

SCALABLE IN-MEMORY DATA MANAGEMENT MODEL FOR ENTERPRISE APPLICATIONS

Anupama Piyumali Pathirage

(138223D)



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
Degree of Master of Science
www.lib.mrt.ac.lk

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

March 2015

SCALABLE IN-MEMORY DATA MANAGEMENT MODEL FOR ENTERPRISE APPLICATIONS

Anupama Piyumali Pathirage

(138223D)



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Thesis submitted in partial fulfilment of the requirements for the Master of Science in
Computer Science.

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

March 2015

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature:

Date:.....

Name: Anupama Piyumali Pathirage



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the Post Graduate Project.

Signature:

Date:.....

Name: Dr. Amal Shehan Perera

ABSTRACT

Project Title: Scalable In-Memory Data Management Model for Enterprise Applications

Authors: Pathirage A.P

Supervisor/s: Dr.Shehan Perera (Supervisor)

Dr.Malaka Walpola (Coordinator)

With the rapid advances in technology and data volume, having efficient and scalable data management system is essential for most of the enterprise applications. So In-Memory data management systems are becoming the highly used data management solution in most of the time critical enterprise solutions. Although In Memory Data Management Systems are widely used, still they are having problems such as scalability issues, concurrency problems etc. This project is an effort that aims to propose a scalable enterprise solution for in memory data management, identifying the bottlenecks in the current In-Memory Data management systems.

Although there are various benchmarks are available for Disk Resident Databases, lack of a fair metric for comparing the performance of different in-memory database systems has become a problem when selecting the appropriate data management system for enterprise applications. Currently there are various in-memory databases are available and when using them with the enterprise applications, developers have to put lot of effort as there is no standard API/Interfaces available for them.

This research project addresses these two problems by providing an unbiased performance benchmark for various in-memory databases and developing a data connector framework to access different data sources such as in-memory databases, disk resident databases, flat file data bases and in-memory data caches.

This report provides details about the problem background, existing system implementations and current research areas in this domain and how I'm going to achieve the objective.

Keywords: In-Memory Database, In-Memory Data Grid, Disk Resident Database, Data Access Layer, Database Benchmarking

ACKNOWLEDGEMENT

I would like to thank Dr. Shehan Perera, my supervisor, for his invaluable support, assistance and advices given throughout this project. His expertise and continuous guidance enabled me to complete my work successfully and his help in moderating the content was invaluable. I would also like to thank Dr. Malaka Walpola, the project co-ordinator, for his continuous support and feedback on the structure of the project which motivates me to do my best.

Further I would like to thank all my colleagues for their help on finding relevant research materials, sharing knowledge and experience and for their encouragement. My sincere appreciation goes to my husband and parents for the continuous support and motivation given to me to make this thesis a success.

Finally I would like to thank all my colleagues at DirectFN, who helped me to enhance my knowledge and for the support given to me to manage my MSc research work.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

TABLE OF CONTENT

| | |
|--|-------------|
| Declaration | I |
| Abstract | II |
| Acknowledgement | III |
| Table of Content | IV |
| Table of Figures | VI |
| Table of Tables | VII |
| List of Abbreviations | VIII |
| 1. Introduction | 2 |
| 1.1 Problem Background | 2 |
| 1.2 In-Memory Data Management Systems | 4 |
| 1.2.1 In-Memory Databases (IMDB)..... | 5 |
| 1.2.2 In-Memory Data Grid (IMDG)..... | 6 |
| 1.3 Limitations of Existing Solutions | 8 |
| 1.4 Motivation | 8 |
| 1.5 Objectives..... | 9 |
| 2. Literature Review | 10 |
| 2.1 Disk Resident Databases vs. IMDS | 10 |
| 2.2 In-Memory Database Architecture | 11 |
| 2.2.1 Impact of Memory Residency on IMDB functionality | 14 |
| 2.3 Application of Main Memory Databases | 18 |
| 2.3.1 IMDB for Embedded Systems | 18 |
| 2.3.2 IMDB for Enterprise Applications..... | 19 |
| 2.4 Performance Benchmarks for In-Memory Database..... | 21 |
| 2.4.1 Wisconsin Benchmark | 21 |
| 2.4.2 TimesTen Performance Throughput Benchmark (TPTBM)..... | 23 |
| 2.4.3 Telecom Application Transaction Processing Benchmark(TATP)..... | 24 |
| 2.4.4 Transaction Processing Performance Council -C Benchmark(TPCC)..... | 26 |
| 2.5 Cloud based In-Memory Databases | 27 |
| 3. Benchmarking Methodology | 29 |
| 3.1 Analysis of Comparison and Evaluation Scenarios | 29 |
| 3.1.1 Overview of Selected IMDB..... | 30 |
| 3.1.2 Overview of Selected DRDB | 34 |
| 3.1.3 Overview of In-Memory Data Caches | 35 |

| | | |
|-----------|---|-----------|
| 3.1.4 | Overview of Flat File Database..... | 36 |
| 3.1.5 | Feature Comparison of Selected Database..... | 37 |
| 3.2 | Analysis of Benchmark Criteria | 37 |
| 3.2.1 | Benchmark Design..... | 39 |
| 3.2.1.1 | System Configuration | 39 |
| 3.2.1.2 | Test Data | 39 |
| 3.2.1.3 | Benchmark Workload and Experimental Design..... | 40 |
| 3.2.2 | Benchmark Execution | 41 |
| 3.2.3 | Benchmark Analysis | 42 |
| 3.3 | Results | 42 |
| 3.3.1 | Results for Insert Operation | 42 |
| 3.3.2 | Results for Select Operation..... | 45 |
| 3.3.3 | Results for Update Operation..... | 48 |
| 3.3.4 | Results for Delete Operation..... | 50 |
| 4. | Framework Implementation | 53 |
| 4.1 | Problem Background | 53 |
| 4.2 | Design of the Framework | 54 |
| 4.3 | Implementation Details..... | 56 |
| 4.3.1 | Implementation of Flat File based DB..... | 56 |
| 4.3.2 | Implementation of In-Memory Cache..... | 58 |
| 4.3.3 | Implementation of the Framework for Data..... | 61 |
| 4.4 | Performance Analysis of Framework | 65 |
| 5. | Conclusion And Future Work | 67 |
| 5.1 | Conclusion..... | 67 |
| 5.2 | Future Work..... | 69 |
| 6. | References..... | 70 |

TABLE OF FIGURES

| | |
|---|----|
| Figure 1 : Moore's Law for Disk Speed | 3 |
| Figure 2 : In Memory Data Management System..... | 5 |
| Figure 3 : IMDG Architecture | 7 |
| Figure 4 : Disk Resident Databases vs. IMDS..... | 11 |
| Figure 5 : IMDB Architecture..... | 13 |
| Figure 6 : Usage of IMDB | 18 |
| Figure 7 : Enterprise Performance In Memory Cycle..... | 20 |
| Figure 8 : Times Ten Benchmark Throughput update (100% Updates)..... | 24 |
| Figure 9 : TATP benchmark on transaction processing time | 25 |
| Figure 10 : SQLite Architecture | 30 |
| Figure 11 : MemSQL Architectue | 33 |
| Figure 12 : Elements of Oracle | 34 |
| Figure 13 : Database System Benchmark Methodology..... | 38 |
| Figure 14: Example Insert Statement..... | 42 |
| Figure 15 : Insert Operation -Run Time Comparison..... | 43 |
| Figure 16 : Insert Operation - Transactions per Second Comparison..... | 43 |
| Figure 17 : Insert Operation - Concurrent Connections vs TPS | 44 |
| Figure 18 : Example Select Statement | 45 |
| Figure 19 : Select Operation - Run Time Comparison..... | 46 |
| Figure 20 : Select Operation - Transactions Per Second Comparison | 46 |
| Figure 21: Select Operation - Concurrent Connections vs TPS | 47 |
| Figure 22 : Select with Joins - TPS Comparison..... | 47 |
| Figure 23 : Example Update Statement | 48 |
| Figure 24 : Update Operation - Run Time Comparison | 48 |
| Figure 25 : Update Operation - Transactions Per Second Comparison..... | 49 |
| Figure 26 : Update Operation - Concurrent Connections vs TPS..... | 49 |
| Figure 27 : Example Delete Operation | 50 |
| Figure 28: Delete Operation -Run Time Comparison..... | 50 |
| Figure 29 : Delete Operation - Transactions Per Second Comparison | 51 |
| Figure 30 : Delete Operation - Concurrent Connections vs TPS..... | 51 |
| Figure 31 : Proposed Architecture for Database API | 55 |
| Figure 32: Database organization in Flat File DB | 56 |
| Figure 33 : Query Execution Method of Flat File DB | 57 |
| Figure 34 : Flat File DB - Table Data | 58 |
| Figure 35 : Example usage of In-Memory Cache..... | 59 |
| Figure 36: Class Diagram of In-Memory Cache..... | 60 |
| Figure 37 : Class diagram of Data Connection Framework | 62 |
| Figure 38 : ExecuteQuery Method for SQLite DB | 64 |
| Figure 39 : Example usage of Framework..... | 64 |
| Figure 40 : Insert Operation Performance of Framework - With Oracle..... | 65 |
| Figure 41 : Select Operation Performance of Framework - With SQLite | 66 |
| Figure 42 : Select Operation Performance of Framework | 66 |

TABLE OF TABLES

| | |
|---|----|
| Table 1 : CSQL Wisconsin Benchmark Results | 23 |
| Table 2 : Feature Comparison of Selected Databases..... | 37 |
| Table 3 : Benchmark System Configurations | 39 |
| Table 4 : Database Table Data | 40 |
| Table 5 : Performance Metrics..... | 41 |
| Table 6 : Benchmark Tool Implementation Details..... | 41 |



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

LIST OF ABBREVIATIONS

| Abbreviation | Description |
|--------------|---|
| ACID | Atomicity, Consistency, Isolation, Durability |
| ANSI | American National Standards Institute |
| API | Application Programming Interface |
| CDC | Change Data Capture |
| CPU | Central Processing Unit |
| CRUD | Create, Read, Update, and Delete |
| CSV | Comma Separated Values |
| DAL | Data Access Layer |
| DBA | Database Administrator |
| DML | Data Manipulation Language |
| DRDB | Disk Resident Database |
| IMDB | In-Memory Database |
| IMDG | In-Memory Data Grid |
| IMDS | In-Memory Data Management System |
| IPC | Inter Process Communication |
| JIT | Just In Time |
| JDBC | Java Database Connectivity |
| ODBC | Open Database Connectivity |
| MMDB | Main Memory Database |
| MVCC | Multi Version Concurrency Control |
| RAM | Random Access Memory |
| RDBMS | Relational Database Management System |
| RTOS | Real Time Operating System |
| SQL | Structured Query Language |
| STL | Standard Template Library |
| TPS | Transactions per Second |



University of Moratuwa, Sri Lanka.
 Electronic Theses & Dissertations
 www.lib.mrt.ac.lk

1. INTRODUCTION

This chapter is intended to provide the introduction to the project with the details of the problem background and the importance of doing this project. This chapter mainly addresses the drawbacks of traditional disk based databases, necessity of having an In Memory Data Management System, including its definition, overview and limitations. This chapter will further discuss about motivation factors which have affected for doing this research project and the objectives of this research project.

1.1 Problem Background

With the development of information technology systems and many other scientific disciplines, large data sets are very common nowadays. In domains such as weather/climate forecasting, financial and stock trading, telecommunication, airline schedulers etc., large volume of data is generated every day and these data needs to be accessed and analysed by sophisticated techniques, so that enterprises can serve their customers quickly and effectively.

Magnetic disks are the primary means of storing online information in most software systems for the past few decades. During that period magnetic disk technology has undergone a dramatic improvement in terms of their capacity and storage mechanism such as file systems, database systems etc. But the performance of disk based systems do not improve with the same pace and most of the large scale enterprises are finding it problematic to scale disk based systems to match the current business requirements.

According to most of the measures, computing power doubles every couple of years. Back in the mid-1960's, Gordon Moore, in his research paper "Cramming More Components into Integrated Circuits" introduced the idea of 'Moore's Law' which postulated that CPU power will get four times faster every three years [1]. While Moore's law is correct for processor speed and cost, later people have over-generalized this principle as it applies to disks and RAM as well. But performance of many of the other components in the overall computer infrastructure has not kept pace with the improvements in processing speeds. Performance improvements in storage systems have noticeably lagged behind. The rotational speed of the disk does not improve with the same pace and it is million times longer than raw RAM seek time. Moore's Law is true for RAM and disk costs as prices are continually falling for them, but the speed growth of disk does not follow the Moore's Law as shown in **Figure 1** [2].

So the combination of large dataset size, geographic distribution of users and resources and computationally intensive analysis result in complex performance demands which are not satisfied by any of the existing disk based data management infrastructures.

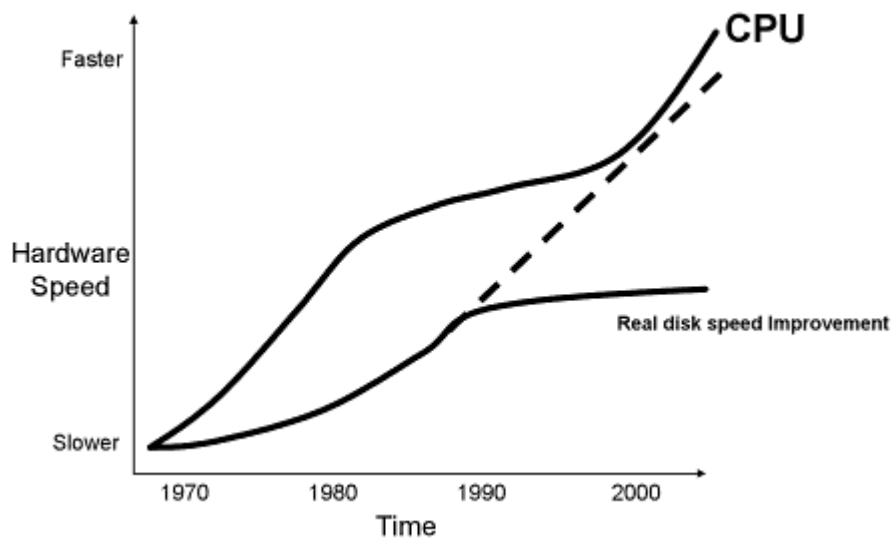


Figure 1 : Moore's Law for Disk Speed

With the rapid improvement in hardware technologies over the past few decades, multicore architectures and the availability of large amount of main memory at low costs have made a new era in data management techniques. Researchers have come up with the idea of in-memory data management and computing where primary locus of online data has shifted from disk to random access memory, with disk relegated to a backup/archival role. By moving data into memory and distributing it across multiple servers, this approach aims for easier access to data, improved scalability and better data analysis.

With the rapid growth in data volume and the requirement to access these data in real time, In-Memory Data Management Systems have become more popular. Following are some of the factors which have mainly contributed to the growth of In- Memory Data Management systems [3].

- **Large volume of Data.**

According to the recent research and surveys the amount of data created and replicated is increasing exponentially and 90% of today's data were created in the last 2-3 years. With this data explosion, enterprises are having the problem of how to manage, analyse

and protect these vast quantities of data. With the increase of data volume, time taken for access and analysis also has increased.

- **Requirement for accessing big data in real time**

As organizations have exponentially increased the volume, velocity and variability of type of data they collect, process and manage, traditional database designs are not adequate. This explosion of data presents many challenges related to scalability, timely access for critical decisions and increased cost due to increased complexity.

- **Requirement for faster analysis.**

The reason that organizations are collecting and storing more data than ever before is that their businesses depend on it and faster analysis and response are essential to survive in competitive business environment.

- **Distributed data.**

In some of the enterprise applications, data are geographically distributed. To have easier access to these data, improved scalability and better data analysis, data needs to move in to memory and may need to distribute it across multiple servers.

- **Application performance and scale**

Application development efforts must consider how database systems are in use to best determine their ultimate performance. For example, data storage systems and data access may not be able to keep up with the increasingly higher volume of transactions in an organization's mission-critical applications and this will lead to an adverse effect on the application's performance.

1.2 In-Memory Data Management Systems

To address the above mentioned problems, In-memory data management systems (IMDS) concept was introduced in the late twentieth century. An in-memory database system is a database management system that stores data entirely in the main memory. This contrasts to traditional disk based database systems, which are designed for data storage on persistent media. Because working with data in memory is much faster than writing to and reading from a file system, IMDSs can perform applications' data management functions in an order of magnitude. Since their design is typically simpler than that of on-disk databases, IMDSs can also impose significantly lower memory and CPU requirements. So In memory data management systems can achieve significant improvements in performance, processing time

and throughput rates over conventional database systems by eliminating the need for I/O to perform database applications. Since the price of random access memory is dropping and a large number of real-time applications are emerging, MMDB has become a hot research topic in database management.

As semiconductor memory becomes cheaper and chip density increases, it becomes feasible to store larger databases in memory. Since computers' main memory has different properties than the magnetic disks, design and the performance of Memory resident data management systems are different from disk resident data management systems. Memory resident database systems store their data in main physical memory and provide very high speed access [4]. How the in-memory data management systems are used in applications are shown in **Figure 2**.

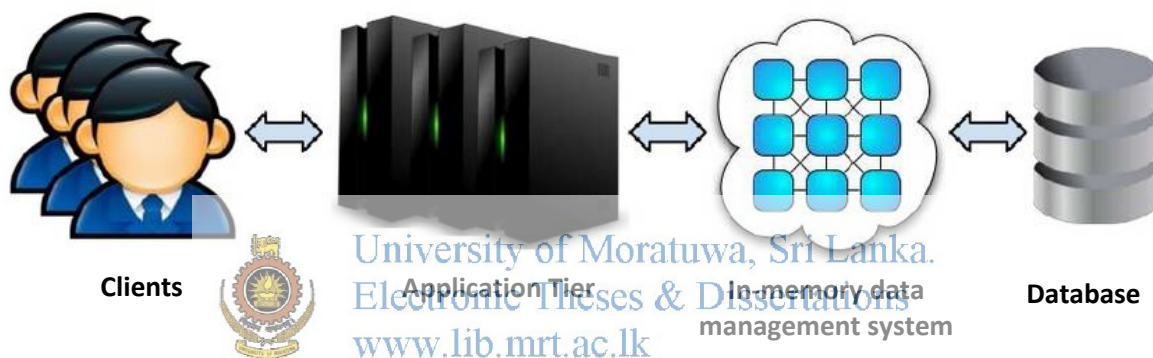


Figure 2 : In Memory Data Management System

The existing In-memory data management systems can be broadly categorized in to two areas.

- **In-Memory Databases (IMDB)** – Also known as Main Memory Database System (MMDB) or Memory Resident Database. IMDBs store their data in the main physical memory and provide very high speed access.
- **In-Memory Data Grid (IMDG)** – IMDGs are off the shelf software products and its data model is distributed across many servers in a single location or multiple locations. All data is stored in the RAM of the server.

1.2.1 In-Memory Databases (IMDB)

In-memory databases (IMDB) can be used with different types of applications and they are most commonly used in applications that demand very fast data access, storage and manipulation, and in systems that don't typically have a disk but required to manage appreciable quantities of data.

In memory database systems can be used with both embedded and non-embedded systems. IMDSs running on real-time operating systems (RTOSs) provide the responsiveness needed in applications which require different functionalities such as IP network routing, telecom switching, and industrial control. Since most embedded systems are highly resource-constrained, the small memory and CPU footprint of In-memory databases make them ideal for these systems.

