMIS_LOGGING==Y
 *
 * - ROMIS INDEXES
 *
 * Format - DATABASE_INDEX==IndexName(col1, col2, etc)
 *
 * DATABASE_INDEX==TestClass_I1(intValue, dateValue)
 * DATABASE_INDEX==TestClass_I2(dateValue)
 */
/*==============================================================================
 */
public class TestClass
{
    public static final int TEST_BIT_1 = 1;

//STARTOFDATAMEMBERSECTION
    /**
     * int value
     *
     * CONSTRAINTS: <intValuePK PRIMARY KEY>
     */
    public int intValue = -1;

    public boolean booleanValue;

    /**
     * float value
     */
    public float floatValue = -2.45f;

    /**
     * short value
     *
     * CONSTRAINTS: <shortValueNN NOT NULL>, <shortValueU UNIQUE>
     */
    public short shortValue = 1234;

    /**
     * double value
     */
    public double doubleValue;

    /**
     * Date value.
     */
    public Date dateValue;

    /**
     * String value
     */
    public char stringValue[] = new char[10];

    /**
     * stringvalue array 1 dimension.
     *
     */
    public char stringValueArray1Dim[][] = new char[4][10];

    /**
     * stringvalue array 2 dimension.
     *
     */
    public char stringValueArray2Dim[][][] = new char[4][5][10];

    /**
     * long value array 1 dimension.
     *
     */
    public int intValueArray1Dim[] = new int[5];

    /**
     * long value array 2 dim
     *
     */
    public int intValueArray2Dim[][] = new int[4][5];

    /**
     * long value array 3 dim
     *
     */
    public int intValueArray3Dim[][][] = new int[4][5][6];

    /**
     * float value array 1 dim.
     *
     */
    public float floatValueArray1Dim[] = new float[6];

    /**
     * Float value array 2 dim
     */
    public float floatValueArray2Dim[][] = new float[4][5];

    /**
     * Float value array 3 dim
     */
    public float floatValueArray3Dim[][][] = new float[4][5][6];

    /**
     * short value array 1 dim.
     *
     */
    public short shortValueArray1Dim[] = new short[7];

    /**
     * Short value array 2 dim
     */
    public short shortValueArray2Dim[][] = new short[4][5];

    /**
     * Short value array 3 dim
     */
    public short shortValueArray3Dim[][][] = new short[4][5][6];

    /**
     * double value array 1 dim
     *
     */
    public double doubleValueArray1Dim[] = new double[8];

    /**
     * Double value array 2 dim
     */
    public double doubleValueArray2Dim[][] = new double[4][5];

    /**
     * Double value array 3 dim
     */
    public double doubleValueArray3Dim[][][] = new double[4][5][6];

    /**
```

```
     * boolean value array 1 dim.
     */
    public boolean booleanValueArray1Dim[] = new boolean[4];

    /**
     * boolean value array 2 dim.
     */
    public boolean booleanValueArray2Dim[][] = new boolean[4][5];

    /**
     * boolean value array 3 dim.
     */
    public boolean booleanValueArray3Dim[][][] = new boolean[4][5][6];

    /**
     * Date value array 1 dim
     */
    public Date dateValueArray1Dim[] = new Date[4];

    /**
     * Date value array 2 dim
     */
    public Date dateValueArray2Dim[][] = new Date[4][5];

    /**
     * Date value array 3 dim
     */
    public Date dateValueArray3Dim[][][] = new Date[4][5][6];

//ENDOFDATAMEMBERSECTION
}
```

6.5.3 Auto generated database access helper

```
/*==============================================================================
 *
 *=============================================================================
 * SYSTEM:      Hot Strip Mill Data Logger
 * SUBSYSTEM:   database
 * FILE NAME:   TestClassROMISAccess.java
 * DESCRIPTION: Interface functions that call the Oracle API and allow JDBC
 *              access to data stored in the database.
 * AUTHOR:      Saugato Mukerji
 * CREATED:     Wed Sep  1 16:19:53 1999

 * LIBRARY:     test.hsmics.database.data_objects
 *
 * REVISION HISTORY:
 *
 *
 *
 *=============================================================================
 */
package test.hsmics.database.data_objects;

// Java imports
import java.math.BigDecimal;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Vector;

// Oracle imports
import oracle.jdbc.driver.*;
import oracle.jdbc.oracore.Util;
import oracle.sql.*;

// Common imports
import test.common.exceptions.DataException;
import test.common.exceptions.DataStorageException;
import test.common.trace.Trace;
import test.common.util.ConnectionManager;

// Project imports
import test.hsmics.database.romis.ROMISHelper;


/*==============================================================================
 * - CLASS NAME
 * TestClassROMISAccess
 *
 * - DESCRIPTION
 */
/**
 * This class provides access to Oracle via the jdbc thin driver.
 */
/*******************************************************************************
 */
public class TestClassROMISAccess
{
    //-------------------------
    // Public static functions
    //-------------------------
    /*==========================================================================
     * = FUNCTION NAME
     * select
     *
     * = DESCRIPTION
     */
    /**
     * Selects an array of database objects from Oracle.
     */
    /*==========================================================================
     */
    public static TestClassData[] select(String selectString)
        throws SQLException, DataException
    {
        Trace.startBlock(Trace.LEVEL5, "TestClassROMISAccess::select");

        ResultSet resultSet;
        ConnectionManager connectionManager = ConnectionManager.getInstance();
        Statement statement = connectionManager.obtainStatement();

        try
        {
            resultSet = statement.executeQuery(selectString);
        }
        catch (SQLException sqle)
        {
            connectionManager.releaseStatement(statement);
            throw sqle;
        }
        int size = 0;
        TestClassData data;
        TestClassData[] dataArray;
        Vector vector = new Vector(256, 256);

        while (resultSet.next())
        {
            data = new TestClassData();

            data.intValue = resultSet.getInt("INT_VALUE");
             data.booleanValue = ROMISHelper.getBoolean(resultSet, "BOOLEAN_VALUE");
             data.floatValue = resultSet.getFloat("FLOAT_VALUE");
             data.shortValue = resultSet.getShort("SHORT_VALUE");
             data.doubleValue = resultSet.getDouble("DOUBLE_VALUE");
             data.dateValue = ROMISHelper.getDate(resultSet, "DATE_VALUE");
             data.stringValue = ROMISHelper.getString(resultSet, "STRING_VALUE", 10);
             data.stringValueArray1Dim = ROMISHelper.getStringArray1Dim(resultSet,
"STRING_VALUE_ARRAY1_DIM", 4);
             data.stringValueArray2Dim = ROMISHelper.getStringArray2Dim(resultSet,
"STRING_VALUE_ARRAY2_DIM", 4, 5);
             data.intValueArray1Dim = ROMISHelper.getIntArray1Dim(resultSet,
"INT_VALUE_ARRAY1_DIM", 5);
             data.intValueArray2Dim = ROMISHelper.getIntArray2Dim(resultSet,
"INT_VALUE_ARRAY2_DIM", 4, 5);
             data.intValueArray3Dim = ROMISHelper.getIntArray3Dim(resultSet,
"INT_VALUE_ARRAY3_DIM", 4, 5, 6);
             data.floatValueArray1Dim = ROMISHelper.getFloatArray1Dim(resultSet,
"FLOAT_VALUE_ARRAY1_DIM", 6);
             data.floatValueArray2Dim = ROMISHelper.getFloatArray2Dim(resultSet,
"FLOAT_VALUE_ARRAY2_DIM", 4, 5);
             data.floatValueArray3Dim = ROMISHelper.getFloatArray3Dim(resultSet,
"FLOAT_VALUE_ARRAY3_DIM", 4, 5, 6);
             data.shortValueArray1Dim = ROMISHelper.getShortArray1Dim(resultSet,
"SHORT_VALUE_ARRAY1_DIM", 7);
             data.shortValueArray2Dim = ROMISHelper.getShortArray2Dim(resultSet,
"SHORT_VALUE_ARRAY2_DIM", 4, 5);
             data.shortValueArray3Dim = ROMISHelper.getShortArray3Dim(resultSet,
"SHORT_VALUE_ARRAY3_DIM", 4, 5, 6);
             data.doubleValueArray1Dim = ROMISHelper.getDoubleArray1Dim(resultSet,
"DOUBLE_VALUE_ARRAY1_DIM", 8);
             data.doubleValueArray2Dim = ROMISHelper.getDoubleArray2Dim(resultSet,
"DOUBLE_VALUE_ARRAY2_DIM", 4, 5);
             data.doubleValueArray3Dim = ROMISHelper.getDoubleArray3Dim(resultSet,
"DOUBLE_VALUE_ARRAY3_DIM", 4, 5, 6);
             data.dateValueArray1Dim = ROMISHelper.getDateArray1Dim(resultSet,
"DATE_VALUE_ARRAY1_DIM", 4);
             data.dateValueArray2Dim = ROMISHelper.getDateArray2Dim(resultSet,
"DATE_VALUE_ARRAY2_DIM", 4, 5);
             data.dateValueArray3Dim = ROMISHelper.getDateArray3Dim(resultSet,
"DATE_VALUE_ARRAY3_DIM", 4, 5, 6);


            vector.insertElementAt(data, size);
            size++;
        }
        connectionManager.releaseStatement(statement);
        dataArray = new TestClassData[size];
        vector.copyInto(dataArray);

        Trace.endBlock(Trace.LEVEL5, "TestClassROMISAccess::select");
        return dataArray;
    }

    /*=========================================================================
     * = FUNCTION NAME
     * select
     *
     * - DESCRIPTION
     */
    /**
     * Inserts an array of database objects into Oracle.
     */
    /*=========================================================================
     */
    public static void insert(TestClassData[] dataArray)
        throws SQLException, DataException
    {
        Trace.startBlock(Trace.LEVEL5, "TestClassROMISAccess::insert");

        int rows = 0;
        ConnectionManager connectionManager = ConnectionManager.getInstance();
        Statement statement = connectionManager.obtainStatement();
        StringBuffer sbuf;
        String queryString;

        for (int i = 0; i < dataArray.length; i++)
        {
            sbuf = new StringBuffer(10000);
            sbuf.append("INSERT INTO TEST_CLASS (");

            sbuf.append("INT_VALUE");
             sbuf.append(", ");
             sbuf.append("BOOLEAN_VALUE");
             sbuf.append(", ");
             sbuf.append("FLOAT_VALUE");
             sbuf.append(", ");
             sbuf.append("SHORT_VALUE");
             sbuf.append(", ");
             sbuf.append("DOUBLE_VALUE");
             sbuf.append(", ");
             sbuf.append("DATE_VALUE");
             sbuf.append(", ");
             sbuf.append("STRING_VALUE");
             sbuf.append(", ");
             sbuf.append("STRING_VALUE_ARRAY1_DIM");
             sbuf.append(", ");
             sbuf.append("STRING_VALUE_ARRAY2_DIM");
             sbuf.append(", ");
             sbuf.append("INT_VALUE_ARRAY1_DIM");
             sbuf.append(", ");
             sbuf.append("INT_VALUE_ARRAY2_DIM");
             sbuf.append(", ");
             sbuf.append("INT_VALUE_ARRAY3_DIM");
             sbuf.append(", ");
             sbuf.append("FLOAT_VALUE_ARRAY1_DIM");
             sbuf.append(", ");
             sbuf.append("FLOAT_VALUE_ARRAY2_DIM");
             sbuf.append(", ");
             sbuf.append("FLOAT_VALUE_ARRAY3_DIM");
             sbuf.append(", ");
             sbuf.append("SHORT_VALUE_ARRAY1_DIM");
             sbuf.append(", ");
             sbuf.append("SHORT_VALUE_ARRAY2_DIM");
             sbuf.append(", ");
             sbuf.append("SHORT_VALUE_ARRAY3_DIM");
             sbuf.append(", ");
             sbuf.append("DOUBLE_VALUE_ARRAY1_DIM");
             sbuf.append(", ");
             sbuf.append("DOUBLE_VALUE_ARRAY2_DIM");
             sbuf.append(", ");
             sbuf.append("DOUBLE_VALUE_ARRAY3_DIM");
             sbuf.append(", ");
             sbuf.append("DATE_VALUE_ARRAY1_DIM");
             sbuf.append(", ");
             sbuf.append("DATE_VALUE_ARRAY2_DIM");
             sbuf.append(", ");
             sbuf.append("DATE_VALUE_ARRAY3_DIM");


            sbuf.append(") VALUES (");

            sbuf.append(dataArray[i].intValue);
             sbuf.append(", ");

sbuf.append(ROMISHelper.convertBooleanToSQLString(dataArray[i].booleanValue));
             sbuf.append(", ");
             sbuf.append(dataArray[i].floatValue);
```