Non-embedded enterprise applications which require exceptional performance are also an important growth area for in-memory database systems. For example, algorithmic trading and other applications for financial markets use IMDSs to provide instant manipulation of data in order to identify and leverage market opportunities. Some multi-user Web applications such as e-commerce and social networking sites use in-memory databases to cache portions of their back end disk based database systems. These large enterprise applications sometimes require very large in-memory data stores. So scalability is an important aspect for in-memory database systems and still it is under research although various solutions are already present.

An in-memory database system can be used either as an embedded database or as a client/server database system. The Client/server database systems are inherently multi-user and can be accessed by multiple users/processes. The embedded in-memory databases are generally single user, but it can also be shared by multiple threads/processes/users. First, the database can be created in shared memory, with the database system providing a mechanism to control concurrent access. Also, embedded databases can provide a set of interfaces that allow processes to execute on network nodes remote from the database node and to read from and write to the database. Also database replication can be used to copy the in-memory database to the nodes where processes are located, so that network traffic and latency can be eliminated [5].

1.2.2 In-Memory Data Grid (IMDG)

An In-Memory Data Grid (IMDG) is a distributed non-relational data or object store. It can be distributed to span more than one server. IMDGs usually support linear scaling to support high loads, data partitioning, redundancy, and automatic data recovery in case of failures. Most IMDGs also support multimode topologies that span WANs. The IMDG is similar to MMDB in that it stores data in the main memory, but it has a totally different architecture. The features of IMDG can be summarized as follows [6].

- Data is distributed and stored in multiple servers.
- Each server operates in the active mode.
- A data model is usually object-oriented (serialized) and non-relational.
- According to the necessity, you often need to add or reduce servers.

IMDG overcomes the limit of capacity by ensuring horizontal scalability using a distributed architecture, and resolves the issue of reliability through a replication system. As shown in figure 3, an application server has a client library provided by IMDG and it accesses IMDG by using this library. Many IMDG products provide the feature of synchronizing data to RDBMS. However establishing a separate permanent storage system such as RDBMS is not essentially required.

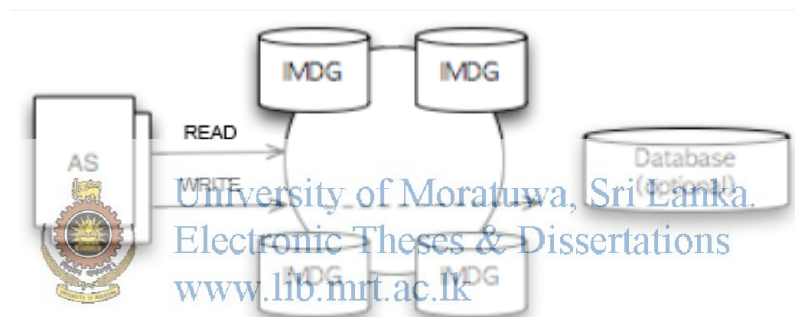


Figure 3 : IMDG Architecture

In general, IMDB enables objects to be stored through serialization. Some products provide the feature of storing objects that implement serializable interface, while some IMDGs provide an independent serialization method. The features of IMDG can be summarized as follows.

- Enhanced performance because data can be written to and read from memory much faster than it is possible with a hard disk.
- The data grid can be easily scaled and upgrades can be easily implemented.
- A key/value data structure rather than a relational structure provides flexibility for application developers.
- The technical advantages provide business benefits in the form of faster decision making, greater productivity and improved customer service.

1.3 Limitations of Existing Solutions

Although existing In-Memory Data Management Systems provide a better performance and scalability over traditional database systems, they have some limitations and still researches are being carried on over these limitations.

- Lack of standard interface/standards and lack of monitoring and visualizing data nodes has become a problem. Although a wide range of In-Memory Data Management solutions are available, there are no unified interfaces or libraries for them to easily use with enterprise applications. Different solutions are implemented with different APIs and using them with applications makes them hard to interoperate.
- Much of database system research and development is centred on innovation in system architectures, algorithms, and paradigms that deliver significant performance advantages. Lack of a fair metric for comparing the performance of different systems has become a problem when selecting the appropriate data management system for enterprise applications.
- Although In-Memory Data Management system is a widely discussed topic over the past few decades, they still have not taken the full advantages of cloud and virtualization technologies [7].
- None of the IMDGs today offer "Change Data Capture (CDC)" capabilities. That is if the backend enterprise repository is updated from the other sources. These events should propagate to the IMDG. But users have to use 3rd party products or combination of triggers and messaging to accomplish this [8].
- Lack of Global Data grid - Current enterprises demanding truly global applications where users in different parts of the world are using the same app and updating the same data set all in real-time. Because IMDGs are distributed by design, it makes them an excellent starting point for building a global data grid.

1.4 Motivation

As detailed above, there are several advantages of using In-Memory Databases in Enterprise applications instead of using traditional databases or In-Memory Data structures. But in today's world, still most of the enterprise applications are based on either traditional disk oriented

databases or In-memory data structures. Although In-Memory data management systems have been studied and developed over the past few decades, lack of performance evaluation or comparison of them is one of the problems that enterprises face today. When selecting data management methodologies such performance comparison details on In-memory databases, traditional databases and in-memory data structures are highly valuable. If such performance evaluation details are available it will be helpful for the researchers for their future studies and also for the enterprises who are willing to use them for their applications.

Although a wide variety of In-memory databases are available, there is no standard API or interface which can be easily integrated with the existing applications. So using them with enterprise applications makes the task of integration more difficult.

Scalability of the database is another major problem that enterprise applications are facing today. Cloud based solutions take the advantage of cloud resources to achieve that target. But still In-memory databases have not taken the full advantage of the cloud based technologies. So to get the maximum utilization and performance within enterprise applications, scalability of the data management system is highly important and more research work needs to be carried on in this area.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

1.5 Objectives

The main objective of this research project is to propose a scalable and interoperable model for in-memory data management system based enterprise applications. During this research, In-Memory Databases will be studied and detailed objectives of this research can be listed as follows.

- Develop a benchmark suit along with suitable workloads which can be used to evaluate the performance of In-memory databases in comparison to In-memory data structures and traditional disk based databases.
- Develop a standard API for In-memory databases which can be used with enterprise applications so that applications can manage data with a seamless interface.

2. LITERATURE REVIEW

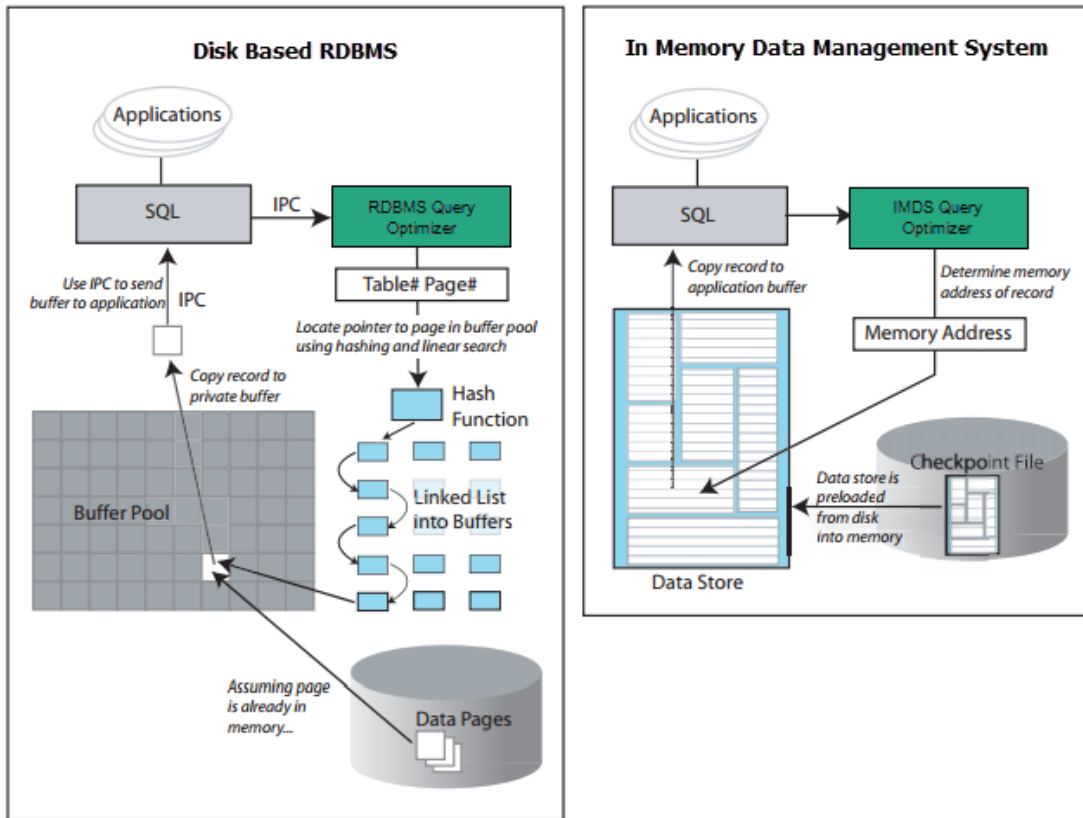
This chapter is intended to provide the details of the in-memory databases along with their architectural design details. This chapter mainly addresses the architectural and design differences of disk resident and in memory databases, in-memory database architecture and applications of in-memory databases. This chapter will further discuss about the existing benchmarks for database evaluation along with their relevance to in-memory database and the use of cloud based technologies with the in-memory databases.

2.1 Disk Resident Databases vs. IMDS

Since computers main memory has different properties than the magnetic disks, design and the performance of Memory resident data management systems are different from disk resident data management systems. These differences can be summarized as follows [9].

- The access time for main memory is orders of magnitude less than disk storage.
- The main memory is normally volatile and disk storage is non-volatile.
- Disks are block oriented storage device and main memory is not block oriented. So disks have high, fixed cost per access that does not depend on the amount of data that is retrieved during the access.
- Sequential access in disk is faster than random access. But sequential access is not important on main memories. So the layout of data on disk is much more critical than layout of data in main memory.
- Since main memory is directly accessible by the processor, it is more vulnerable to software errors than disk resident systems.

As shown in **Figure 4** [10], in a conventional RDBMS, client applications communicate with a database server process over some type of IPC connection, which adds substantial performance overhead to all SQL operations. But an application can link in-memory databases directly into its address space to eliminate the IPC overhead and streamline query processing. In disk resident databases most of the work is done under the assumption that data is primarily disk resident. So Optimization algorithms, buffer pool management, and indexed retrieval techniques are designed based on this fundamental assumption. On the other hand IMDB is designed with the knowledge that data resides in main memory and can therefore take more direct routes to data, reducing code path length and simplifying both algorithm and structure.



University of Moratuwa, Sri Lanka.
 Electronic Theses & Dissertations
 www.lib.mru.ac.lk

Figure 4 : Disk Resident Databases vs. IMDS

The complexity of IMDB is dramatically reduced since the assumption of disk-residency is not present and the advantages are as follows [10].

- The number of machine instructions drops dramatically.
- Buffer pool management is not required.
- Extra data copies are not required
- Index pages shrink, and their structure is simplified.
- The Database design gets simpler and compact.

2.2 In-Memory Database Architecture

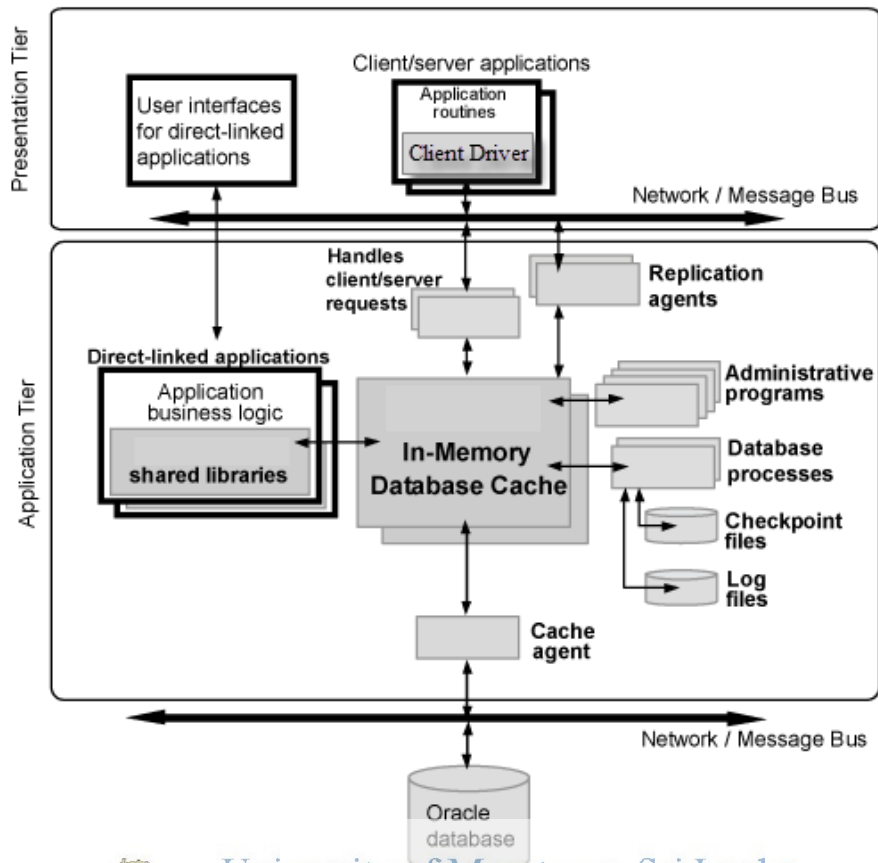
Since IMDB is not a new concept, the architecture of it has evolved during the past few decades. Memory residency of data has become a key factor on the IMDB architecture and this actually leads to much simpler design as compared to disk databases. There are six areas of difference which has made the architecture of the IMDBs are different from Disk resident databases [11].

1. **Query optimisation** - In disk DBs, the I/O cost factor dominates the optimisation. However, in IMDBs there is no such clear factor, which makes query optimisation very

tricky. This is generally solved by taking constants and falling back on rule-based optimisation.

2. **Indexing** - More memory-friendly data structures and algorithms are used for indexing. While most disk DBs use B-Tree as a primary indexing data structure/algorithm, IMDBs tend to use T-Tree as a primary indexing data structure/algorithm.
3. **Internal data representation** - Compactness of representation dominates concerns for IMDBs. With all data being in memory, IMDBs tend to use direct memory pointers heavily. This is very typical of the IMDB memory page, index data or relation representations.
4. **Durability and recovery** - Contrary to popular belief, IMDBs are durable. They use algorithms similar to disk DBs for persistence. However, the buffer management, which is the biggest performance bottleneck for disk DBs, is eliminated. During database loading, IMDBs tend to take a bit more time as they have to load the complete data into memory. Hence, recovery is a bit slower.
5. **Access methodology** - Generally, disk DBs offer client server over sockets as a primary access method. However, with no disk I/O, if IMDBs only offer sockets for access, this will become a bottleneck. Hence; most IMDBs tend to offer shared-memory access as a primary method. In a few cases, JDBC/ODBC interfaces are also supported.
6. **Concurrency control** - Due to inherent speed in processing, IMDBs can take coarser locks and also do less to persist them. However, disk DBs take finer locks and take elaborate measures to persist them.

A typical architecture for in-memory database is shown in **Figure 5** [10]. The routines that implement the IMDB functionality are embodied in a set of shared libraries that developers link with their applications and execute as a part of the application's process. This shared library approach is in contrast to a more conventional RDBMS, which is implemented as a collection of executable programs to which applications connect, typically over a client/server network. Applications can also use a client/server connection to access an IMDB Cache, though in most cases the best performance will be realized with a directly linked application.



University of Moratuwa, Sri Lanka.
 Electronic Theses & Dissertations
 www.lib.mrt.ac.lk

Figure 5 : IMDB Architecture

The IMDB Cache resides entirely in main memory at runtime. It is maintained in shared memory segments in the operating system and contains all user data, indexes, system catalogues, log buffers, lock tables and temp space. Multiple applications can share one database, and a single application can access multiple databases on the same system. Utility programs are explicitly invoked by users, scripts, or applications to perform services such as interactive SQL, bulk copy, backup and restore, database migration and system monitoring.

Checkpoint files contain an image of the database on disk. Some IMDB uses dual checkpoint files for additional safety, in case the system fails while a checkpoint operation is in progress. Changes to databases are captured in transaction logs that are written to disk periodically. If a database needs to be recovered, IMDB merges the database checkpoint on disk with the completed transactions that are still in the transaction log files. Normal disk file systems are used for checkpoints and transaction log files.

IMDB usually assigns a separate process to each database to perform operations including the following tasks.

- Loading the database into memory from a checkpoint file on disk
- Recovering the database if it needs to be recovered after loading it into memory
- Performing periodic checkpoints in the background against the active database
- Detecting and handling deadlocks
- Performing data aging
- Writing log records to files

IMDB replication allows to achieve near-continuous availability or workload distribution by sending updates between two or more servers. A master server is configured to send updates and a subscriber server is configured to receive them. A server can be both a master and a subscriber in a bidirectional replication scheme. Time-based conflict detection and resolution are used to establish precedence in case the same data is updated in multiple locations at the same time.

2.3 Impact of Memory Residency on IMDB functionality

In in-memory database systems data resides permanently in main physical memory and in disk based databases data resides in disk. In Disk based databases data may be cached in to memory for access and in IMDB the memory resident data may have a backup copy on the disk. So in both cases, a given object can have copies on both in memory and on disk. The key difference is that in IMDB the primary copy resides permanently in memory and this has important implications on how it is structured and accessed. These differences can affect the IMDB functionality as discussed in following section [9].

1. Concurrency Control

Since the access to the main memory is much faster than access to the disk, transactions complete more quickly in IMDBs. So in lock based concurrency control systems locks will be held on only for short period and the lock contention may not be as important as it is in DRDBs. Usually small locking granules are used to reduce the locking contention. But in IMDBs the contention is already low because data is memory resident and very large locking granules such as relation level granules are most appropriate for IMDBs. In extreme, the lock granule could be chosen to be the entire database [12]. This results in serial execution of transactions and it is highly desirable since the cost of concurrency

control such as setting and releasing locks, coping with deadlocks are almost completely eliminated. Also the number of CPU cache flushes are greatly reduced.

However serial transactions are not practical when long transactions are present and there should be some way to run short transactions concurrently with the long transactions. Further multiprocessor systems may require some form of concurrency control even if all transactions are short.

2. Commit Processing

Having a backup copy and keeping a log of transaction activities are essential to protect against media failures. Since memory is usually volatile, this log must reside in stable storage and before a transaction can commit, its activity records must be written to the log. Logging can impact response time, since each transaction must wait for at least one stable write before committing. Logging can also affect throughput if the log becomes a bottleneck. Although these problems also exist when data is disk resident, they are more severe in main memory systems because the logging represents the only disk operation each transaction will require.

Several methodologies can be used to solve this problem. A small amount of stable main memory can be used to hold a portion of the log and a transaction is committed by writing its log information into the stable memory, a relatively fast operation [13]. A special process or processor is then responsible for copying data from the stable memory to the log disks. Although stable memory will not alleviate a log bottleneck, it can eliminate the response time problem, since transactions need never wait for disk operations.

Group commits technique can also be used to solve the log bottleneck. Under group commit, a transaction's log record need not be sent to the log disk as soon as it commits and the records of several transactions are allowed to accumulate in memory. When enough have accumulated all are flushed to the log disk in a single disk operation and it reduces the total number of operations performed by the log disks [14].

3. Access Methods

A wide variety of index structures have been proposed and evaluated for main memory databases including various forms of hashing and of trees. Trees such as the T-Tree have been designed explicitly for memory-resident databases and they need not have the

short, bushy structure of a B-Tree, since traversing deeper trees is much faster in main memory than on a disk [15].

Since random access is fast in main memory, pointers can be followed quickly. Therefore, index structures can store pointers to the indexed data, rather than the data itself. This eliminates the problem of storing variable length fields in an index and saves space as long as the pointers are smaller than the data they point to.

4. Data Representation

Main memory databases can also take advantage of efficient pointer following for data representation. Relational tuples can be represented as a set of pointers to data values. The use of pointers is space efficient when large values appear multiple times in the database, since the actual value needs to only be stored once. Pointers also simplify the handling of variable length fields since variable length data can be represented using pointers into a heap.

5. Query Processing

Since sequential access is not significantly faster than random access in a memory resident database, query processing techniques that take advantage of faster sequential access lose that advantage. An example is sort-merge join processing, which first creates sequential access by sorting the joined relations. Although the sorted relations could be represented easily in a main memory database using pointer lists, there is really no need for this since much of the motivation for sorting is already lost.

Because data is in memory, it is possible to construct appropriate, compact data structures that can speed up queries. When relational tuples are implemented as a set of pointers to the data values some relational operations can be performed very efficiently. Query processors for memory resident data must focus on processing costs, whereas most conventional systems attempt to minimize disk access [16].

6. Recovery

To protect against the loss of volatile data, backups of memory resident databases must be maintained on disk. The recovery procedure has several components such as the procedure used during normal database operation to keep the backup up-to-date, and the procedure used to recover from a failure. Commit processing and check pointing can be used for this purpose and check pointing brings the disk resident copy of the database more up-

to-date, thereby eliminating the need for the least recent log entries. In an in-memory database system, check pointing and failure recovery are the only reasons to access the disk-resident copy of the database and check pointing should interfere as little as possible with transaction processing [17].

7. Performance

Other than the commit processing, the performance of an in-memory database manager depends primarily on processing time, and not on the disks. Even recovery management, which involves the disks, affects performance primarily through the processor, since disk operations are normally performed outside the critical paths of the transactions [18].

But in IMDB, backups will be more frequent and will involve writes to devices an order of magnitude slower than memory. Thus the performance of backup or check pointing algorithms is much more critical and need to handle more carefully.

8. Application Programming Interface and Protection

In conventional disk based databases, applications exchange data with the database management system via private buffers. In an IMDB, access to objects can be more efficient since applications may be given the actual memory position of the object, which is used instead of a more general object id. After the first read, the system returns the memory address of the tuple, and it is used for subsequent accesses. However, there are some potential problems such as once transactions can access the database directly, they can read or modify unauthorized parts and the system has no way of knowing what has been modified, so it cannot log the changes [19].

9. Data Clustering and Migration

In a DRDB, data objects such as tuples, fields that are accessed together are frequently stored together, or clustered. But in an IMDB there is no need to cluster objects. This introduces a problem that does not arise in conventional systems. That is when an object is to migrate to disk, how and where it should be stored. There are a variety of solutions for this, ranging from ones where the users specify how objects are to be clustered if they migrate, to ones where the system determines the access patterns and clusters automatically [20].

2.4 Application of Main Memory Databases

In-memory databases are most commonly used in applications that demand very fast data access, storage and manipulation, and in systems that don't typically have a disk but must manage appreciable quantities of data. Applications that use IMDBs can be categorized in to two main categories as embedded systems and enterprise applications. According to a survey done by Elliot King in 2011 the usage of IMDBs in applications is shown in **Figure 6** [21].

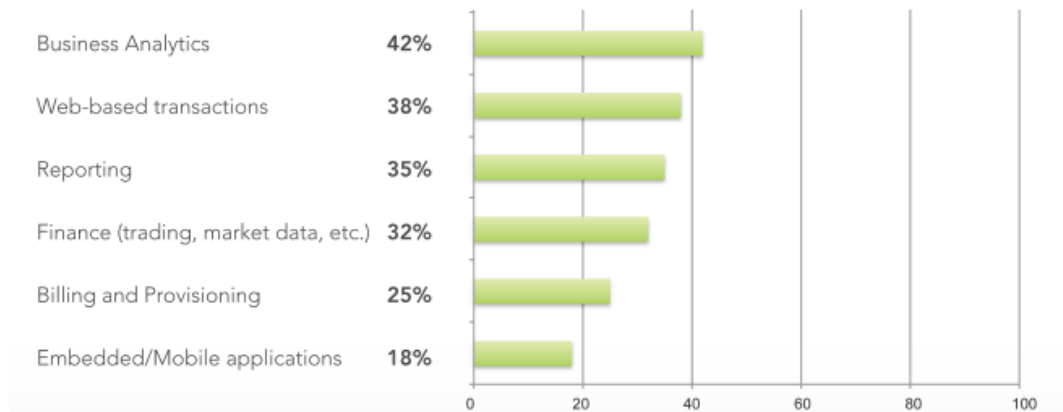


Figure 6 : Usage of IMDB

2.4.1 IMDB for Embedded Systems



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

An important use for in-memory database systems is in real-time embedded systems. IMDSs running on real-time operating systems (RTOSs) provide the responsiveness needed in applications including IP network routing, telecom switching, and industrial control. IMDSs manage music databases in MP3 players and handle programming data in set-top boxes. In-memory databases' typically small memory and CPU footprint make them ideal because most embedded systems are highly resource-constrained. The main issues for IMDBs in embedded systems can be summarized as follows [22].

- **Minimization of the memory footprint:** The memory demand for an embedded system are most often, mainly for economic reasons, kept as low as possible. A typical footprint for an embedded database is within the range of some kilobytes to a couple of megabytes.
- **Reduction of resource allocations:** In an embedded system, the database management system and the application are most often run on the same processor, putting a great demand on the database process to allocate minimum CPU bandwidth to leave as much capacity as possible to the application.

- **Support for multiple operating systems:** In an enterprise database system, the DBMS is typically run on a dedicated server using a normal operating system. The clients, that could be desktop computers, other servers, or even embedded systems, connect to the server using a network connection. Because a database most often run on the same piece of hardware as the application in an embedded system, and that embedded systems often use specialized operating systems, the database system must support these operating systems.
- **High availability:** In contrast to a traditional database system, most embedded database systems do not have a system administrator present during run-time. Therefore, an embedded database must be able to run on its own [23].

2.4.2 IMDB for Enterprise Applications

The enterprise applications are going through a transformation in regulatory requirements, technology, and operational resource needs. The era of highly customized, proprietary hardware and software is no longer desirable because it breeds high infrastructure costs and extends the time from concept to inception and implementation. For many years, financial platforms were often based on home-grown software, using closed proprietary frameworks and data management solutions. While the resulting home-grown infrastructures achieved some measure of success, they often did not scale well and lacked the flexibility to cost-effectively accommodate new services and technological innovation.

Non-embedded applications requiring exceptional performance are an important growth area for in-memory database systems. For example, algorithmic trading and other applications for financial markets use IMDSs to provide instant manipulation of data, in order to identify and leverage market opportunities. Some multi-user Web applications – such as e-commerce and social networking sites – use in-memory databases to cache portions of their back-end on-disk database systems. These enterprise-scale applications sometimes require very large in-memory data stores, and this need is met by 64-bit IMDS editions [24].

Whether running on enterprise servers, embedded in appliances, in the cloud, or processing constantly-changing complex data, financial applications need a platform characterized by low latency, high availability, and a scalable infrastructure that allows for rapid growth. IMDBs provide the necessary agility for companies developing and deploying financial applications that meet or exceed today's stringent requirements. Also they provides developers a superior

alternative to building or deploying other data management solutions and helps developers deliver greater innovation with shorter time to market.

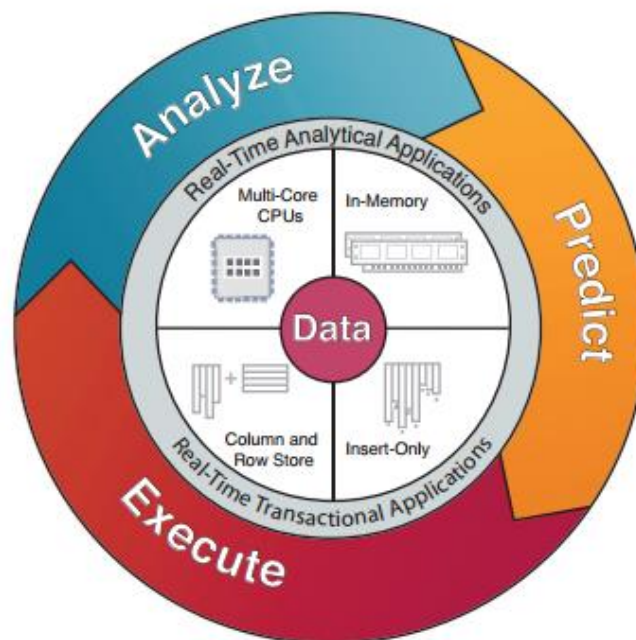


Figure 7. Enterprise Performance In-Memory Cycle
 University of Moratuwa, Sri Lanka.
 Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Currently, most of the data within a company is still distributed throughout a wide range of applications and stored in several disjoint silos. Creating a unified view on this data is a time-consuming procedure. Additionally, analytical reports typically do not run directly on operational data, but on aggregated data from a data warehouse. Operational data is transferred into this data warehouse in batch jobs, which makes flexible, ad-hoc reporting on up-to-date data almost impossible. As a consequence, enterprises have to make decisions based on insufficient information, which is not what the term real-time suggests. Since the hardware architectures have evolved dramatically in the last decade this is changing now. Multi-core processors and the availability of large amounts of main memory at low cost are creating new breakthroughs in the software industry. It has become possible to store data sets of whole companies entirely in main memory, which offers performance that is orders of magnitudes faster than traditional disk-based systems. Hard disks will become obsolete. The only remaining mechanical device in a world of silicon will soon only be necessary for backing up data. With in-memory computing and insert-only databases using row- and column-oriented storage, transactional and analytical processing can be unified. High performance in-memory computing will change how enterprises work and finally offer the promise of real-time computing. As shown in **Figure 7**, the combination of the technologies finally enables an

iterative link between the instant analysis of data, the prediction of business trends, and the execution of business decisions without delays [25].

2.5 Performance Benchmarks for In-Memory Database

With the recent, but widespread, acceptance of the Main-memory databases, there has been a lot of different companies and people interested in the potential and advantages of main-memory databases. There is currently dozens of different databases that use main-memory techniques. The performance of databases does not rely solely on the actual speed of the database. A big part of how effective a database is comes from how you use it. Different databases are good at different things and different types of databases focus on optimizing different utilities. In several studies, the performance of either traditional disk resident database and a selected in-memory database or several in-memory databases are compared.

Another way to understand performance trade-offs between different in-memory databases is to review independent benchmarks that are produced which compare each database under different workloads. While such tests can never take the place of proof of concepts done using the exact use cases and infrastructure that a new application is targeting, they can be useful to understand the general strengths and weaknesses of a database under various workloads. In the following section various benchmarks which can be used to evaluate the performance of in-memory databases are discussed.

2.5.1 Wisconsin Benchmark

The Wisconsin Benchmark was introduced in 1983 and it was the first real benchmark for relational databases. At that time no standard database benchmark existed and there were only a few application-specific benchmarks. The benchmark was designed with two objectives in mind. First, the queries in the benchmark should test the performance of the major components of a relational database system. Second, the semantics and statistics of the underlying relations should be well understood so that it is easy to add new queries and to their behaviour.

The database is designed so that one can quickly understand the structure of the relations and the distribution of each attribute value. Consequently, the results of the benchmark queries are easy to understand and additional queries are simple to design. The attributes of each relation are designed to simplify the task of controlling selectivity factors in selections and joins, varying the number of duplicate tuples created by a projection, and controlling the number of partitions in aggregate function queries. It is also straightforward to build an index (primary or

secondary) on some of the attributes, and to reorganize a relation so that it is clustered with respect to an index.

The suite of benchmark queries was designed to measure the performance of all the basic relational operations including:

- Selection with different selectivity factors.
- Projections with different percentages of duplicate attributes.
- Single and multiple joins.
- Simple aggregates and aggregate functions.
- Append, delete, modify.

In addition, for most queries, the benchmark contains two variations: one that can take advantage of a primary, clustered index, and a second that can only use a secondary, non-clustered index. Elapsed time is used as the performance metric [26].

Limitations:

- It is a benchmark designed to evaluate Disk-based databases and no IMDB concept is taken in to account.
- It is a single user benchmark and no tests for concurrency control and recovery.
- It tests features of the query optimizer only.

This benchmark is used to evaluate some main memory databases in past such example is as follows. For the above said operations, time taken is measured in microsecond for leading traditional database system and for CSQL Main Memory Database System. CSQL is an open source main memory high-performance relational database management system developed at sourceforge.net. It is designed to provide high performance for SQL queries and DML statements. The benchmarking application and the database server runs in the same machine/host and table fully cached in RAM during the test. The elapsed time is measured in micro seconds and the results are shown in **Table 1**. From these results, CSQL is claimed that it is approximately 30 times faster than leading database with standard JDBC interface for real time database operations [27].

Table 1 : CSQL Wisconsin Benchmark Results

| Statement Type | Leading DRDB | | | CSQL | | | Times Faster | | |
|----------------|--------------|------------|------------|----------|------------|------------|--------------|------------|------------|
| | No Index | Hash Index | Tree Index | No Index | Hash Index | Tree Index | No Index | Hash Index | Tree Index |
| Select Int | 6097 | 331 | 325 | 247 | 11 | 11 | 24.68 | 30.09 | 29.55 |
| Select Str | 6495 | 979 | 356 | 286 | 16 | 15 | 22.71 | 61.19 | 23.73 |
| Select -100 | 6861 | NA | 826 | 508 | NA | 120 | 13.51 | NA | 6.88 |
| Insert | 218 | 265 | 213 | 20 | 13 | 11 | 10.9 | 20.38 | 19.36 |
| Update | 5572 | 217 | 188 | 473 | 14 | 12 | 11.78 | 15.5 | 15.67 |
| Delete | 5741 | 200 | 168 | 573 | 15 | 13 | 10.02 | 13.33 | 12.92 |
| Join 10K * 1K | 6459 | 320 | 292 | 35 | 11 | 11 | 184.54 | 29.09 | 26.55 |
| Join 10K * 10K | 14916 | 411 | 320 | 36 | 13 | 14 | 414.33 | 31.62 | 22.86 |

2.5.2 TimesTen Performance Throughput Benchmark (TPTBM)

Oracle TimesTen In-Memory database is a high performance event-processing software component that enables applications to capture, store, use, and distribute information in real-time, while preserving transactional integrity and continuous availability. TimesTen Performance Throughput Benchmark (TPTBM) is shipped with TimesTen and measures the total throughput of the system. The workload can test read-only, update-only, delete and insert operations or mix of them as required. It is a multi-user throughput benchmark. By default, the transaction mix consists of 80% SELECT (read) transactions and 20% UPDATE (write) transactions. The ratio of SELECTs, UPDATEs and INSERTs can be specified and each transaction consists of one or more SQL operations [10].

Limitations:

- TPTBM is a proprietary benchmark and shifts with oracle times ten only.
- TPTBM is vendor specific.

Figure 8 shows the performance impact of placing the TimesTen logs on file cache, compared to traditional approaches that place the logs on cached disk-array storage. These tests were conducted using the TimesTen TPTBM benchmark running on a 2-processor Sun E450 server [28].

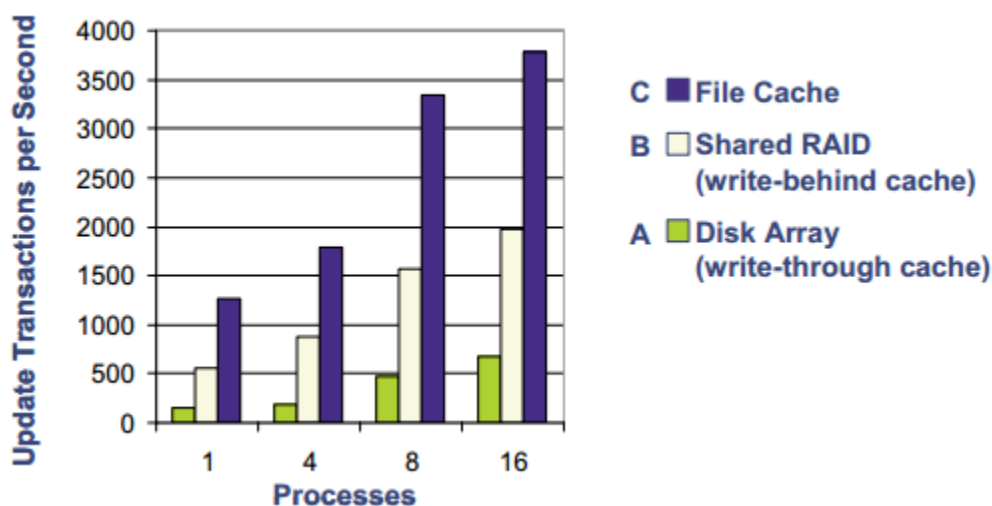


Figure 8 : Times Ten Benchmark Throughput update (100% Updates)

2.5.3 Telecom Application Transaction Processing Benchmark(TATP)

The Telecommunication Application Transaction Processing (TATP) Benchmark is an open source workload designed specifically for high-throughput applications, well suited for in-memory database performance analysis and system comparison.

The TATP benchmark simulates a typical Home Location Register (HLR) database used by a mobile carrier. The HLR is an application mobile network operators use to store all relevant information about valid subscribers, including the mobile phone number, the services to which they have subscribed, access privileges, and the current location of the subscriber's handset. Every call to and from a mobile phone involves look ups against the HLRs of both parties, making it a perfect example of a demanding, high-throughput environment where the workloads are pertinent to all applications requiring extreme speed: telecommunications, financial services, gaming, event processing and alerting, reservation systems, and so on. The benchmark generates a flooding load on a database server. This means that the load is generated up to the maximum throughput point that the server can sustain. The load is composed of pre-defined transactions run against a specified target database.

The benchmark uses four tables and a set of seven transactions that may be combined in different mixes. The most typical mix is a combination of 80% or read transactions and 20% of modification transactions [29].

The TATP software collects two types of results from the benchmark, namely Mean Qualified Throughput (MQTh) and transaction response time distributions. MQTh is the number of successful transactions per time unit. In TATP, we use one second as a time unit, resulting in MQTh tps. The response time is measured for each individual transaction and reported by transaction type. This provides seven distributions measured with a millisecond resolution. The maximum response time recorded is set to be 10,000 millisecond (10 seconds). Longer response times are discarded.

The TATP benchmark transaction response time comparison between an in-memory database and a hybrid database is shown in **Figure 9** [30].

Limitations:

- It is an Application specific benchmark - simulates a typical Home Location Register (HLR) database used by a mobile carrier.