```
                sbuf.append(", ");
                sbuf.append(dataArray[i].shortValue);
                sbuf.append(", ");
                sbuf.append(dataArray[i].doubleValue);
                sbuf.append(", ");
                sbuf.append(ROMISHelper.convertDateToSQLString(dataArray[i].dateValue));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertStringToSQLString(dataArray[i].stringValue, 10));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertString1DToSQLString(dataArray[i].stringValueArray1Dim,
10, 4));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertString2DToSQLString(dataArray[i].stringValueArray2Dim,
10, 4, 5));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertInt1DToSQLString(dataArray[i].intValueArray1Dim, 5));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertInt2DToSQLString(dataArray[i].intValueArray2Dim, 4,
5));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertInt3DToSQLString(dataArray[i].intValueArray3Dim, 4, 5,
6));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertFloat1DToSQLString(dataArray[i].floatValueArray1Dim,
6));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertFloat2DToSQLString(dataArray[i].floatValueArray2Dim, 4,
5));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertFloat3DToSQLString(dataArray[i].floatValueArray3Dim, 4,
5, 6));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertShort1DToSQLString(dataArray[i].shortValueArray1Dim,
7));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertShort2DToSQLString(dataArray[i].shortValueArray2Dim, 4,
5));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertShort3DToSQLString(dataArray[i].shortValueArray3Dim, 4,
5, 6));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertDouble1DToSQLString(dataArray[i].doubleValueArray1Dim,
8));
                sbuf.append(", ");

sbuf.append(ROMISHelper.convertDouble2DToSQLString(dataArray[i].doubleValueArray2Dim,
4, 5));
             sbuf.append(", ");

sbuf.append(ROMISHelper.convertDouble3DToSQLString(dataArray[i].doubleValueArray3Dim,
4, 5, 6));
             sbuf.append(", ");

sbuf.append(ROMISHelper.convertDate1DToSQLString(dataArray[i].dateValueArray1Dim, 4));
             sbuf.append(", ");

sbuf.append(ROMISHelper.convertDate2DToSQLString(dataArray[i].dateValueArray2Dim, 4,
5));
             sbuf.append(", ");

sbuf.append(ROMISHelper.convertDate3DToSQLString(dataArray[i].dateValueArray3Dim, 4,
5, 6));


            sbuf.append(")");

            queryString = new String(sbuf);
            Trace.debug(Trace.LEVEL6, "The query string is: ", queryString);

            try
            {
                rows += statement.executeUpdate(queryString);
            }
            catch (SQLException sqle)
            {
                connectionManager.releaseStatement(statement);
                throw sqle;
            }
        }
        connectionManager.releaseStatement(statement);
        if (rows != dataArray.length)
        {
            throw new DataStorageException("Not all objects were successfully
inserted");
        }

        Trace.endBlock(Trace.LEVEL5, "TestClassROMISAccess::insert");
    }

    /*===========================================================================
     * - FUNCTION NAME
     * update
     *
     * - DESCRIPTION
     */
    /**
     * Updates an array of database objects in Oracle.
     */
    /*===========================================================================
     */
    public static void update(TestClassData[] dataArray)
        throws SQLException, DataException
    {
        Trace.startBlock(Trace.LEVEL6, "TestClassROMISAccess::update");
        int rows = 0;
        ConnectionManager connectionManager = ConnectionManager.getInstance();
        Statement statement = connectionManager.obtainStatement();
        StringBuffer sbuf;
        String queryString;

        for (int i = 0; i < dataArray.length; i++)
        {
            sbuf = new StringBuffer(10000);

            sbuf.append("UPDATE TEST_CLASS SET ");

            sbuf.append("INT_VALUE = ");
             sbuf.append(dataArray[i].intValue);
             sbuf.append(", ");
             sbuf.append("BOOLEAN_VALUE = ");

sbuf.append(ROMISHelper.convertBooleanToSQLString(dataArray[i].booleanValue));
             sbuf.append(", ");
             sbuf.append("FLOAT_VALUE = ");
             sbuf.append(dataArray[i].floatValue);
             sbuf.append(", ");
             sbuf.append("SHORT_VALUE = ");
             sbuf.append(dataArray[i].shortValue);
             sbuf.append(", ");
             sbuf.append("DOUBLE_VALUE = ");
             sbuf.append(dataArray[i].doubleValue);
             sbuf.append(", ");
             sbuf.append("DATE_VALUE = ");
             sbuf.append(ROMISHelper.convertDateToSQLString(dataArray[i].dateValue));
             sbuf.append(", ");
             sbuf.append("STRING_VALUE = ");

sbuf.append(ROMISHelper.convertStringToSQLString(dataArray[i].stringValue, 10));
             sbuf.append(", ");
             sbuf.append("STRING_VALUE_ARRAY1_DIM = ");

sbuf.append(ROMISHelper.convertString1DToSQLString(dataArray[i].stringValueArray1Dim,
10, 4));
             sbuf.append(", ");
             sbuf.append("STRING_VALUE_ARRAY2_DIM = ");

sbuf.append(ROMISHelper.convertString2DToSQLString(dataArray[i].stringValueArray2Dim,
10, 4, 5));
             sbuf.append(", ");
             sbuf.append("INT_VALUE_ARRAY1_DIM = ");

sbuf.append(ROMISHelper.convertInt1DToSQLString(dataArray[i].intValueArray1Dim, 5));
             sbuf.append(", ");
             sbuf.append("INT_VALUE_ARRAY2_DIM = ");

sbuf.append(ROMISHelper.convertInt2DToSQLString(dataArray[i].intValueArray2Dim, 4,
5));
             sbuf.append(", ");
             sbuf.append("INT_VALUE_ARRAY3_DIM = ");

sbuf.append(ROMISHelper.convertInt3DToSQLString(dataArray[i].intValueArray3Dim, 4, 5,
6));
             sbuf.append(", ");
            sbuf.append("FLOAT_VALUE_ARRAY1_DIM = ");

sbuf.append(ROMISHelper.convertFloat1DToSQLString(dataArray[i].floatValueArray1Dim,
6));
            sbuf.append(", ");
            sbuf.append("FLOAT_VALUE_ARRAY2_DIM = ");

sbuf.append(ROMISHelper.convertFloat2DToSQLString(dataArray[i].floatValueArray2Dim, 4,
5));
            sbuf.append(", ");
            sbuf.append("FLOAT_VALUE_ARRAY3_DIM = ");

sbuf.append(ROMISHelper.convertFloat3DToSQLString(dataArray[i].floatValueArray3Dim, 4,
5, 6));
            sbuf.append(", ");
            sbuf.append("SHORT_VALUE_ARRAY1_DIM = ");

sbuf.append(ROMISHelper.convertShort1DToSQLString(dataArray[i].shortValueArray1Dim,
7));
            sbuf.append(", ");
            sbuf.append("SHORT_VALUE_ARRAY2_DIM = ");