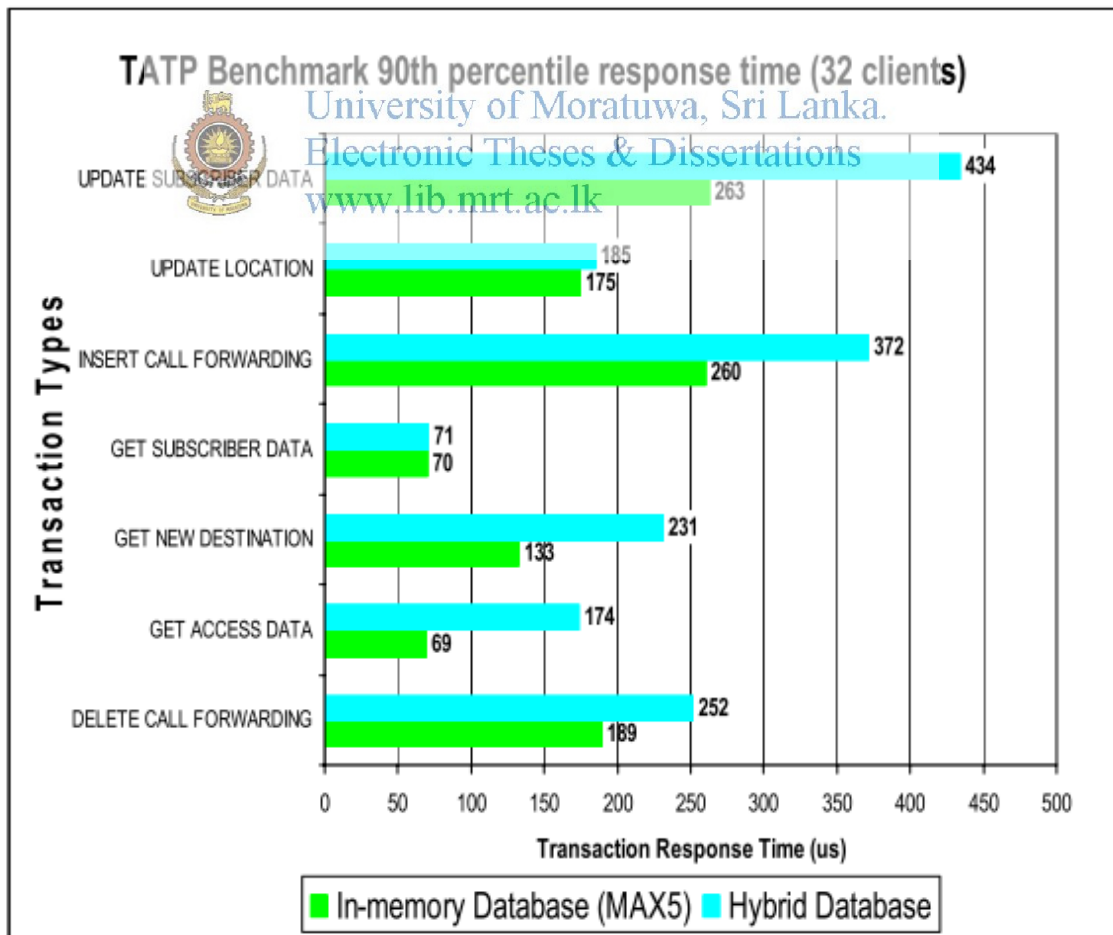


Figure 9 : TATP benchmark on transaction processing time

2.5.4 Transaction Processing Performance Council -C Benchmark(TPCC)

The Transaction Processing performance Council introduced the TPC Benchmark C in August 1992. At the time the TPC had two other OLTP benchmarks, TPC-A and TPC-B. The TPC continued to support and publish results on TPC-A, its first OLTP benchmark until December 1995. TPC-A simulates all the major functions of a simple OLTP system and was, until its retirement by the TPC, accepted by the industry as the leading tool for comparing systems. Since then, TPC-C has replaced it in that role and gained even greater recognition.

TPC-C was designed to carry over many of the characteristics of TPC-A. Therefore, TPC-C includes all the components of a basic OLTP benchmark. For the benchmark to be applicable to systems of varying computing powers, TPC-C implementations must scale both the number of terminals and the size of the database proportionally to the computing power of the measured system. To test whether the measured system is fully production-ready, including efficient recovery capabilities, the database must provide what are defined as the ACID properties: atomicity, consistency, isolation, and durability.

TPC-C involves a mix of five concurrent transactions of different types and complexity that are executed either on-line or queued for deferred execution. The major characteristics that TPC-C added beyond TPC-A can be summarized as follows [31].

- Multiple types of transactions of varying complexity
- On-line and deferred execution of transactions
- More complex database structure, resulting in
 - ✓ Greater diversity in the data that are manipulated
 - ✓ Higher levels of contention for data access and update
- Input data that include basic real-life characteristics, such as:
 - ✓ Non-uniform patterns of data access to simulate data hot spots
 - ✓ Data access by primary as well as secondary keys
- More realistic requirements, such as:
 - ✓ Terminal input/output with full-screen formatting
 - ✓ Required support for basic features of users' interface
 - ✓ Required application transparency for all database partitioning
- Transaction rollbacks

TPC-C performance is measured in new-order transactions per minute. The primary metrics are the transaction rate (tpmC), the associated price per transaction (\$/tpmC), and the availability date of the priced configuration.

Limitations:

- It is a benchmark designed to evaluate Disk based databases and no IMDB concept is taken in to account.

2.6 Cloud based In-Memory Databases

Traditionally the server and their applications of a business are located in private or exclusive computer centres. The availability of broadband internet connections makes it possible to dispense of internal computer centres and to utilize dynamically the computer capacity of a Computing Cloud of an external server.

Cloud Computing is of interest to business as no capital expenditure occurs and through the use of scale effect running costs can be minimized. The cost to customers can also be reduced by taking advantage of the elasticity of the cloud concept. Enterprises pay only for the required computing performance. Is less or more computing output required, the supplier can make this automatically available through an interface. While in classical computer centres hardware has to be dimensioned for a maximum load, using cloud computing enables to employ only the actually required hardware resources which are expanded or minimized depending on the required capacity. Cloud computing systems are not customer-based (on-premise) but are used and scaled depending on demand (on-demand). The operating risk of the computer centre is outsourced from the enterprise to the manager of the cloud. This goes together with the promise that employees from everywhere at any time have access to their data within the cloud, although this can lead to security problems.

Today most enterprises have consider using one of the many available cloud platforms to improve on speed of delivery, cost saving, and reliability. One of the most attractive features of today's cloud offerings is that they enable IT to extend the capacity of their solutions beyond the scope of on-premise servers. This can be in terms of high availability, disaster recovery, or scaling to meet planned and unplanned spikes in usage.

A major challenge in moving applications from on-premise data centres to public clouds is the reluctance to store sensitive data on the cloud, for various reasons such as perceived lack of

control over the storage, security concerns or non-compliance issues when data is stored beyond the enterprise's boundaries, or the need to store the data on-premise for other internal applications to access. There might also be cases where the data resides within systems or servers that simply have no equivalent component available on the cloud, such as a proprietary data store like a file system, or mainframe database [24].

Still the in-memory databases with cloud based solutions are under the research and only few database vendors stepped in to that. Oracle Exalogic Elastic Cloud (Exalogic) is an integrated hardware and software system designed to provide a complete platform for a wide range of application types and widely varied workloads using oracle in-memory database called TimesTen. Oracle Exalogic is intended for large-scale, performance-sensitive, mission-critical application deployments. It combines Oracle Fusion Middleware software and industry-standard Sun hardware to enable a high degree of isolation between concurrently deployed applications, which have varied security, reliability, and performance requirements. Real-time OLTP applications can benefit greatly from the combined compute power of Exalogic and TimesTen [28].



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

3. BENCHMARKING METHODOLOGY

This chapter is intended to compare various different data management systems against the in memory databases, with the intention of integrating in to the proposed data management framework which is discussed in **Section 4.2**.

This chapter provides a comprehensive analysis among different IMDBs, DRDBs, in memory data structures and flat file based DBs, with the intention of identifying the best possible candidate to be integrated in the final solution. The details of the evaluation scenarios considered in this research project is given in **Section 3.1**. In **Section 3.2**, a detailed analysis of benchmark procedure is given along with the performance metrics and workload parameters. In **Section 3.3**, the results of the benchmark procedure is discussed under different selected operation categories.

3.1 Analysis of Comparison and Evaluation Scenarios

During the initial phase, the evaluation scenarios which are considered under this research project was clearly identified. Although various performance tests and benchmark results are available in literature, they are considering either only few solutions or they are fully vendor specific which are biased towards a particular vendor. So main objective of the research is to provide unbiased evaluation results for in-memory databases, so that any enterprise level application can choose the suitable solution based on that. For this evaluation, several open source and proprietary in-memory and disk based databases were selected and following evaluation scenarios are considered.

- ✓ IMDB vs DRDB – To evaluate the performance between selected in-memory databases and disk resident databases.
- ✓ IMDB vs In-memory data structures – To evaluate the performance between the selected in-memory databases and selected in-memory data cache which is based on structures such as maps, vectors, queues etc.
- ✓ IMDB vs Flat File database systems – To evaluate the performance between the selected in-memory databases and text file databases.
- ✓ Different IMDBs – To have an unbiased comparison for the existing popular in-memory DBs, different IMDBs are evaluated.

The details of selected data management systems for this evaluation is given in the next section.

3.1.1 Overview of Selected IMDB

SQLite

SQLite is an in-process library which provides an embedded SQL database engine and designed in 2000. It is a self-contained, serverless, zero-configuration, transactional SQL database engine and it distributed as a free and open source database engine. The SQLite database is normally stored in a single ordinary disk file and it can be configured to work as an in-memory database where required. Unlike client–server database management systems, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the application program. The library can also be called dynamically. A block diagram of SQLite Architecture components and how they interrelate is shown in the **Figure 10**.

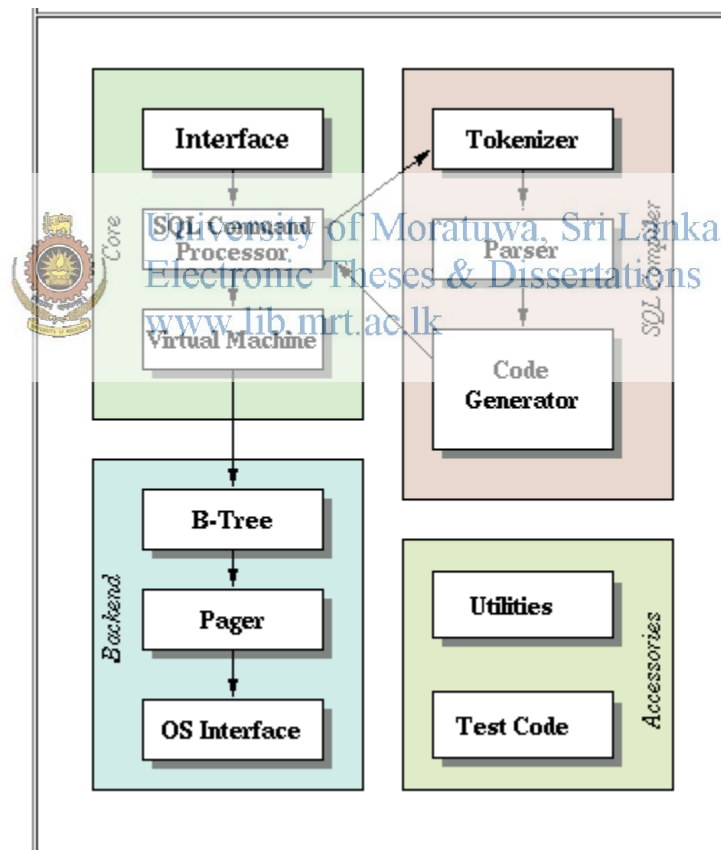


Figure 10 : SQLite Architecture

The application program uses SQLite's functionality through simple function calls, which reduce latency in database access because function calls within a single process are more efficient than inter-process communication. Some important features of SQLite is as follows [33].

- Transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.
- Supports most of the query language features found in the SQL92 (SQL2) standard.
- Written in ANSI-C and provides simple and easy-to-use API.
- Available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT).
- Interface API support available for C++, Java, PHP, Perl and Python

A SQLite database is normally stored in a single ordinary disk file. However, in certain circumstances, the database might be stored in memory. The most common way to force an SQLite database to exist purely in memory is to open the database using the special filename ":memory:". When this is done, no disk file is opened. Instead, a new database is created purely in memory. The database ceases to exist as soon as the database connection is closed. Every ":memory:" database is distinct from every other. So, opening two database connections each with the filename ":memory:" will create two independent in-memory databases.

H2 Database



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

H2 is a relational database management system written in Java. It can be embedded in Java applications or run in the client-server mode. H2 implements an embedded and standalone ANSI-SQL89 compliant SQL engine on top of a B-tree based disk store. The following connection modes are supported by H2 database:

- Embedded mode (local connections using JDBC)
- Server mode (remote connections using JDBC or ODBC over TCP/IP)
- Mixed mode (local and remote connections at the same time)

It is possible to create both in-memory tables, as well as disk-based tables. Tables can be persistent or temporary. In H2 database the index types are implemented as a hash table and a tree is for in-memory tables, and b-tree for disk-based tables. All data manipulation operations are transactional. Table level locking and multi-version concurrency control are implemented. The 2-phase commit protocol is supported in this database, but no standard API for distributed transactions is implemented. Following connection scenarios are available for in-memory mode [34].

- Only one connection to an in-memory database: This means the database to be opened is private. Opening two connections within the same virtual machine means opening two different (private) databases.
- Multiple connections to the same in-memory database: The database URL must include a name. Accessing the same database using this URL only works within the same virtual machine and class loader environment.
- Access an in-memory database from another process or from another computer: Need to start a TCP server in the same process as the in-memory database was created. The other processes then need to access the database over TCP/IP or TLS, using a database URL.

According to the literature, following reasons are given as the advantages of H2 over SQLite Database [35].

- Full Unicode support including UPPER() and LOWER().
- Streaming API for BLOB and CLOB data.
- Full text search.
- Multiple connections.
- User defined functions and triggers.
- Database file encryption.
- Reading and writing CSV files (this feature can be used outside the database as well).
- Referential integrity and check constraints.
- Better data type and SQL support.
- In-memory databases, read-only databases, linked tables.
- Better compatibility with other databases which simplifies porting applications.
- Possibly better performance (so far for read operations).
- Server mode (accessing a database on a different machine over TCP/IP).

MemSQL

MemSQL is a distributed, in-memory database that is part of the NewSQL movement. It is an ACID-compliant RDBMS that most notably converts SQL into C++ through code generation. It is being developed by MemSQL Inc., which was founded in 2011.

It uses multi-version concurrency control (MVCC) and lock-free data structures to enable high throughput for large concurrent workloads without sacrificing consistency. As a result, reads

do not block writes, and vice versa, providing the fast access necessary to achieve real-time analytics on a Big Data scale. MemSQL combines lock-free data structures and a just-in-time (JIT) compiler for processing highly volatile workloads. More specifically, MemSQL implements lock-free hash tables and lock-free skip lists in memory for fast random access to data. Queries sent to the MemSQL server are converted into C++ and compiled through GCC. MemSQL works best on workloads with highly concurrent read and write queries. Each query is individually executed on exactly one core. Read queries are never blocked by other reads or writes because of multi-version concurrency control.

MemSQL architecture is shown in **Figure 11**. It has a two-tiered, clustered architecture that consists of two types of nodes:

- **Aggregator nodes** serve as mediators between the client and the cluster. They query the relevant leaf nodes and aggregate results before sending them back to the client. Aggregators store only metadata. An aggregator is responsible for breaking up the query across the relevant leaf nodes and aggregating results back to the client.
- **Leaf nodes**, store and process data. MemSQL has a shared-nothing architecture, which means that no two nodes share memory, disk, or CPU. A leaf node is a MemSQL database. MemSQL uses hash partitioning to distribute data uniformly across the number of leaf nodes.

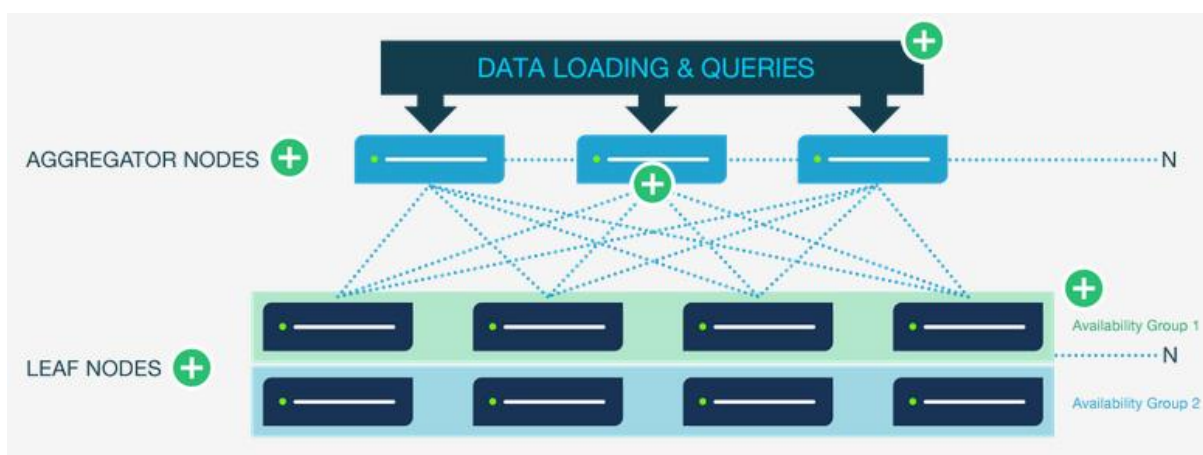


Figure 11 : MemSQL Architectue

MemSQL has two types of tables: reference tables and distributed tables. Each node in the cluster has an identical copy of all reference tables. Distributed tables are spread across all nodes in the cluster, so each node has a piece of each distributed table. This enables joins to be more efficient, with compute overhead offloaded to the leaf nodes [36].

3.1.2 Overview of Selected DRDB

Oracle

Oracle Database is a disk resident object-relational database management system produced and marketed by Oracle Corporation. It is a fourth generation relational database management system and Oracle server provides efficient and effective solutions for the major database features. Oracle revolutionized the field of enterprise database management systems with the release of Oracle Database 10g and currently oracle can be considered as the market leader in database solutions. Oracle Database is the first database designed for enterprise grid computing, the most flexible and cost effective way to manage information and applications.

The Oracle RDBMS stores data logically in the form of tablespaces and physically in the form of data files. Tablespaces can contain various types of memory segments, such as Data Segments, Index Segments, etc. An Oracle database is a collection of data treated as a unit. The database has logical structures and physical structures. Because the physical and logical structures are separate, the physical storage of data can be managed without affecting the access to logical storage structures.

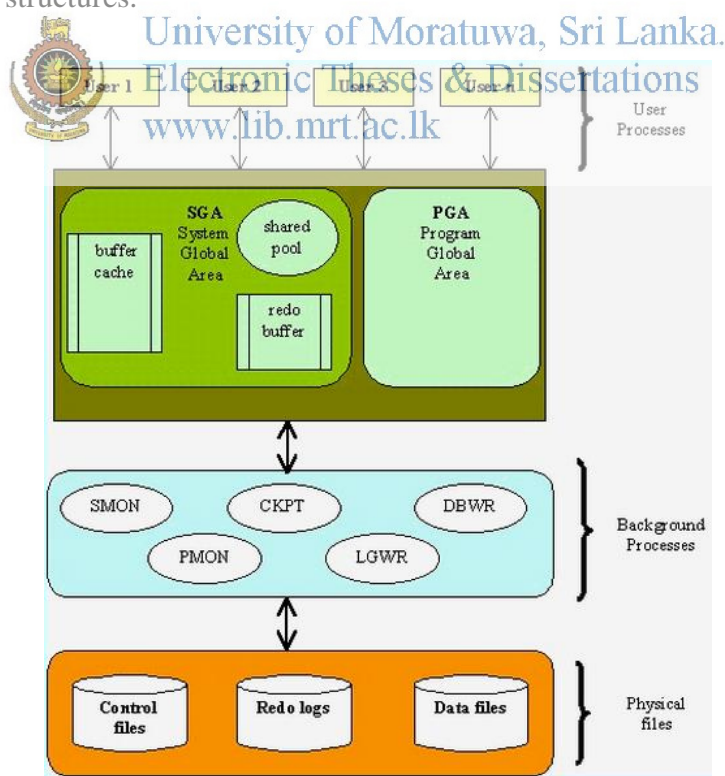


Figure 12 : Elements of Oracle

Optimising performance is ensuring that Oracle is reading from disk as little as possible, and minimize the contention between users as far as possible. A normally active database will

consist of an Instance running on a server, which manages requests from user processes to access the data files which may, or may not, be on permanent media within the server. The elements of the active database is shown in **Figure 12**. The background processes are all internally managed by Oracle, although a DBA can alter some of the processes. It demonstrates the various disk, memory, and process components of the Oracle instance. All of these features working together allow Oracle to handle data management for applications ranging from small "data marts" with fewer than five users to enterprise-wide client/server applications designed for online transaction processing for 50,000+ users in a global environment.

The latest version of the database is Oracle 12c and Oracle Database 11g Enterprise Edition Release has been used for this research [37].

3.1.3 Overview of In-Memory Data Caches

The need for caching behaviour sometimes arises during system development because a complex calculation is needed to obtain the result, or because it must be obtained via a time consuming I/O operation. If the total number of such results dealt with over the lifetime of the system does not consume excessive memory, it may suffice to store them in simple key-value containers such as `std::map`, `std::sets`.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

In memory data caches can do part of what a database do with a high performance on big sets of data as long as complex queries are not required. But any database system goes far beyond giving a set of interfaces to manage collections, lists, etc. This typically includes support for ACID (atomic, consistent, isolated and durable) transactions, multi-user access, a high level data definition language, one or more programming interfaces (including industry-standard SQL), triggers/event notifications, and more.

A key-value container based caching system is a useful tool in any programmer's performance optimisation tool-kit. Although there are lot of such solutions for Java language, there is no ready-to-use implementations provided in the standard library or the widely used boost libraries for C++ Language. So C++ developers are likely resort to inefficient or incorrect approximations to the logic. For this research an in-memory caching system is used, which is designed to increase application performance by holding frequently-requested data in memory, while reducing the need for database queries to get that data. The implementation is based on the C++ standard library's map data types. The implementation details of this caching system is given in **Section 4.3.2**

3.1.4 Overview of Flat File Database

A flat file database is a database which, when not being used, is stored on its host computer system as an ordinary, non-indexed "flat" file. To access the structure of the data and manipulate it, the file must be read in its entirety into the computer's memory. Upon completion of the database operations, the file is again written out in its entirety to the host's file system.

A flat file database is the simplest form of database systems. There is no possibility to access the multiple tables like a RDBMS. Because it uses the simple structure, a text file considered as a table. Every line of the text file is rows of table and the columns are separated by delimiters like comma, tab, and some special characters. The database does not have specific data type. A flat file can be a plain text file or a binary file. There are usually no structural relationships between the records. Some advantages and disadvantages of flat file databases are as follows.

Advantages

- Easy to understand.
- Easy to implement.
- Less hardware and software requirements.
- Less Skills set are required to hand flat database systems.
- Best for small databases.

Desadvantages

- Less security easy to extract information.
- Data Inconsistency
- Redundancy
- Sharing of information is cumbersome task
- Slow for huge database
- Searching process is time consuming

During this research, to compare the performance of in-memory databases, flat file database is also used as it is the simplest form of database systems. The flat file database system developed for this research has been implemented using C++ language and the details of the design and implementation is given in **Section 4.3.1**.

3.1.5 Feature Comparison of Selected Database

The features and support for various programming models of these selected databases is summarized in **Table 2**.

Table 2 : Feature Comparison of Selected Databases

| | SQLite | H2 | MemSQL | Oracle |
|----------------------|--|---|---------------------------------------|--|
| Licence | Public domain | Eclipse Public License | Proprietary | Proprietary |
| Database model | Relational | Relational | Relational Distributed data structure | Relational |
| Data Storage | File System Volatile memory | File System Volatile memory | File System Volatile memory | File System ASM |
| Embeddable | Yes | Yes | Yes | No |
| OS Support | Windows, OS X, Linux, BSD, Unix, Amigaz/OS, Symbion, iOS, Android | Windows, OS X, Linux, BSD, Unix, z/OS, Android | 64-bit Linux-based OS | Windows, OS X, Linux, Unix, z/OS |
| Programming Language | Java , Delphi, Python | Java | C++ | C++ |
| Query Language | SQL | SQL | SQL | SQL, HTTP,Xquery, Xpath, Java API, REST |

3.2 Analysis of Benchmark Criteria

Benchmarking is one of several alternate methods of performance evaluation, which is a key aspect in the selection of database systems. Database benchmarking is a process of performing well defined tests on that particular database management system for the purpose of evaluating its performance. Benchmarking requires that the systems be implemented so that experiments

can be run under similar system environments. Although benchmarks are costly and time consuming, it provides the most valid performance results. In data management system benchmarking, a system configuration, a database, and a workload to be tested should be clearly identified and defined [38].

During this research, a suite of benchmarks is created to compare the run-times of different data management implementations under the same work load. Different benchmarks stress different aspects of a system by making small adjustments to the workload, such as the transaction type, record count and the table properties. Various benchmark suites discussed in **Section 2.4** are taken in to consideration when finalizing the benchmark criteria.

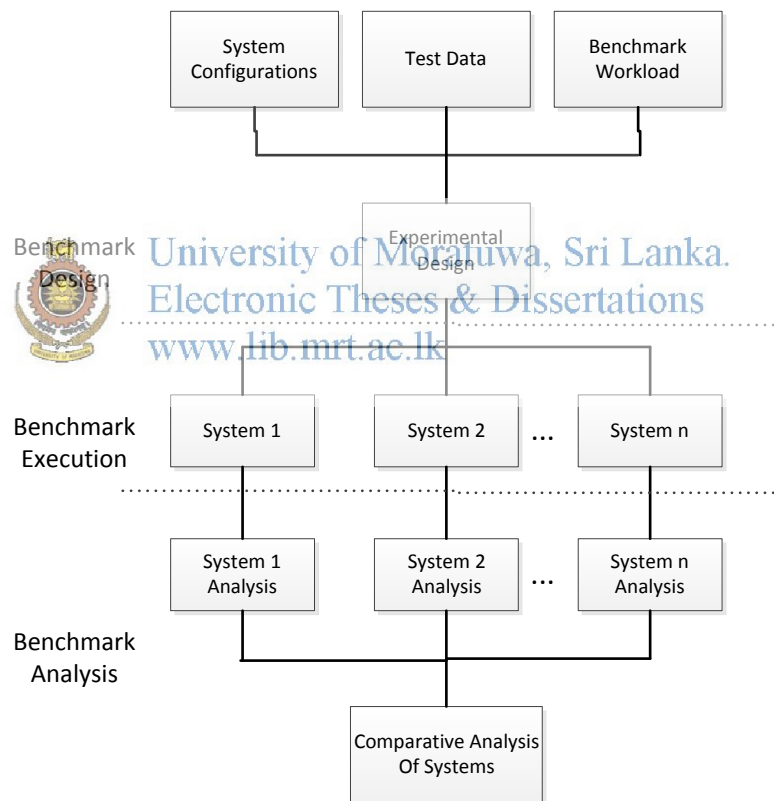


Figure 13 : Database System Benchmark Methodology

The benchmark methodology for database systems consists of three stages as Benchmark Design, Benchmark Execution and Benchmark Analysis. The **Figure 13** illustrates the methodology as a flow chart and the remainder of this chapter will discuss each phase in detail.

3.2.1 Benchmark Design

The benchmark design is the first step of benchmarking process and it is made up of four stages which provide input to the final step of experimental design. The design of a benchmark involved establishing the environment of the database system to be tested and developing the actual tests to be performed. These four areas of the benchmark design phase: system configuration, test data, benchmark workload and experimental parameters of this research project are discussed in this section.

3.2.1.1 System Configuration

To evaluate the selected databases and data caches hardware and software configurations given in **Table 3** are used in the test servers. The system configuration consists of a wide variety of parameters which relate to both hardware and software.

Table 3 : Benchmark System Configurations

| | |
|------------------|---|
| Operating System | Red Hat Enterprise Linux Server release 5.9 (Tikanga) |
| Memory Page Size | 4096 Bytes |
| CPU Speed | 2.70GHz 64 Cores |
| Main Memory | 64GB |
| Hard Disk | 4 TB |

3.2.1.2 Test Data

One of the major considerations in the benchmark experiment is that of what test data will be used for the testing. Theoretically there are two methods for obtaining a test database. That is either using an already existing application database or developing a synthetic database. For this research, an already existing application database is used and it was implemented on each of the candidate systems to be tested. The use of real data, has the advantage that it demonstrates database system performance on realistic application environments. So this is clearly the best method when the evaluation is done to select a system for a known database environment.

The TPC-C benchmark is used as a reference benchmark when designing the benchmark for this research and the application database has taken from the financial market domain. The test database contains data of stock symbols and their historical price data. The test database has two tables named as Tickers and History. Tickers table has the master details of the stock symbols of larger number of stock exchanges. The history table has trade price details of each stock on daily basis. The primary key for the tickers table is Ticker_serial which is a unique identifier for a stock symbol. The combination of the ticker_serial and the transaction_date is

used as the primary key for the history table. During this benchmarking process, these tables were created in each database under study and tables were populated with initial data required for each evaluation. The structure of these tables are given in **Table 4**.

Table 4 : Database Table Data

Tickers Table:

| Colum Name | Data Type |
|---------------------------|-----------|
| ticker_serial | Long |
| ticker_id | String |
| source_id | String |
| sector_id | String |
| market_id | String |
| source_ticker_id | String |
| instrument_type_id | int |
| currency_id | String |
| country_code | String |
| decimal_places | int |
| decimal_correction_factor | int |
| parent_ticker_id | String |
| parent_source_id | String |
| isin_code | String |
| lot_size | int |
| unit | String |
| display_ticker | String |
| comments | String |
| status | Character |
| tick_size | Double |
| last_updated_time | DateTime |

History Table:

| Colum Name | Data Type |
|------------------|-----------|
| ticker_serial | Long |
| transaction_date | DateTime |
| open | Double |
| high | Double |
| low | Double |
| close | Double |
| volume | Long |
| number_of_trades | Long |
| turnover | Double |
| vwap | Double |
| change | Double |
| pct_change | Double |
| prev_closes | Double |
| cf_in_count | Long |
| cf_in_volume | Long |
| cf_in_turnover | Double |
| cf_out_count | Long |
| cf_out_volume | Long |
| cf_out_turnover | Double |
| is_ann | int |
| news_provider | String |
| increment_id | Long |
| last_updated_on | DateTime |

3.2.1.3 Benchmark Workload and Experimental Design

In this important phase of the benchmark design, parameters were selected to be varied in the benchmark testing. During this research, the system throughput measured in queries per second is used as the principal performance metric. Where illustrative, response time has also been used as a performance indicator. The system performance was measured against the row count and the number of concurrent connections. The definitions of the performance metrics used here are as follows.

- System throughput - The average number of transactions (queries) processed per unit time.
- Response time - The time-to-last-record. i.e., from the time the query enters the system until the time the last record in the response is returned.

The details of the experimental design used for this benchmark is given in **Table 5**.

Table 5 : Performance Metrics

| Category | ID | Operation | |
|----------|------|---|---|
| Insert | OP1 | Number of Transactions vs RunTime | |
| | OP2 | Number of Transactions vs Transactions per second | |
| | OP3 | Number of Concurrent Connections vs Transactions Per Second | |
| Select | OP4 | Simple Select | Number of Transactions vs Runtime |
| | OP5 | | Number of Transactions vs Transactions per second |
| | OP6 | | Number of Concurrent Connections vs Transactions Per Second |
| | OP7 | Complex Select with Joins | Number of Transactions vs Runtime |
| | OP8 | | Number of Transactions vs Transactions per second |
| | OP9 | | Number of Concurrent Connections vs Transactions Per Second |
| Update | OP10 | Number of Transactions vs RunTime | |
| | OP11 | Number of Transactions vs Transactions per second | |
| | OP12 | Number of Concurrent Connections vs Transactions Per Second | |
| Delete | OP13 | Number of Transactions vs RunTime | |
| | OP14 | Number of Transactions vs Transactions per second | |
| | OP15 | Number of Concurrent Connections vs Transactions Per Second | |

3.2.2 Benchmark Execution

To evaluate the selected scenarios, a simple test tool has been implemented which can test different workload parameters and give the performance measures for each scenarios as the output.

Table 6 : Benchmark Tool Implementation Details

| Database | Language | Support Libraries |
|----------|----------|----------------------------------|
| Oracle | C++ | Oracle Call Interface (OCI) |
| SQLite | C++ | SQLite C/C++ Interface |
| MemSQL | C++ | MySQL Client and Drivers |
| H2 | JAVA | Java Database Connectivity(JDBC) |

The system time is read and recorded in log files immediately before and after each query is executed by each concurrently executing program. When all iterations of an experiment are concluded, each the measurements of the each program is analysed. The details of the

programming languages used to implement the benchmark tool and the support libraries used for each database is given in **Table 6**. To have more accuracy in the results, each operation has executed three times and the result is taken as the average of these three iterations.

3.2.3 Benchmark Analysis

After completing the evaluation, the gathered data was extracted from logs of the Benchmark Tool and the performance related comparisons were derived. During this phase the performance results on individual database systems were analysed and performance across different data management systems were compared. Graphs were plotted for each benchmark criteria and the details are given in the **Section 3.3**. This will result in an unbiased benchmark for various in-memory databases and their performance.

3.3 Results

The various benchmark tests discussed in **Section 3.2.1.3** have been carried out for each selected data management system and the results of each test is presented in this section. To have better result, each operation is repeated three times and average value of the three results is taken as the final result.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations

3.3.1 Results for Insert Operation

To evaluate the insert operation performance on selected DBs and caches, previously described tables were created in each database and the History table was populated with 1 million records at the beginning. Then at each iteration, defined number of rows varied from 1 to 5 million was inserted to history table and time taken for each set of transactions was recorded. To evaluate the performance with multiple concurrent connections, the same steps were done with several connections created using multiple threads. Sample Insert statement used in SQL based databases is given in **Figure 14**.

```

INSERT INTO
HISTORY
(TICKER_SERIAL, TRANSACTION_DATE, OPEN, HIGH, LOW, CLOSE, VOLUME,
NUMBER_OF_TRADES, TURNOVER, VWAP, CHANGE, PCT_CHANGE, PRV_CLOSED,
CF_IN_COUNT, CF_IN_VOLUME, CF_IN_TURNOVER, CF_OUT_COUNT, CF_OUT_VOLUME,
CF_OUT_TURNOVER, IS_ANN, NEWS_PROVIDER, INCREMENT_ID)
VALUES
(1421, TO_DATE('1994-07-22 20:14:40', 'YYYY-MM-DD HH24:MI:SS'),
10.21, 14.64, 9.32, 11.43, 11465, 238, 7186932.25, 12.94, -0.5, 1.61, 11.94,
120, 8435, 64235.65, 118, 4564, 35245.45, 0, 'ALSHAMIL', 10043);

```

Figure 14: Example Insert Statement

OP1: Number of Transactions vs Run Time

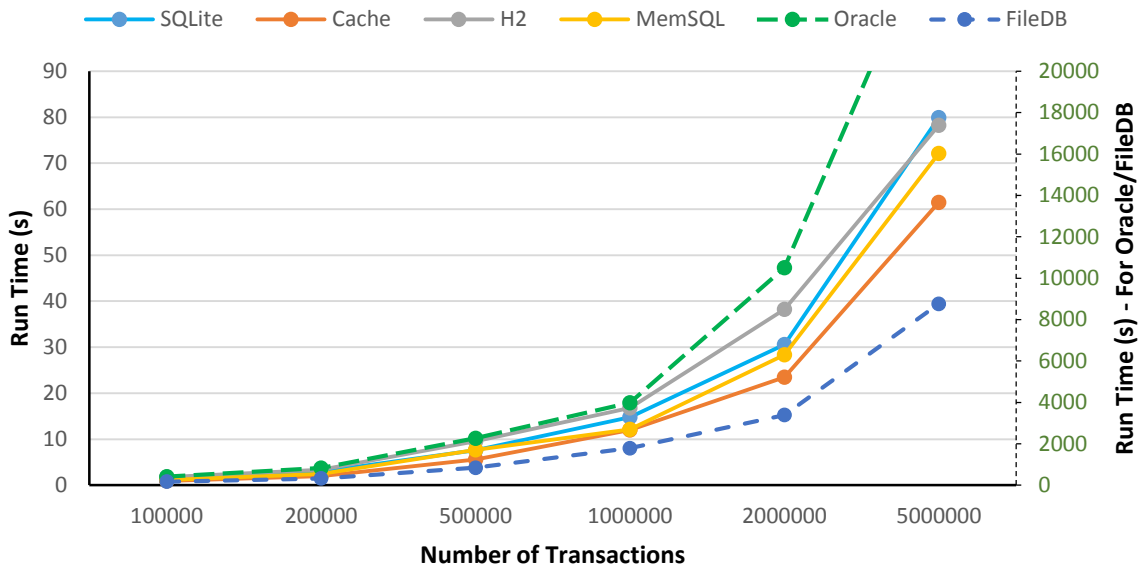


Figure 15 : Insert Operation -Run Time Comparison

OP2: Number of Transactions vs Transactions per Second

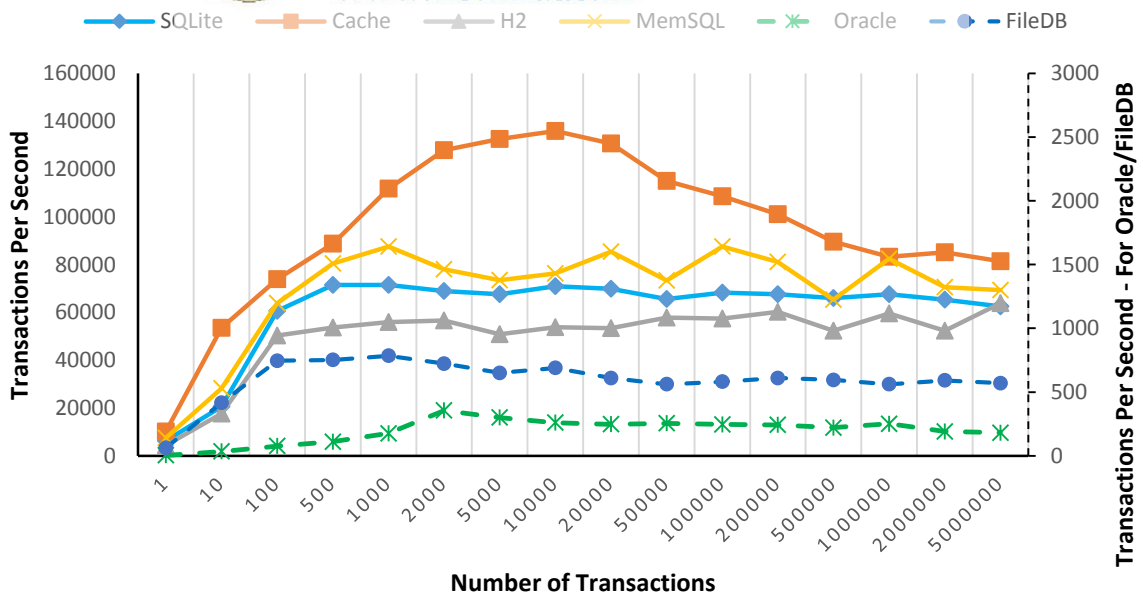


Figure 16 : Insert Operation - Transactions per Second Comparison

OP3 : Number of Concurrent connections vs Transactions per second

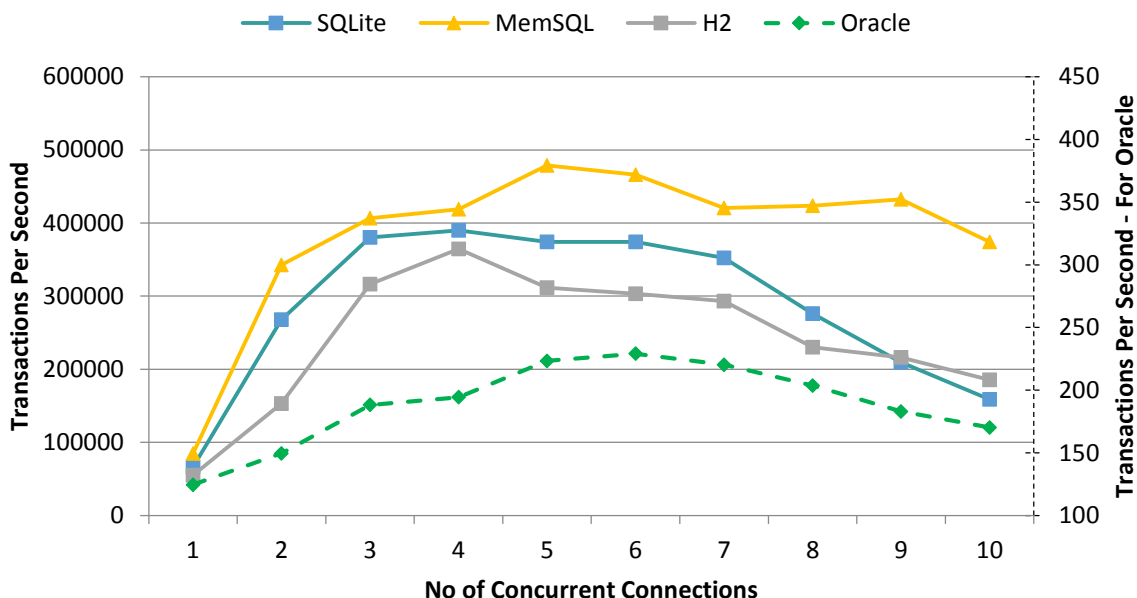


Figure 17 : Insert Operation - Concurrent Connections vs TPS

When analysing the insert operation results, we can clearly see that IMDBs has higher performance than the Oracle DRDB. For insert operation IMDBs are around 200 times faster than the Oracle. Flat file based has higher transactions per second (TPS) than oracle for small number of inserts. But as the number of records growth, the TPS of flat file based DB gradually reduced.