sbuf.append(ROMISHelper.convertShort2DToSQLString(dataArray[i].shortValueArray2Dim, 4,
5));
            sbuf.append(", ");
            sbuf.append("SHORT_VALUE_ARRAY3_DIM = ");

sbuf.append(ROMISHelper.convertShort3DToSQLString(dataArray[i].shortValueArray3Dim, 4,
5, 6));
            sbuf.append(", ");
            sbuf.append("DOUBLE_VALUE_ARRAY1_DIM = ");

sbuf.append(ROMISHelper.convertDouble1DToSQLString(dataArray[i].doubleValueArray1Dim,
8));
            sbuf.append(", ");
            sbuf.append("DOUBLE_VALUE_ARRAY2_DIM = ");

sbuf.append(ROMISHelper.convertDouble2DToSQLString(dataArray[i].doubleValueArray2Dim,
4, 5));
            sbuf.append(", ");
            sbuf.append("DOUBLE_VALUE_ARRAY3_DIM = ");

sbuf.append(ROMISHelper.convertDouble3DToSQLString(dataArray[i].doubleValueArray3Dim,
4, 5, 6));
            sbuf.append(", ");
            sbuf.append("DATE_VALUE_ARRAY1_DIM = ");

sbuf.append(ROMISHelper.convertDate1DToSQLString(dataArray[i].dateValueArray1Dim, 4));
            sbuf.append(", ");
            sbuf.append("DATE_VALUE_ARRAY2_DIM = ");

sbuf.append(ROMISHelper.convertDate2DToSQLString(dataArray[i].dateValueArray2Dim, 4,
5));
            sbuf.append(", ");
            sbuf.append("DATE_VALUE_ARRAY3_DIM = ");

sbuf.append(ROMISHelper.convertDate3DToSQLString(dataArray[i].dateValueArray3Dim, 4,
5, 6));
```

```
            sbuf.append(" where INT_VALUE = " + dataArray[i].intValue + " and
DATE_VALUE = " + ROMISHelper.convertDateToSQLString(dataArray[i].dateValue));

            queryString = sbuf.toString();
            Trace.debug(Trace.LEVEL6, "The query string is: ", queryString);

            try
            {
                rows += statement.executeUpdate(queryString);
            }
            catch (SQLException sqle)
            {
                connectionManager.releaseStatement(statement);
                throw sqle;
            }
        }
        connectionManager.releaseStatement(statement);
        if (rows != dataArray.length)
        {
            throw new DataStorageException("Not all objects were successfully
updated");
        }

        Trace.endBlock(Trace.LEVEL6, "TestClassROMISAccess::update");
    }

    /*==========================================================================
     * = FUNCTION NAME
     * remove
     *
     * = DESCRIPTION
     */
    /**
     * Removes a number of database objects from Oracle.
     */
    /*==========================================================================
     */
    public static int remove(String sqlString)
        throws SQLException
    {
        Trace.startBlock(Trace.LEVEL6, "TestClassROMISAccess::remove");

        int rows = 0;
        ConnectionManager connectionManager = ConnectionManager.getInstance();
        Statement statement = connectionManager.obtainStatement();
        try
        {
            rows = statement.executeUpdate(sqlString);
        }
        catch (SQLException sqle)
        {
            connectionManager.releaseStatement(statement);
            throw sqle;
        }
        connectionManager.releaseStatement(statement);

        Trace.endBlock(Trace.LEVEL6, "TestClassROMISAccess::remove");
        return rows;
    }
}
```

6.5.4 Auto generated table creation script that maps the java class data members.

```
--===========================================================================
--
--===========================================================================
-- SYSTEM:      Hot Strip Mill Data Logger
-- SUBSYSTEM:   ROMIS
-- FILE NAME:   createTestClass.sql
-- DESCRIPTION: Creates table and types associated with TestClass object
-- AUTHOR:      Saugato Mukerji
-- CREATED:     Mon Oct 11 13:02:01 1999

-- USAGE:
-- INPUTS:
-- OUTPUTS:
-- NOTES:
--
-- REVISION HISTORY:
-- $Author$
-- $Id$
--
-- $Log$
--
--==========================================================================

PROMPT *******************************************
PROMPT * executing createTestClass.sql script
PROMPT *******************************************

CREATE TYPE STRING_10_ARR_4 AS VARRAY(4) OF VARCHAR2( 10 );
/
CREATE TYPE STRING_10_ARR_20 AS VARRAY(20) OF VARCHAR2( 10 );
/
CREATE TYPE NUMBER_ARR_5 AS VARRAY(5) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_20 AS VARRAY(20) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_120 AS VARRAY(120) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_6 AS VARRAY(6) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_7 AS VARRAY(7) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_8 AS VARRAY(8) OF NUMBER;
/
CREATE TYPE DATE_ARR_4 AS VARRAY(4) OF NUMBER;
/
CREATE TYPE DATE_ARR_20 AS VARRAY(20) OF NUMBER;
/
CREATE TYPE DATE_ARR_120 AS VARRAY(120) OF NUMBER;
/

CREATE TYPE TEST_CLASS_T AS OBJECT
(
    INT_VALUE NUMBER,
    BOOLEAN_VALUE VARCHAR2( 5 ),
    FLOAT_VALUE NUMBER,
    SHORT_VALUE NUMBER,
    DOUBLE_VALUE NUMBER,
    DATE_VALUE NUMBER,
    STRING_VALUE VARCHAR2( 10 ),
    STRING_VALUE_ARRAY1_DIM STRING_10_ARR_4,
    STRING_VALUE_ARRAY2_DIM STRING_10_ARR_20,
    INT_VALUE_ARRAY1_DIM NUMBER_ARR_5,
    INT_VALUE_ARRAY2_DIM NUMBER_ARR_20,
    INT_VALUE_ARRAY3_DIM NUMBER_ARR_120,
    FLOAT_VALUE_ARRAY1_DIM NUMBER_ARR_6,
    FLOAT_VALUE_ARRAY2_DIM NUMBER_ARR_20,
    FLOAT_VALUE_ARRAY3_DIM NUMBER_ARR_120,
    SHORT_VALUE_ARRAY1_DIM NUMBER_ARR_7,
    SHORT_VALUE_ARRAY2_DIM NUMBER_ARR_20,
    SHORT_VALUE_ARRAY3_DIM NUMBER_ARR_120,
    DOUBLE_VALUE_ARRAY1_DIM NUMBER_ARR_8,
    DOUBLE_VALUE_ARRAY2_DIM NUMBER_ARR_20,
    DOUBLE_VALUE_ARRAY3_DIM NUMBER_ARR_120,
    DATE_VALUE_ARRAY1_DIM DATE_ARR_4,
    DATE_VALUE_ARRAY2_DIM DATE_ARR_20,
    DATE_VALUE_ARRAY3_DIM DATE_ARR_120
);
/