Out of the three IMDBs selected, MemSQL has the highest TPS. SQLite has the next highest TPS and H2 has the lowest. IMDBs are around 100 times faster than the flat file database for smaller number of inserts. In-memory data cache has the highest performance for any transaction count. Cache is around 500 times faster than Oracle and 2 times faster than IMDBs.

MemSQL is built from the ground up to take advantage of modern hardware, leveraging dozens of cores per machine, terabytes of memory, and horizontal scale-out on commodity hardware. SQLite and other in memory databases are same as the disk-based one which is paged, and the only difference is that the pages are never written to disk. So this disk I/O overhead is not present in these databases.

Other than the disk residency, one cause of poor performance in Oracle is high communication overhead. Oracle must process SQL statements one at a time. Thus, each statement results in another call to Oracle and higher overhead. In a networked environment, SQL statements must

be sent over the network, adding to network traffic. Heavy network traffic can slow down the application significantly.

3.3.2 Results for Select Operation

Select operation was evaluated with Simple Select queries and Complex select queries with table joins. To evaluate the simple Select operation performance on selected DBs and caches, previously described tables were created in each database and the History table was populated with 10 million records at the beginning. Then at each iteration, defined number of rows varied from 1 to 5 million was selected from history table and time taken for each set of transactions was recorded.

To evaluate the Join operation performance, both Tickers table and History table were used. Tickers table was populated with 1 million records which corresponds to 1 million stock symbols. Then history table was populated with 10 million records which corresponds to the history data of the symbols in tickers table. Then at each iteration, defined number of rows varied from 1 to 5 million was selected by joining both history table and tickers table and time taken for each set of transactions was recorded.

To evaluate the performance with multiple concurrent connections, the same steps were done with several connections created using multiple threads. Sample SQL statements used for SQL based databases is given in **Figure 18**. File based DB is not used in Join operation test since Join operation is not currently implemented in File based DB. For In memory data cache, exact match selection procedure was taken as the equivalent for Select operation. It also not included in Join statement test since no join operation is defined in the cache implementation.

Simple Select Statement:

```
SELECT * FROM history WHERE volume = 54343;
```

Select Statement with Joins:

```
SELECT * FROM history h
LEFT JOIN tickers t
ON t.ticker_serial = h.ticker_serial
WHERE h.volume = 54343 and t.source id = 'NSDQ'
```

Figure 18 : Example Select Statement

OP4 : Number of Transactions vs Run Time (Simple Query)

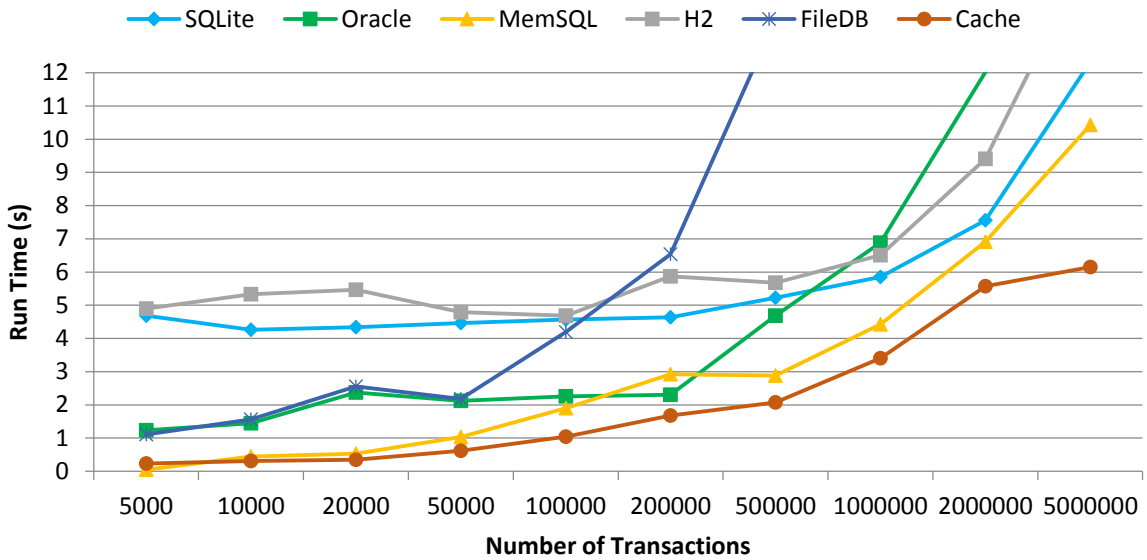


Figure 19 : Select Operation - Run Time Comparison

OP5: Number of Transactions vs Transactions per Second (Simple Query)

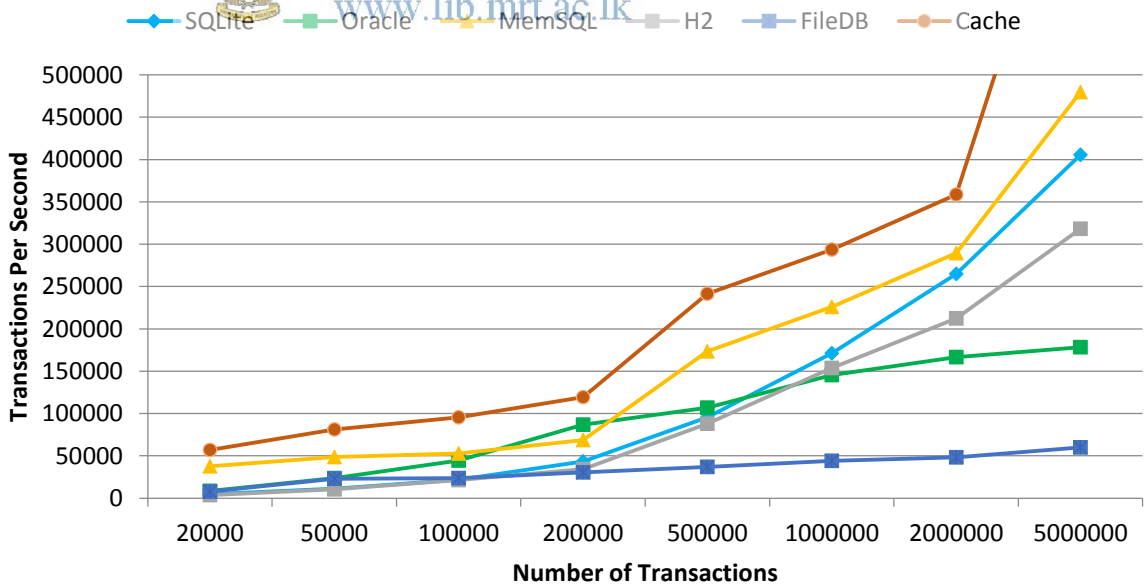


Figure 20 : Select Operation – Transactions Per Second Comparison

OP6 : Number of Concurrent connections vs Transactions per second (Simple Query)

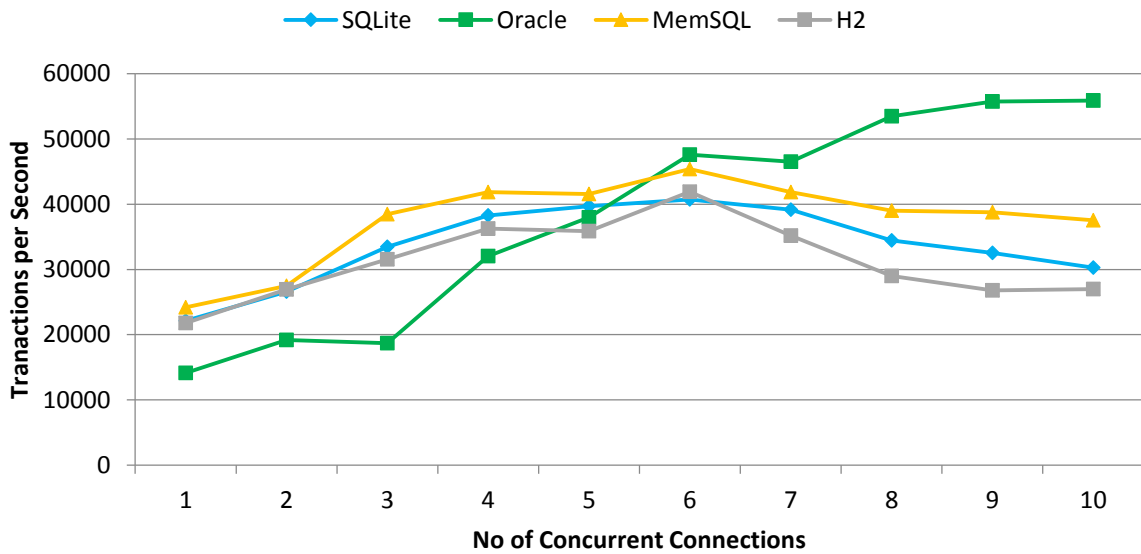


Figure 21: Select Operation - Concurrent Connections vs TPS

OP7: Number of Transactions vs Transactions per Second (With Joins)

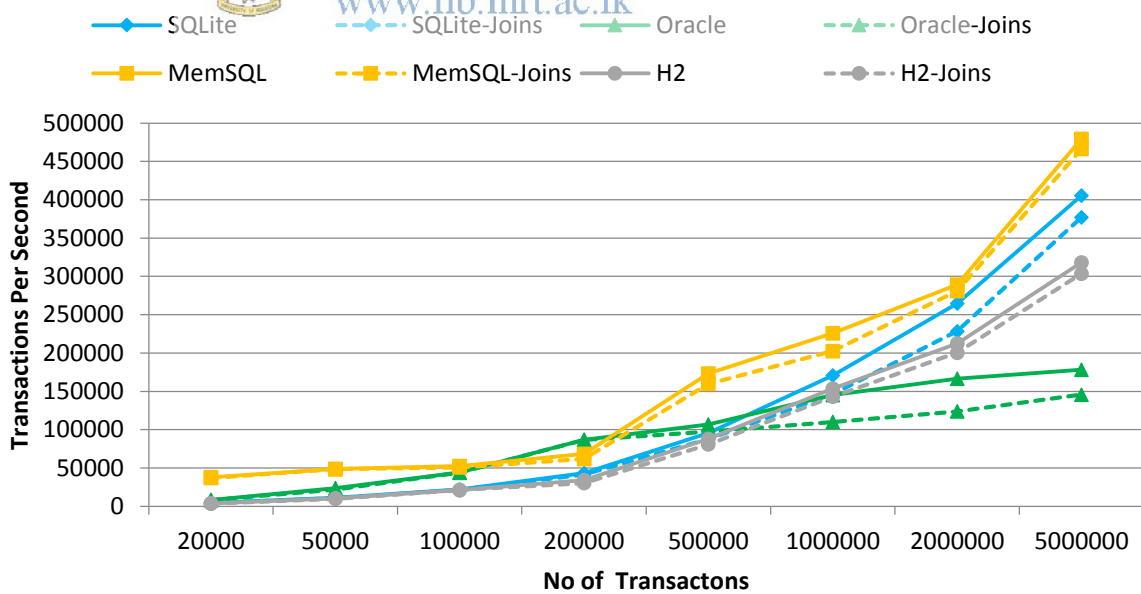


Figure 22 : Select with Joins - TPS Comparison

When analysing the Select operation results, we can clearly see that IMDBs has higher performance than the Oracle DRDB when the number of select operations are higher. For select operation IMDBs are around 2 times faster than the Oracle. Flat file based has less transactions

per second (TPS) than all the other data sources. Oracle DB is around 3 times faster than the Flat File DB and IMDBs are around 8 times faster than it. In-memory cache is around 15 times faster than it. Out of the three IMDBs selected, MemSQL has the highest TPS. SQLite has the next highest TPS and H2 has the lowest. In-memory data cache has the highest performance for any transaction count. Cache is around 5 times faster than Oracle and 1.5 times faster than IMDBs.

Transactions with join operations shows same curve shape but has less TPS than the simple select operations for all databases. The TPS difference between the two curves is significant in Oracle database. Join operation performs within main memory is faster than the disk based operation.

3.3.3 Results for Update Operation

To evaluate the update statement performance, History table was initially populated with 10 million records and at each iteration, defined number of rows varied from 1 to 5 million was updated from history table. To evaluate the performance with multiple connections, the same test was done with multiple threads with each thread creating a new connection to the database. Sample SQL statements used for SQL based databases is given in Figure 23.

```
UPDATE history SET volume = 50000 WHERE volume = 54243;
```

Figure 23 : Example Update Statement

OP8: Number of Transactions vs Run Time

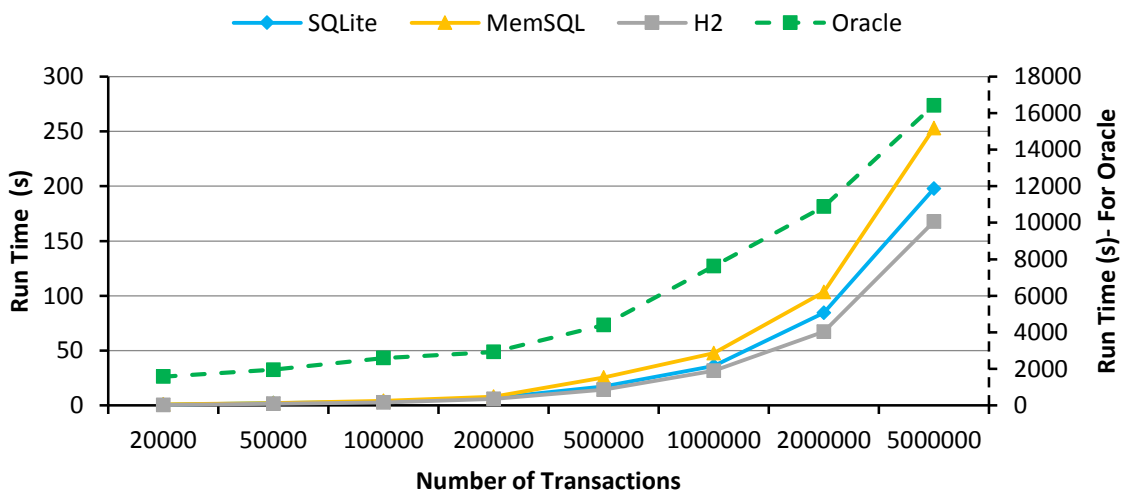


Figure 24 : Update Operation - Run Time Comparison

OP9: Number of Transactions vs Transactions per Second

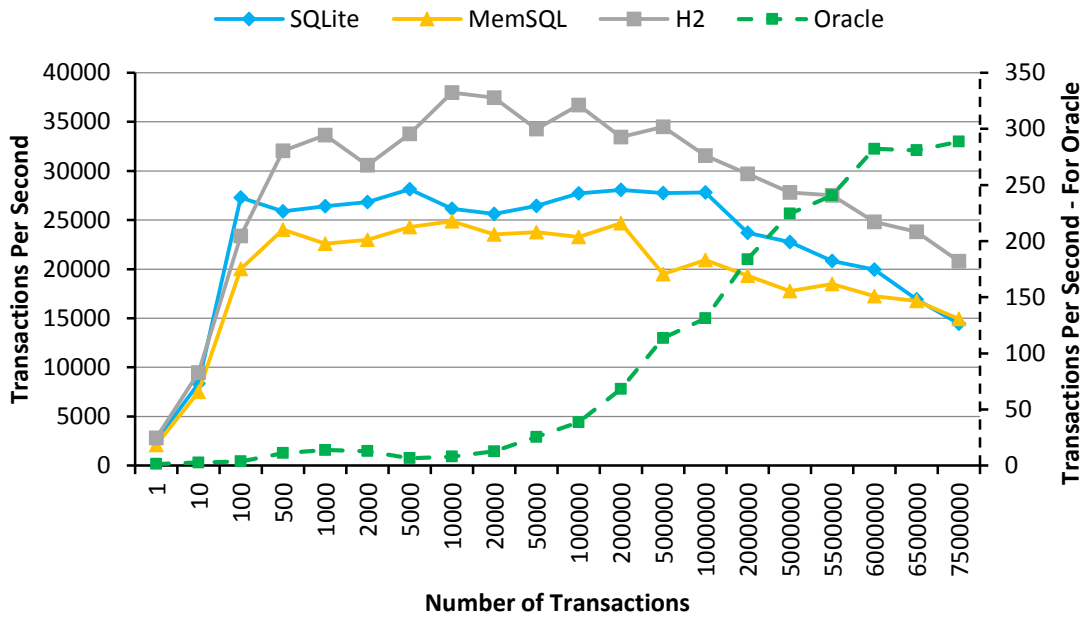


Figure 25 : Update Operation - Transactions Per Second Comparison

OP10 : Number of Concurrent connections vs Transactions per second

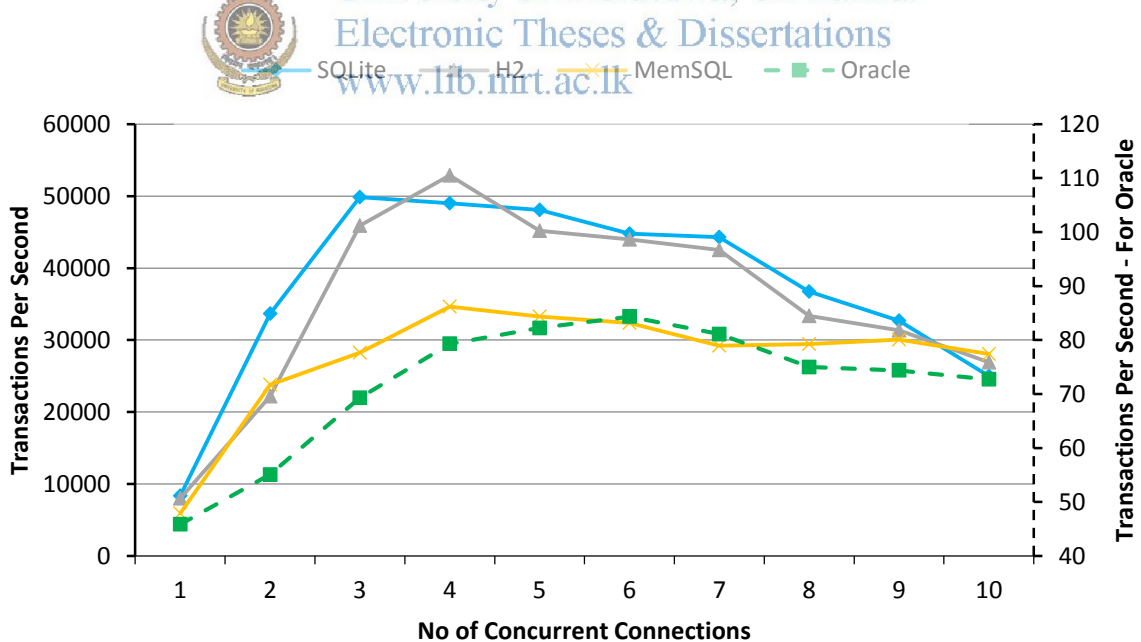


Figure 26 : Update Operation - Concurrent Connections vs TPS

When analysing the Update operation results, we can see that IMDBs has higher performance than the Oracle DRDB. For update operation IMDBs are around 80 times faster than the Oracle. Out of the three IMDBs selected, H2 has the highest TPS for update operation. SQLite has the

next highest TPS and MemSQL has the lowest TPS. For In-memory databases the transactions per second become nearly constant even the number of transactions increasing. For Oracle, TPS gradually increasing when the number of transactions increasing.

When number of concurrent connections are increasing the TPS of SQLite and H2 databases are gradually decreased after showing a peak value when number of concurrent connections are at 3 and 4 respectively. But for Oracle and MemSQL TPS remains nearly constant when number of concurrent connections increasing.

3.3.4 Results for Delete Operation

To evaluate the Delete statement performance, History table was initially populated with 10 million records and at each iteration, defined number of rows varied from 1 to 5 million was deleted from history table. To evaluate the performance with multiple connections, the same test was done with multiple threads with each thread creating a new connection to the database. Sample SQL statements used for SQL based databases is given in **Figure 27**.



OP11: Number of Transactions vs Run Time

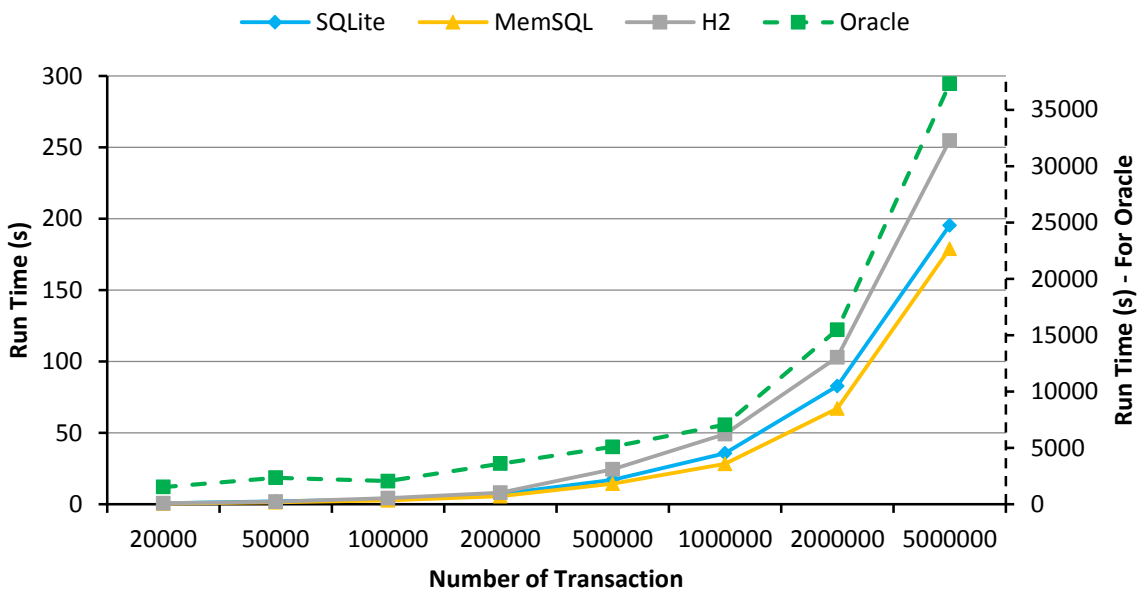


Figure 28: Delete Operation -Run Time Comparison

OP12: Number of Transactions vs Transactions per Second

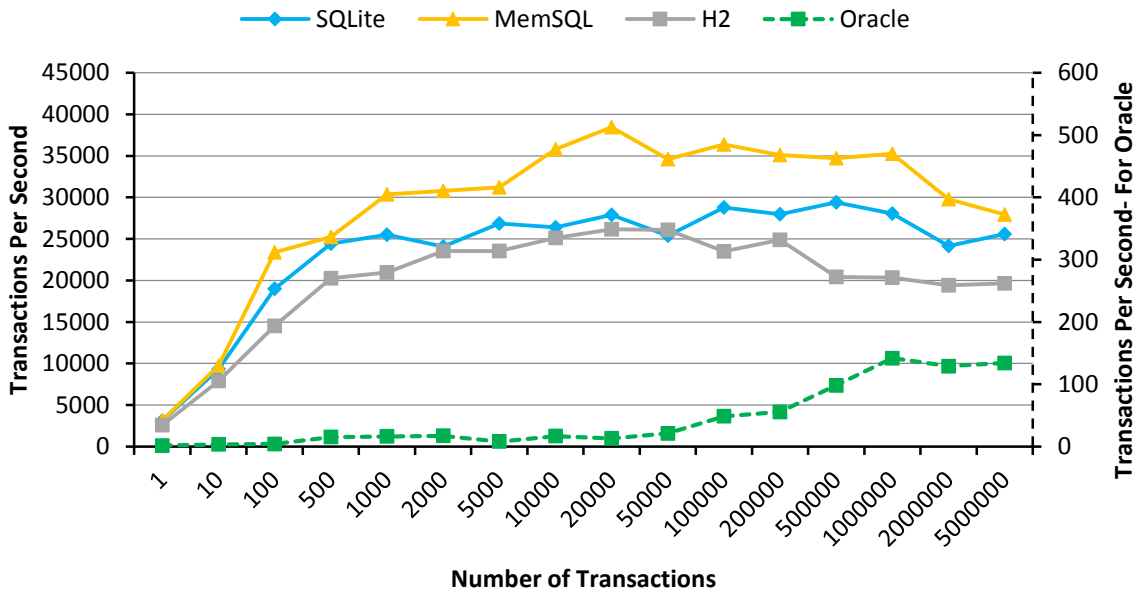


Figure 29 : Delete Operation - Transactions Per Second Comparison

OP13: Number of Concurrent connections vs Transactions per second

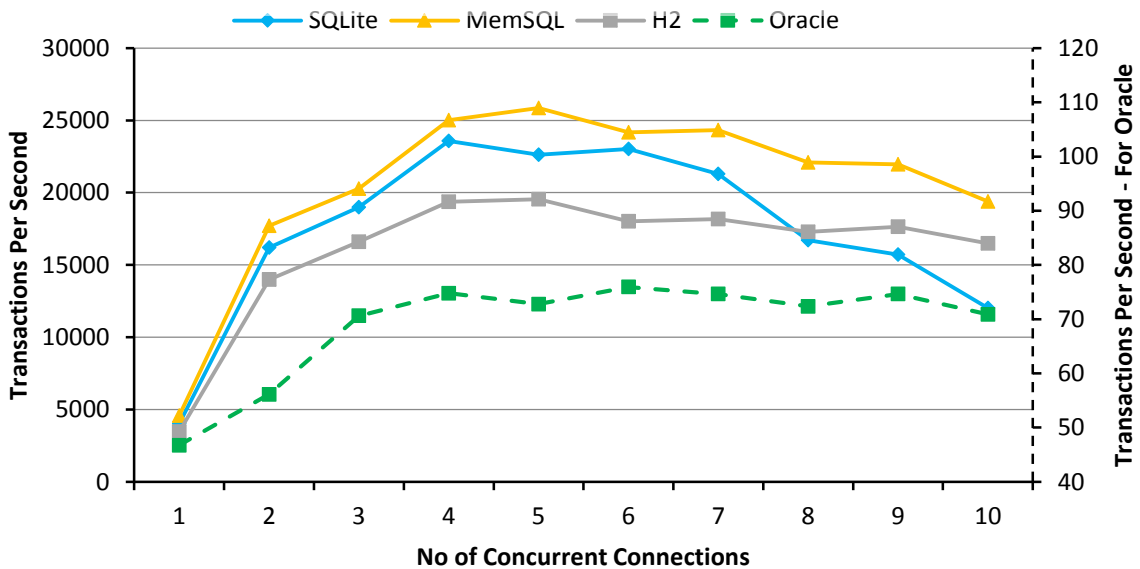


Figure 30 : Delete Operation - Concurrent Connections vs TPS

As with the other operations, for delete operation also IMDBs has higher performance than the Oracle DRDB. For delete operation, IMDBs are around 250 times faster than the Oracle. Out

of the three IMDBs selected, MemSQL has the highest TPS for update operation. SQLite has the next highest TPS and H2 has the lowest TPS. For all IMDBs the TPS become gradually increasing for small number of transactions, and then become a flat graph. For delete operation, Oracle also shows a similar behaviour.

When the number of concurrent connections are increasing, the TPS gradually increasing for all databases up to 4-5 number of connections and then remains constant. But for SQLite when number of concurrent connections exceed 7, TPS gradually decreased.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

4. FRAMEWORK IMPLEMENTATION

This chapter is intended to illustrate the complete implementation effort of the data management framework proposed by this research, highlighting the important design decisions made during implementation phase. While **Section 4.1** gives a brief summary of the requirement of such a framework, **Section 4.2** will illustrate further on implementation of In-memory data cache, File based DB and finally the data management Framework.

In **Section 4.3** the details of the performance comparison of the framework based implementation and pure database API calls based implementation is given. There the framework is evaluated against the results presented in Section 3.3.

4.1 Problem Background

In almost all the enterprise applications written today, it is required to incorporate database CRUD (Create, Read, Update, and Delete) operations. A large enterprise application will typically have one or more databases to store data and on top of this a data access layer (DAL) to access the database. On top of this there may be some repositories to communicate with the DAL, a business layer containing logic and classes representing the business domain, a service layer to expose the business layer to clients and finally some user interface application such as a desktop application or an web application.

Many developers often make database calls directly from an application layer, but this results in maintenance or code change is extremely difficult when database access changes are necessary. As with any application development endeavour, there is more than one way to tackle it. A current industry trend is to separate the data access code from the rest of the code. With this approach, it is possible to use the necessary database calls via the data access code. This allows the developer to make database access or code changes without touching the rest of an application. So a data access layer is an important part of a software application.

A data access layer follows the idea of "separation of concerns" where all of the logic required for the business logic to interact with the data sources is isolated to a single set of classes (layer). This allows developers to more easily change the back-end physical data storage technology without having a large impact to the business logic.

The standard for cross platform SQL database connectivity is Open Database Connectivity (ODBC) which a standard database access method developed by the SQL Access group in

1992. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS [39]. ODBC is more than a database interface, it also defines an underlying connection protocol etc. So the application developers has to deal with the code complexity associated with ODBC when connecting with the database. Although there are several C++ wrappers and libraries for it, there is no widely used free API for this. Another limitation with these libraries is they are mainly focused on SQL based databases and other forms of data sources such as file based DB, in-memory caches are not addressed.

So the proposed framework will address these problems and it is implemented as a C++ library to access various data sources such as SQL based in-memory and disk based databases, flat file databases and in-memory data caches. Since it is implemented in an extensible way, support for any other new data source can be integrated with it. The details of the implementation of the framework is given in the following sections of this chapter.

4.2 Design of the Framework

A data access layer (DAL) in computer software, is a layer of a computer program which provides simplified access to data stored in persistent storage of some kind, such as an entity-relational database. It is an application programming interface which unifies the communication between a computer application and databases. Traditionally, all database vendors provide their own interface tailored to their products, which leaves it to the application programmer to implement code for all database interfaces developer would like to support. Database abstraction layers reduce the amount of work by providing a consistent API to the developer and hide the database specifics behind this interface as much as possible. This approach provides flexibility to change an application's persistence mechanism over time without the need to re-engineer application logic that interacts with the data access layer.

The high-level logical diagram for the proposed Data Connector Framework is shown in **Figure 31**. The presentation layer is what a system user sees or interacts with. It can consist of visual objects such as screens, web pages or reports or non-visual objects such as an interactive voice recognition interface. To provide the required functionalities to the client, the application needs to interact with the Data Layer. The business logic layer represents the business rules that are enforced via programming logic regarding how those rules are applied.

The data access layer consists of the definitions of database tables and columns and the computer logic that is needed to navigate the database.

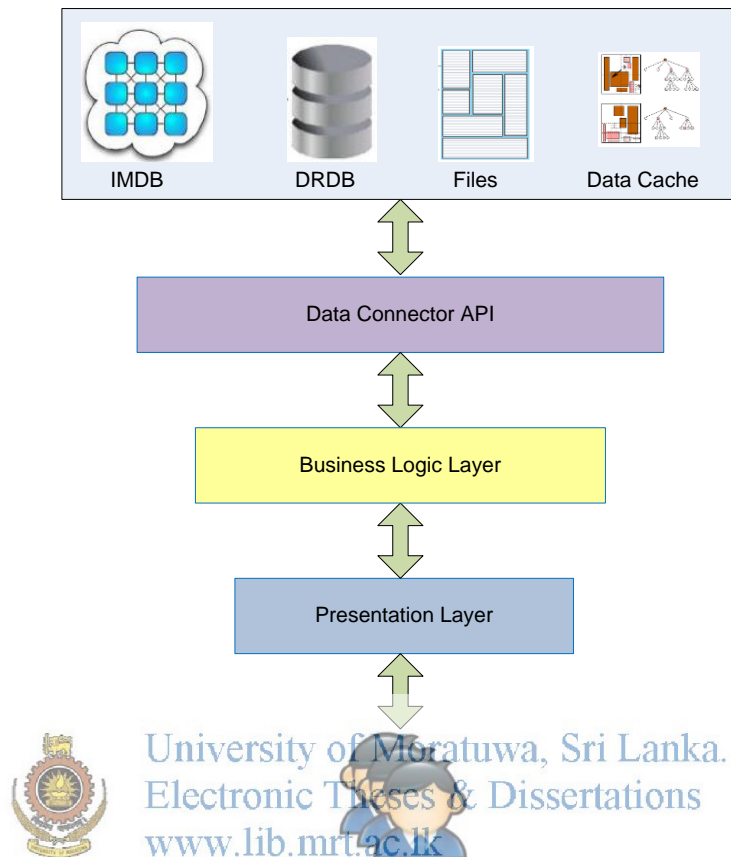


Figure 31 : Proposed Architecture for Database API

The data can be stored in various forms such as in-memory database, disk resident database or simple flat files. In the current enterprise applications the application layer is tightly coupled with the data layer and the data storage method cannot be changed later based on the business requirements. The proposed solution is a database connector API which provides a seamless interface for the application developers so that the underlying data storage mechanism does not affect the application interface. The connector API will provide all the required functionalities for either IMDB, DRDB or flat files so that all data handling logics will be excluded from the application layer.

4.3 Implementation Details

The proposed data connector framework in this research is a C++ library for accessing multiple SQL based disk resident and in memory databases, flat file database and in memory data cache. It uses native APIs of target data source so applications developed with this framework library run swiftly and efficiently. This library acts as middle-ware and delivers database portability across various different data sources. The In-memory data cache and the flat file database is developed using C++ language and both of them were integrated with the data connector framework. The following sections of this chapter describes the implementation details of the in-memory cache, flat file database and finally the data connector framework along with their features and design.

4.3.1 Implementation of Flat File based DB

The flat file database system is implemented based on File Input Output processing and Streams. To access the structure of the data and manipulate it, the file is read in it's entirely into the computer's memory. The database is a system folder with the given database name, which is created in the predefined database location within the file system. In this system the tables are holding all the data in the form of flat files. Organization of databases and tables are shown in **Figure 32**.

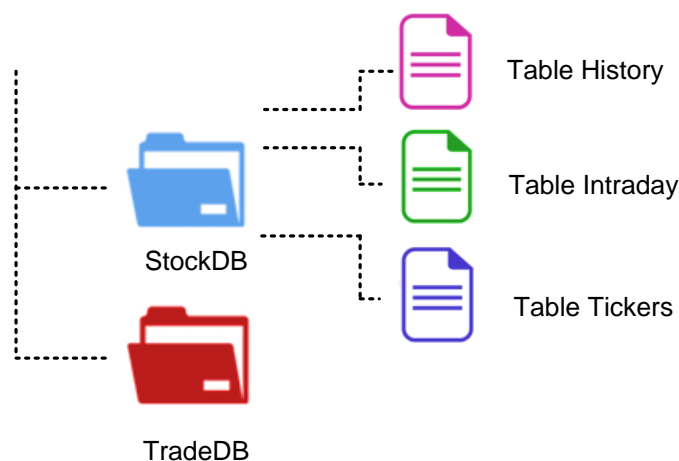


Figure 32: Database organization in Flat File DB

In this System, the table structure has two parts as header rows and data rows. Header Row consists the column names. Data Rows consist the records related to the columns. Some special symbols are used as a delimiters for data columns.

```

void FileDB::ExecuteQuery()
{
    std::vector<String> vecQueryWords;

    String zQuery = p_FileDBStmt->GetQuery();
    CreateDataArrayFromString(vecQueryWords,zQuery,' ');

    if(vecQueryWords[0] == "CREATE")
    {
        if (vecQueryWords[1] == "DATABASE")
            CreateDB(vecQueryWords[2]);
        else if (vecQueryWords[1] == "TABLE")
            CreateTable(vecQueryWords[2],p_FileDBStmt->GetDatabase(),vecQueryWords[3]);
    }
    else if(vecQueryWords[0] == "USE")
    {
        UseDB(vecQueryWords[1]);
    }
    else if(vecQueryWords[0] == "DELETE")
    {
        if (vecQueryWords.size() >= 5)
        {
            std::vector<String> vecOperatorData = SplitByString(vecQueryWords[4],"=");
            DeleteRecord(vecQueryWords[2], p_FileDBStmt->GetDatabase(), vecOperatorData[0],
                vecOperatorData[1]);
        }
    }
    else if(vecQueryWords[0] == "INSERT")
    {
        String zColumnArray = vecQueryWords[3];
        zColumnArray.ReplaceAll("(", "");
        zColumnArray.ReplaceAll(")", "");

        String zValueArray = vecQueryWords[5];
        zValueArray.ReplaceAll(",");
        zValueArray.ReplaceAll("'", "");

        InsertRecord(p_FileDBStmt->GetDatabase(),vecQueryWords[2],zColumnArray,zValueArray);
    }
    else if(vecQueryWords[0] == "SELECT")
    {
        String zColumnArray = vecQueryWords[1];
        String zTableName = vecQueryWords[3];

        std::vector<String> vecClauseData = SplitByString(zQuery,"WHERE");
        String zWhereClause = vecClauseData[0];

        bool bOpFound = false;
        String zOperator;
        GetOperatorFromClause(zWhereClause,bOpFound,zOperator);
        String zComparedColum,zComparedValue;

        if(bOpFound)
        {
            std::vector<String> vecOpData = SplitByString(zWhereClause,zOperator);
            zComparedColum = vecOpData[0];
            zComparedValue = vecOpData[1];
        }

        std::vector<DBRecord*> vecRes = SelectRecords(p_FileDBStmt->GetDatabase(),zTableName,
            zOperator,zComparedColum,zComparedValue);
    }
    else
        std::cout<<"invalid operation"<<std::endl;
}

```

Figure 33 : Query Execution Method of Flat File DB

The record and column separators used in this flat file database is as follows. Example table is shown in **Figure 34**.