CREATE TABLE TEST_CLASS OF TEST_CLASS_T
tablespace AUTO_GEN;
ALTER TABLE TEST_CLASS
ADD (CONSTRAINT TEST_CLASS_PK PRIMARY KEY (
    INT_VALUE,
    DATE_VALUE
));
CREATE INDEX TEST_CLASS__I2 ON TEST_CLASS(
    DATE_VALUE
);

exit;
```

6.6 Scripts related to section 3.2.6.

Scripts Associated with section 3.2.6

The following scripts were used to create the TESTCLASS table and gather the experimental data

RunAndAnalyse.bat
analyseClass.sql
timedTestClassInsert.sql
timedTestClassInsertNull.sql

Reproduced here are contents of the scripts

```
>type testClass.sql
--
=======================================================================
-- Copyright 1996 BHP IT All Rights Limited
=======================================================================
-- SYSTEM:       Plate Rolling Mill Control System
-- SUBSYSTEM:    ROMIS
-- FILE NAME:    testClass.sql
-- DESCRIPTION:  Generates TestClass object in Oracle database.
-- AUTHOR:       Saugato Mukerji
-- CREATED:
-- USAGE:
-- INPUTS:
-- OUTPUTS:
-- NOTES:
--
-- REVISION HISTORY:
-- $Author$
-- $Id$
--
-- $Log$
--
--
=======================================================================

PROMPT *******************************************
PROMPT * executing TestClass.sql script
PROMPT *******************************************


DROP TABLE TestClass ;
DROP TYPE TestClass_T ;
DROP TYPE NUMBER_ARR_5;
DROP TYPE NUMBER_ARR_6;
DROP TYPE NUMBER_ARR_7;
DROP TYPE NUMBER_ARR_8;
DROP TYPE NUMBER_ARR_24;

CREATE TYPE NUMBER_ARR_5 AS VARRAY( 5 ) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_6 AS VARRAY( 6 ) OF NUMBER;
```

```
/
CREATE TYPE NUMBER_ARR_7 AS VARRAY( 7 ) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_8 AS VARRAY( 8 ) OF NUMBER;
/
CREATE TYPE NUMBER_ARR_24 AS VARRAY( 24 ) OF NUMBER;
/
CREATE TYPE DATE_ARR_4 AS VARRAY( 4 ) OF DATE;
/

CREATE TYPE TestClass_T AS OBJECT
(
      intValue NUMBER,
      floatValue NUMBER,
      shortValue NUMBER,
      doubleValue NUMBER,
      stringValue VARCHAR( 10 ),
      dateValue DATE,
      stringValueArray1Dim VARCHAR( 40 ),
      intValueArray1Dim NUMBER_ARR_5,
      floatValueArray1Dim NUMBER_ARR_6,
      shortValueArray1Dim NUMBER_ARR_7,
      doubleValueArray1Dim NUMBER_ARR_8,
      intValueArray2Dim NUMBER_ARR_6,
      intValueArray3Dim NUMBER_ARR_24,
      dateValueArray1Dim DATE_ARR_4
);
/


CREATE TABLE TestClass OF TestClass_T;

//=============================================================================

>type timedTestClassInsert.sql

set serveroutput on
--set feedback off

DECLARE
      STATUS NUMBER(10);
      MESSAGE VARCHAR2(80);
      ITEM_COUNT NUMBER;
      start_time  CHAR(5);
      end_time    CHAR(5);
      aname   CHAR(30);
      X   NUMBER;
      numRows NUMBER;

BEGIN

    ITEM_COUNT := 10000;


    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
```

```
         INSERT INTO SCOTT.TESTCLASS
             VALUES ( 5,
             1.1,
       200,
       400.4,
       'hello',
       sysdate,
       'qwertyuiopasdfghjklzxcvbnm',
       SCOTT.NUMBER_ARR_5(101, 201, 301, 401, 501),
       SCOTT.NUMBER_ARR_6(1.1,1.2,1.3,1.4,1.5,1.6666),
       SCOTT.NUMBER_ARR_7(11,12,13,14,15,16,17),
       SCOTT.NUMBER_ARR_8(2.1,2.2,2.3,2.4,2.5,2.6,2.7,2.8),
       SCOTT.NUMBER_ARR_6(101,102,103,104,105,106),
             SCOTT.NUMBER_ARR_24(    1,2,3,4,5,6,7,8,
                                     9,10,11,12,13,14,15,16,
                                     17,18,19,20,21,22,23,24),
       SCOTT.DATE_ARR_4(SYSDATE,SYSDATE,SYSDATE,SYSDATE) );
    END LOOP;

      SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
            -- time the end of the 1000 inserts


      select count(*) INTO numRows from SCOTT.TESTCLASS;

      DBMS_OUTPUT.PUT_LINE( 'owner'
                                      ||' '||'tablename'
                                      ||' '||'start_time'
                                      ||' '||'end_time'
                                      ||' '||'numRows' );

      DBMS_OUTPUT.PUT_LINE(     'scott'
                                            ||' '||'TESTCLASS'
                                            ||' '||start_time
                                            ||' '||end_time
                                            ||' '||numRows );

END;

/

exit;

//=========================================================

>type  timedTestClassInsertNull.sql

set serveroutput on
--set feedback off

DECLARE
      STATUS NUMBER(10);
      MESSAGE VARCHAR2(80);
      ITEM_COUNT NUMBER;
      start_time   CHAR(5);
      end_time     CHAR(5);
      aname    CHAR(30);
      X   NUMBER;
      numRows NUMBER;

BEGIN

    ITEM_COUNT := 10000;


    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
         INSERT INTO SCOTT.TESTCLASS
         (intvalue)
             VALUES ( J );
    END LOOP;

      SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
            -- time the end of the 1000 inserts


      select count(*) INTO numRows from SCOTT.TESTCLASS;

      DBMS_OUTPUT.PUT_LINE( 'owner'
                                         ||' '||'tablename'
                                         ||' '||'start_time'
                                         ||' '||'end_time'
                                         ||' '||'numRows' );

      DBMS_OUTPUT.PUT_LINE(     'scott'
                                               ||' '||'TESTCLASS'
                                               ||' '||start_time
                                               ||' '||end_time
                                               ||' '||numRows );

END;

/

--exit;
```

//===

```
>type runandanalyse.bat

SQLPLUS scott/tiger @TIMEDTESTCLASSINSERT
SQLPLUS SYSTEM/MANAGER @ANALYSECLASS scott TESTCLASS
```

//===

```
>type analyseClass.sql

-- Name analyseClass.sql tableowner TABLENAME
-- NOTE THE CASE OF THE TABLENAME MUST BE CORRECT
-- THE DBA_SEGMENTS TABLE IS ONLY VISIBLE TO SYSTEM OR SYS
-- OR SVRMGRL
--

 ANALYZE TABLE &1..&2 COMPUTE STATISTICS;

 SELECT COUNT(*) FROM &1..&2;

 SELECT OWNER , SEGMENT_NAME, EXTENTS , BYTES, BLOCKS,
INITIAL_EXTENT, MAX_EXTENTS
                    FROM DBA_SEGMENTS
                    WHERE SEGMENT_NAME='&2';


//
```

6.7 Vendor benchmarks related to 3.4.4

Vendor benchmark data related to 3.4.4

Improvements in Disk I/O performance with Advanced Cache Mechanisms

Presented here are benchmarks provided by the vendor to show how impressive disk I/O performance can be extracted by using specialised cacheing Software coupled with a fast multiprocessor system and a good SCSI drive. Infact the measured benchmarks of 70Mb/sec indicate that it is possible to achieve shared memory like performance especially where the data size of each read/write is large.The I/O rate of 8800 - 10000 calls/second is an equally impressive figure. This is more than adequate for many Automation situations.

The response time is however is significantly lower at an average of 1ms and worst case of 2ms. This will mean the developers will have to code sensibly and read the data into their local memory manipulate and then write back. They will not have the luxury of using the shared memory as a scratchpad or accessing the shared memory inside loops of > 20 iterations without crippling performance. It is worth observing that the vendor used SCSI disk and multithreading with 4 worker threads to improve the throughput.

The platform was an Intel Pentium II/Xeon 450MHz server with 4 CPUs and 8GB of RAM. The hard disk on the system was an IBM 8GB SCSI drive. The SuperCache partition size was set to 1.7 GB.

However it may be worth noting is that the impressive hardware used for this benchmark is not cheap the price at this time is estimated to be in the USD30000-40000 region.

In our opinion this technology may still be a viable as a means of implementing shared memory between co operating applications where the project size already justifies the hardware investment. Though the best application of this caching technology is in situations where large data size and a high number of separate IO calls/second occur together. An example of this may be a Internet Web Server which has to sustain a high hit rate when serving multimedia data to many concurrent users.

Given below are test results reported by the vendor of the SuperCache-NT disk caching product.

EEC Iometer Results Using SuperCache-NT/Enterprise Edition

What is Iometer?

The Iometer benchmark is a new I/O performance analysis tool forservers developed by Intel. It measures system I/O performance while stressing the system with a controlled workload. A full description is available at http://developer.intel.com/design/servers/devtools/iometer/index.htm.

The Iometer benchmark enabled EEC to quantitatively test the effects of SuperCache on system performance. Tests under a range of workloads were conducted both with and without SuperCache. The results showed that performance is dramatically accelerated
with SuperCache.

SuperCache Set Up and Results

As the results below show, SuperCache provided speedups which were often greater than 100 times on the benchmark system. The system used for this test was an Intel Pentium II/Xeon 450MHz server with 4 CPUs and 8GB of RAM. The SuperCache partition size was set to 1.7 GB. The hard disk on the system was an IBM 8GB SCSI drive.

The Iometer test was configured with four worker threads. The outstanding queue depth was set to eight. Transfer size was 8Kb and I/O was set to 100% random. The relationship of reads to writes was varied from 90%-10% to 0%-100% yielding the results below.

90% Reads, 10% Writes

	With SuperCache	**Without SuperCache**	**Speedup with SuperCache**
I/O per second	8781.28	111.4	78.8
MB/s	68.69	0.87	79
response time(ms)			
average	0.919	44.6	48.5
maximum	1.956	206.38	105.5

80% Reads, 20% Writes

	With SuperCache	**Without SuperCache**	**Speedup with SuperCache**
I/O per second	8833.76	109.4	80.7
MB/s	69.04	0.86	80.3
response time(ms)			
average	0.9155	45.59	49.8
maximum	1.89	248.8	131.6

50% Reads, 50% Writes

	With SuperCache	**Without SuperCache**	**Speedup with SuperCache**
I/O per second	9066.2	107	84.7
MB/s	70.87	0.84	84.4
response time(ms)			
average	0.9073	37.44	41.3

maximum	2.041	109.07	53.4

0% Reads, 100% Writes

	With SuperCache	Without SuperCache	Speedup with SuperCache
I/O per second	10203.1	102.6	99.4
MB/s	79.69	0.8	99.6
response time(ms)			
average	0.81	57	70.4
maximum	1.88	217	115.4

6.8 Code for testing JDBC performance. Ref 3.4.1

```
/>type run.bat

@echo off
rem *************************
rem Setup java home directory
rem *************************
SET JAVA_HOME=d:\jdk1.2.1

rem *******************
rem Setup java bin path
rem *******************
PATH=%JAVA_HOME%\bin;%PATH%
set CLASSPATH=e:\Oracle\JDBC\lib\classes111.zip;.

java -cp .;%CLASSPATH%;.  Employee

pause




/>type run2.bat

start run
start run
pause


/>type run3.bat

start run
start run
start run
pause


/>type run5.bat

start run
start run
start run
start run
start run
pause
```

```
/*
 * This sample shows how to list all the names from the EMP table
 *
 * It uses the JDBC THIN driver.  See the same program in the
 * oci7 or oci8 samples directories to see how to use the other drivers.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

class Employee
{
    // The connect string
    static final String connect_string =
                  "jdbc:oracle:thin:scott/tiger@fast:1521:oraglob";

    //static final String connect_string =
    //              "jdbc:oracle:thin:scott/tiger@home:1521:ORCL";

    static ResultSet rset;
    PreparedStatement anInsertStatement;

    int salary = 1000;

    public static void main (String args [])
       throws SQLException, ClassNotFoundException
    {
        // Load the Oracle JDBC driver
        Class.forName ("oracle.jdbc.driver.OracleDriver");

        // Connect to the database
        // You must put a database name after the @ sign in the connection URL.
        // You can use either the fully specified SQL*net syntax or a short cut
        // syntax as <host>:<port>:<sid>.  The example uses the short cut syntax.

        // System.out.println("time after="+System.currentTimeMillis() );

        Employee emp = new Employee();

        long startTime = System.currentTimeMillis();
        int loopCount = 10000;
        for( int i=0; i<loopCount; i++ )
        {
            emp.send();
        }
        long endTime = System.currentTimeMillis();

        long delta = endTime - startTime;

        System.out.println( "rate in updates/second ="+(long)( (1000*loopCount)/delta)
);

    }

    void send()
    {
        try
        {
            // System.out.print(".");
            if( anInsertStatement == null )
            {
                System.out.println("Connecting ......");
                // Load the Oracle JDBC driver
                Class.forName ("oracle.jdbc.driver.OracleDriver");
```

```
                 // Connect to the database
                 // You must put a database name after the @ sign in the connection
URL.
                 // You can use either the fully specified SQL*net syntax or a short
cut
                 // syntax as <host>:<port>:<sid>.  The example uses the short cut
syntax.


                 Connection conn =
                 DriverManager.getConnection (connect_string);

                 conn.setAutoCommit( true );

                 System.out.println("got connection......");

                 anInsertStatement = conn.prepareStatement
                 ("update emp set sal=? where empno = 7934" );

             }
             salary++;
             anInsertStatement.setInt(1, salary );
                 // write timestamp value into the first column
             anInsertStatement.execute();

         }
         catch( java.sql.SQLException e)
         {
             System.out.println( "Exception :"+e);
         }
         catch( ClassNotFoundException e)
         {
             System.out.println( "Exception :"+e);
         }

     }
 }
```

```
/*
 * This sample shows how to list all the names from the EMP table
 *
 * It uses the JDBC THIN driver.  See the same program in the
 * oci7 or oci8 samples directories to see how to use the other drivers.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

class Employee
{
    // The connect string
    static final String connect_string =
                  "jdbc:oracle:thin:scott/tiger@fast:1521:oraglob";

    //static final String connect_string =
    //              "jdbc:oracle:thin:scott/tiger@home:1521:ORCL";

    static ResultSet rset;
    PreparedStatement anInsertStatement;

    int salary = 1000;

    public static void main (String args [])
       throws SQLException, ClassNotFoundException
    {
        // Load the Oracle JDBC driver
        Class.forName ("oracle.jdbc.driver.OracleDriver");

        // Connect to the database
        // You must put a database name after the @ sign in the connection URL.
        // You can use either the fully specified SQL*net syntax or a short cut
        // syntax as <host>:<port>:<sid>.  The example uses the short cut syntax.

        // System.out.println("time after="+System.currentTimeMillis() );

        Employee emp = new Employee();

        long startTime = System.currentTimeMillis();
        int loopCount = 10000;
        for( int i=0; i<loopCount; i++ )
        {
            emp.send();
        }
        long endTime = System.currentTimeMillis();

        long delta = endTime - startTime;

        System.out.println( "rate in updates/second ="+(long)( (1000*loopCount)/delta)
);

    }

    void send()
    {
        try
        {
            // System.out.print(".");
            if( anInsertStatement == null )
            {
                System.out.println("Connecting ......");
                // Load the Oracle JDBC driver
                Class.forName ("oracle.jdbc.driver.OracleDriver");
                // Connect to the database
                // You must put a database name after the @ sign in the connection
URL.
                // You can use either the fully specified SQL*net syntax or a short
cut
                // syntax as <host>:<port>:<sid>.  The example uses the short cut
syntax.


                Connection conn =
                DriverManager.getConnection (connect_string);

                conn.setAutoCommit( true );

                System.out.println("got connection......");

                anInsertStatement = conn.prepareStatement
                ("update emp set sal=? where empno = 7934" );

            }
            salary++;
            anInsertStatement.setInt(1, salary );
                // write timestamp value into the first column
            anInsertStatement.execute();

        }
        catch( java.sql.SQLException e)
        {
            System.out.println( "Exception :"+e);
        }
        catch( ClassNotFoundException e)
        {
            System.out.println( "Exception :"+e);
        }

    }
}
```

6.9 PLSQL Scripts and batch files for testing performance with multiple concurrent processes Ref 3.4.1

```
/>type runsel.bat

sqlplus scott/tiger@oraglob @nls

/>type runsel2proc.bat

start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls

..
..
..
/>type runsel5proc.bat

start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls



/>type runsel2procDiffRow.bat

start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls1
..
..
..
/>type runsel5procDiffRow.bat

start sqlplus scott/tiger@oraglob @nls
start sqlplus scott/tiger@oraglob @nls1
start sqlplus scott/tiger@oraglob @nls2
start sqlplus scott/tiger@oraglob @nls3
start sqlplus scott/tiger@oraglob @nls4



/>type runupd.bat

sqlplus scott/tiger@oraglob @nlu


/>type runupd2procDiffRow.bat

start sqlplus scott/tiger@oraglob @nlu
start sqlplus scott/tiger@oraglob @nlu1
..
..
..

/>type runupd5procDiffRow.bat

start sqlplus scott/tiger@oraglob @nlu
start sqlplus scott/tiger@oraglob @nlu1
start sqlplus scott/tiger@oraglob @nlu2
start sqlplus scott/tiger@oraglob @nlu3
start sqlplus scott/tiger@oraglob @nlu4
```

```
/>type nls.bat

--==============================================================
-- Name nls.sql
-- Description a test to determine the select performance
--
--==============================================================

set serveroutput on


DECLARE
    STATUS NUMBER(10);
    MESSAGE VARCHAR2(80);
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname   CHAR(30);
    X   NUMBER;
BEGIN

    ITEM_COUNT := 10000;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
            SELECT ENAME INTO aname FROM EMP WHERE EMPNO = 7369;
    END LOOP;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                          ' loopcount='||ITEM_COUNT||' aname='||aname );
END;

/


/>type nls1.bat

--==============================================================
-- Name nls1.sql
-- Description a test to determine the select performance
--
--==============================================================

set serveroutput on


DECLARE
    STATUS NUMBER(10);
    MESSAGE VARCHAR2(80);
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname   CHAR(30);
    X   NUMBER;
BEGIN

    ITEM_COUNT := 10000;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
            SELECT ENAME INTO aname FROM EMP WHERE EMPNO = 7782;
    END LOOP;
    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                         ' loopcount='||ITEM_COUNT||' aname='||aname );
END;

/


/>type nls2.bat

--==============================================================
-- Name nls2.sql
-- Description a test to determine the select performance
--
--==============================================================

set serveroutput on


DECLARE
    STATUS NUMBER(10);
    MESSAGE VARCHAR2(80);
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname   CHAR(30);
    X   NUMBER;
BEGIN

    ITEM_COUNT := 10000;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
            SELECT ENAME INTO aname FROM EMP WHERE EMPNO = 7499;
    END LOOP;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                         ' loopcount='||ITEM_COUNT||' aname='||aname );
END;

/


/>type nls3.bat


--==============================================================
-- Name nls3.sql
-- Description a test to determine the select performance
--
--==============================================================

set serveroutput on


DECLARE
    STATUS NUMBER(10);
    MESSAGE VARCHAR2(80);
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname   CHAR(30);
    X   NUMBER;
```

```
BEGIN

     ITEM_COUNT := 100000;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
     FOR I IN 1..ITEM_COUNT LOOP
               SELECT ENAME INTO aname FROM EMP WHERE EMPNO = 7521;
     END LOOP;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
     DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                              ' loopcount='||ITEM_COUNT||' aname='||aname );
END;

/

/ type nls4.sql

BEGIN

     ITEM_COUNT := 100000;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
     FOR J IN 1..ITEM_COUNT LOOP
               SELECT ENAME INTO aname FROM EMP WHERE EMPNO = 7521;
     END LOOP;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
     DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                              ' loopcount='||ITEM_COUNT||' aname='||aname );
END;

/


/ type nlu.sql

 -============================================================
 - Name nlu.sql
 - Description a test to determine the update performance

 -============================================================

set serveroutput on
alter table emp NOLOGGING;
set autocommit on

DECLARE
     ITEM_COUNT NUMBER;
     start_time  CHAR(5);
     end_time    CHAR(5);
     aname    CHAR(30);
     aSal    NUMBER;
BEGIN

     ITEM_COUNT := 100000;
     select sal into aSal from emp WHERE EMPNO = 7369;
     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
     FOR J IN 1..ITEM_COUNT LOOP
          aSal := 1000; --aSal + 1;
          UPDATE EMP SET sal = aSal WHERE EMPNO = 7369;
     END LOOP;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
     select sal into aSal from emp WHERE EMPNO = 7369;
     DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                              ' loopcount='||ITEM_COUNT||' aSal='||aSal );
END;

/


/ type nlul.sql

--============================================================
-   Name nlul.sql
-   Description a test to determine the update performance
-
--============================================================

set serveroutput on
alter table emp NOLOGGING;
set autocommit on
```

```
DECLARE
     ITEM_COUNT NUMBER;
     start_time  CHAR(5);
     end_time     CHAR(5);
     aname    CHAR(30);
     aSal    NUMBER;
BEGIN

     ITEM_COUNT := 100000;
     select sal into aSal from emp WHERE EMPNO = 7788;
     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
     FOR J IN 1..ITEM_COUNT LOOP
          aSal := 1000; --aSal + 1;
          UPDATE EMP SET sal = aSal WHERE EMPNO = 7788;
     END LOOP;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
     select sal into aSal from emp WHERE EMPNO = 7788;
     DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                              ' loopcount='||ITEM_COUNT||' aSal='||aSal );
END;

/

/ type nlu2.sql

--=============================================================
-- Name nlu2.sql
-- Description a test to determine the update performance
--
--=============================================================

set serveroutput on
alter table emp NOLOGGING;
set autocommit on

DECLARE
     ITEM_COUNT NUMBER;
     start_time  CHAR(5);
     end_time     CHAR(5);
     aname    CHAR(30);
     aSal    NUMBER;
BEGIN

     ITEM_COUNT := 100000;
     select sal into aSal from emp WHERE EMPNO = 7521;
     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
     FOR J IN 1..ITEM_COUNT LOOP
          aSal := 1000; --aSal + 1;
          UPDATE EMP SET sal = aSal WHERE EMPNO = 7521;
     END LOOP;

     SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
     select sal into aSal from emp WHERE EMPNO = 7521;
     DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                              ' loopcount='||ITEM_COUNT||' aSal='||aSal );
END;

/

/type nlu3.sql

--=============================================================
-- Name nlu3.sql
-- Description a test to determine the update performance
--
--=============================================================

set serveroutput on
alter table emp NOLOGGING;
set autocommit on

DECLARE
     ITEM_COUNT NUMBER;
     start_time  CHAR(5);
     end_time     CHAR(5);
     aname    CHAR(30);
     aSal    NUMBER;
BEGIN

     ITEM_COUNT := 100000;
     select sal into aSal from emp WHERE EMPNO = 7566;
```

```
    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
        aSal := 1000; --aSal + 1;
        UPDATE EMP SET sal = aSal WHERE EMPNO = 7566;
    END LOOP;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    select sal into aSal from emp WHERE EMPNO = 7566;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                         ' loopcount='||ITEM_COUNT||' aSal='||aSal );
END;

/


/>type nlu4.sql

--==============================================================
-- Name nlu4.sql
-- Description a test to determine the update performance
--
--==============================================================

set serveroutput on
alter table emp NOLOGGING;
set autocommit on

DECLARE
    ITEM_COUNT NUMBER;
    start_time  CHAR(5);
    end_time    CHAR(5);
    aname    CHAR(30);
    aSal    NUMBER;
BEGIN

    ITEM_COUNT := 100000;
    select sal into aSal from emp WHERE EMPNO = 7654;
    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO start_time FROM sys.dual;
    FOR J IN 1..ITEM_COUNT LOOP
        aSal := 1000; --aSal + 1;
        UPDATE EMP SET sal = aSal WHERE EMPNO = 7654;
    END LOOP;

    SELECT TO_CHAR( SYSDATE, 'SSSSS' ) INTO end_time FROM sys.dual;
    select sal into aSal from emp WHERE EMPNO = 7654;
    DBMS_OUTPUT.PUT_LINE('start_time='||start_time||' end_time='||end_time||
                         ' loopcount='||ITEM_COUNT||' aSal='||aSal );
END;

/
```

7.0 Glossary.

Glossary

7x24	24 hour operation
AKA	also known as
API	Application Program Interface
CORBA	Common Object Request Broker Architecture. Refer www.omg.com
COS	Change of State. Occurs when a field in the database is altered by an update.
DCS	Distributed Control System
ESP	Extended Simple Protocol – used on GEM PLCs
JDBC	Java Database Connectivity (JDBC) is a standard SQL database access interface, providing uniform access to a wide range of relational databases.
JVM	Java Virtual Machine
PLC	Programmable Logic Controller
RAID	Redundant Array Integrated Disk
SCADA	Supervisory Control and Data Acquisition
VARRAY	An array object in Oracle 8i
WORA	Write Once Run Anywhere