- Record Begin – Hex 2: STX (Start of Text)
- Record End - Hex 3: ETX (End of Text)
- Column Separator – ‘|’ Pipe

```
STX TICKER_SERIAL|TXN_DATE|OPEN|HIGH|LOW|CLOSE|VOLUME|T|OVER|VWAP|CHANGE|PCT_CHG|PREV_CLOSED|NEWS_PRV|INC_ID|ETX
STX 1421|2013/3/27:20:26:11|10.42|11.32|10.31|11.02|6302|32125.32|10.95|1.53|0.32|10.92|DFNS|1000|ETX
STX 1421|2013/4/29:20:27:11|10.43|11.02|10.23|10.92|11302|64125.32|11.95|1.53|0.62|11.92|DJNS|1001|ETX
```

Figure 34 : Flat File DB - Table Data

The flat file database system is implemented using C++ language and Input/output stream class to operate on files. The queries are implemented in a similar way to standard SQL query language and Create, Insert, Delete and Select and Drop statements are supported in the current version and the DB query execution method of flat file DB is given in **Figure 33**.

4.3.2 Implementation of In-Memory Cache

The C++ Standard Template Library (STL) is a powerful and versatile collection of classes and functions that provides an efficient, lightweight, and extensible framework for application development. STL also offers a sophisticated level of abstraction that promotes the use of generic data structures and algorithms without the overhead of a generic solution. A STL container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements. The in-memory cache is developed using these STL containers and maps, sets, lists, arrays and vectors have been extensively used for that. The main features of this cache is as follows.

- Able to define Data tables, data rows and cells so that data organization look similar to traditional database.
- Able to define primary key columns for tables.
- Provide support for indexing for faster access operations.
- Provide support for data types including bool, int, long, float, double and DateTime.
- Able to query the data table for various operations including exact match, partial match, greater than, less than and between.
- Able to delete records based on given criteria.
- Able to clear tables and delete tables and alter tables by adding new columns.

When using this in-memory cache, first the cache tables need to define. Table column names, their data types and primary key columns for the tables are initially defined. Then data records can be added to each table by setting values for each column of the record. Then these cache tables can be queried for various operations such as exact match, greater than, less than etc. Example usage of this in-memory cache is shown in **Figure 35**.

```

//Define Table Columns
CacheColumnInfo* pInfo0 = new CacheColumnInfo("TICKER_SERIAL", CACHE_DATA_TYPE_LONG);
CacheColumnInfo* pInfo1 = new CacheColumnInfo("TRANSACTION_DATE", CACHE_DATA_TYPE_DATE_TIME);
CacheColumnInfo* pInfo2 = new CacheColumnInfo("OPEN", CACHE_DATA_TYPE_DOUBLE);
CacheColumnInfo* pInfo3 = new CacheColumnInfo("HIGH", CACHE_DATA_TYPE_DOUBLE);
CacheColumnInfo* pInfo4 = new CacheColumnInfo("LOW", CACHE_DATA_TYPE_DOUBLE);
CacheColumnInfo* pInfo5 = new CacheColumnInfo("CLOSE", CACHE_DATA_TYPE_DOUBLE);

std::vector<CacheColumnInfo> vec_ColInfo;
vec_ColInfo.push_back(*pInfo0);
vec_ColInfo.push_back(*pInfo1);
vec_ColInfo.push_back(*pInfo2);
vec_ColInfo.push_back(*pInfo3);
vec_ColInfo.push_back(*pInfo4);
vec_ColInfo.push_back(*pInfo5);

//Define primary keys
std::vector<String> vec_KeyCols;
vec_KeyCols.push_back("TICKER_SERIAL");
vec_KeyCols.push_back("TRANSACTION_DATE");

//Create Cache and Table History
Cache* pCache = new Cache();
CacheTable* pTableHistory = pCache->CreateTable("TABLE_HISTORY",vec_ColInfo,vec_KeyCols);

//Adding Data Records
long lRowCount = 10;
DateTime dtTransactionDate.SetDateTime(2014,2,1,23,12,55);
long iVal = 0;
while(iVal < lRowCount)
{
    CacheRecord* pCacheRec = pTableHistory->GetEmptyRecord();
    pCacheRec->Get(0).Set(iVal);
    dt_TransactionDate.AddMinutes(1);
    pCacheRec->Get(1).Set(dt_TransactionDate);
    pCacheRec->Get(2).Set(10.12);
    pCacheRec->Get(3).Set(12);
    pCacheRec->Get(4).Set(8.21);
    pCacheRec->Get(5).Set(9.32);
    pTableHistory->Insert(pCacheRec);
    ++iVal;
}

//Fetching Row Count and Exact Match Results
int iRecordCount = pTableHistory->GetRecordCount();
std::set<CacheRecord*> setResults;
pTableHistory->QueryForExactMatch(0, 2, setResults);

```

Figure 35 : Example usage of In-Memory Cache

The implementation of in-memory cache has four basic classes namely Cache, CacheTable, CacheRecord and CacheCell. The detailed class view of the system is shown in **Figure 36**.

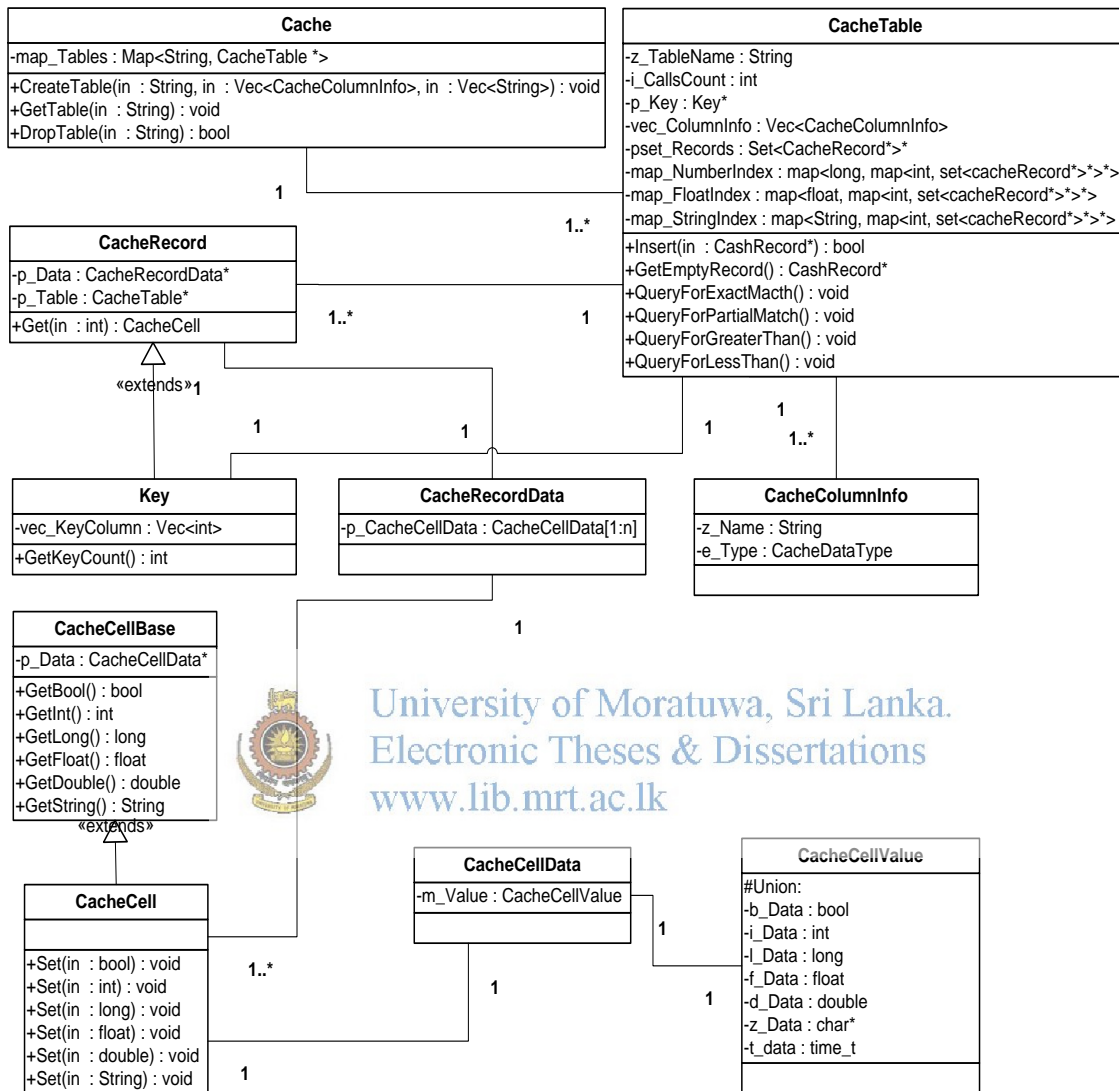


Figure 36: Class Diagram of In-Memory Cache

4.3.3 Implementation of the Framework for Data

The data connector API is designed to create a development experience that insulates application developers from being domain experts in the data persistence layer. This allows database experts to optimize interaction with the persistence layer without impacting the application development process. The decoupling was obtained by defining a set of interfaces setting the contracts for retrieving and persisting objects. This data connector framework is designed as a C++ dynamic library and it directly calls native API's of target data source. The features of the framework is as follows.

- Provide support for Oracle, SQLite, Flat File DB and In-Memory cache and designed in an extensible way so that new data sources can be added at any point.
- The procedures for database connection creation and query execution are simplified and the developers are not directly exposed to complex database specific code. So this framework reduce the developer effort and time.
- Since it is designed as a C++ library, it can be easily integrated with enterprise applications
- Provide support for Select and other non-query operations; Insert, update and delete
- Data source and connection parameters can be configured in XML based configuration file. So the underline data source of the enterprise applications can be changed with no code changes.

In this framework, the DataConnector class is the one which reads the data source configuration files and initialize the defined data connection in that file. The DataConnection is the base class for all the data sources and it has two basic methods as ExecuteQuery and ExecuteNonQuery. All the sub classes which corresponds to different data sources inherit these two methods and implement them using the data source specific API methods. So all the database specific method calls and other complex data structures required are used only at this level and it has simplified the application developer's effort.

To provide a generic result set for all select queries in different data sources, several wrapper classes are used to wrap the selected data. At the application level, developer has to iterate this generic ResultSet class to get the results for a particular query. The DBRecord and DBField corresponds to data value and data record in database. The underline data structure used to store different data values is a union. So it is possible to save memory by using the same memory region for storing different objects at different times. The detailed class diagram of the

framework is given in **Figure 37** and SQLite query execution method is shown in **Figure 38**.

An example code which shows how this framework can be used is given in **Figure 39**.

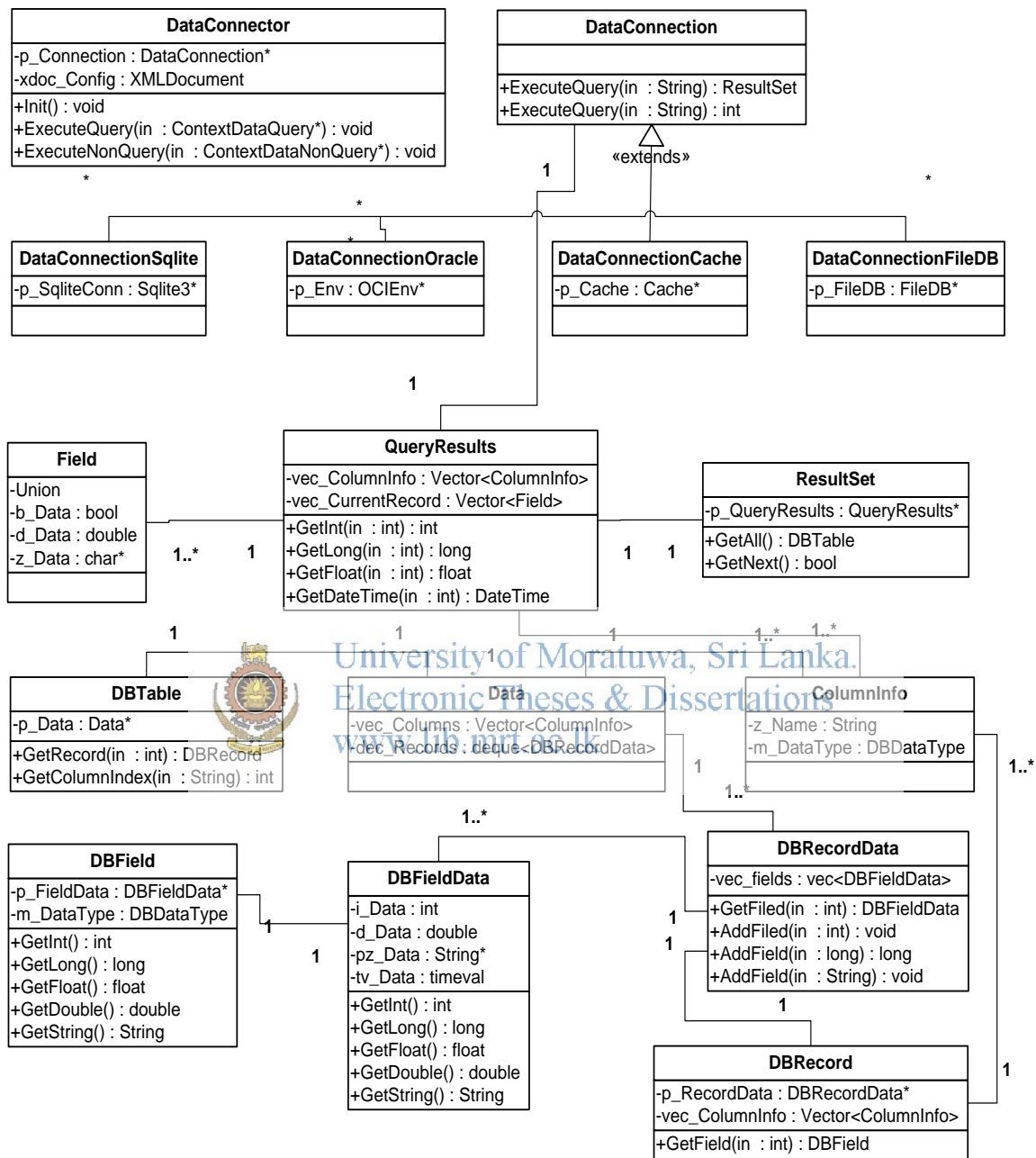


Figure 37 : Class diagram of Data Connection Framework

```

ResultSet DataConnectionSQLite::ExecuteQuery(String _zSQL)
{
    //Prepare Query Statement
    sqlite3_stmt* pStmt;
    const char *tail2;
    int iError = sqlite3_prepare(p_SQLiteConn, _zSQL.GetCString(), _zSQL.GetLength(), &pStmt, &tail2);
    if (iError != SQLITE_OK)
    {
        return NULL;
    }

    //Initialize query results.
    SQLiteQueryResults* pQueryResults = new SQLiteQueryResults(this, pStmt);

    //Get Column Count
    int col_cnt = sqlite3_column_count(pStmt);
    fprintf(stdout, "%d columns\n", col_cnt);
    int rowcount = 0;

    //Execute the Query
    sqlite3_step(pStmt);
    for (int i=0; i<col_cnt; i++)
    {
        DBDataType dtColumn = DB_DATA_TYPE_NONE;

        //Get column data type and the name
        int iDataType = sqlite3_column_type(pStmt,i);
        const char* zColumnName = sqlite3_column_name(pStmt, i);
        String zColumnName(zColumnName);
        switch(iDataType)
        {
            case SQLITE_INTEGER:
                dtColumn = DB_DATA_TYPE_INT;
                break;
            case SQLITE_TEXT:
                dtColumn = DB_DATA_TYPE_STRING;
                break;
            case SQLITE_FLOAT:
                dtColumn = DB_DATA_TYPE_FLOAT;
                break;
            case SQLITE_BLOB:
                dtColumn = DB_DATA_TYPE_BLOB;
                break;
            default:
                break;
        }

        pQueryResults->vec_ColumnInfo.push_back(ColumnInfo(zColumnName, dtColumn, iDataLength, 1000));

        fprintf(stdout, "%d. %s|%d\n", i, sqlite3_column_name(pStmt, i),sqlite3_column_type(pStmt,i));
    }

    // Define local variables
    for(unsigned int j = 0; j < col_cnt; ++j)
        pQueryResults->vec_CurrentReocrd.push_back(SQLiteQueryResults::Field());

    std::vector<ColumnInfo>::iterator itrInfo = pQueryResults->vec_ColumnInfo.begin();
    std::vector<ColumnInfo>::iterator itrInfoEnd = pQueryResults->vec_ColumnInfo.end();

    std::vector<SQLiteQueryResults::Field>::iterator itrDefine = pQueryResults->vec_CurrentReocrd.begin();

    //Allocate data for fields
    int k = 1;
    while(itrInfo != itrInfoEnd)
    {
        int iLength = 0;
        DBDataType dtColumn = itrInfo->GetDataType();
        switch(dtColumn)
        {
            case DB_DATA_TYPE_INT:
            case DB_DATA_TYPE_LONG:
            case DB_DATA_TYPE_FLOAT:
            case DB_DATA_TYPE_DOUBLE:
                break;
            case DB_DATA_TYPE_STRING:
                iLength = itrInfo->GetMaxDataLength();
                iLength *= 3;
        }
    }
}

```

```

        itrDefine->z_Data = new char[iLength + 1];
        itrDefine->z_Data[0] = 0;
        break;
    case DB_DATA_TYPE_USTRING:
        iLength = itrInfo->GetMaxDataLength();
        iLength *= 2;
        itrDefine->uz_Data = new UChar[iLength + 1];
        itrDefine->uz_Data[0] = 0;
        break;
    default:
        break;
}

    ++k;
    ++itrDefine;
    ++itrInfo;
}

return pQueryResults;
}

```

Figure 38 : ExecuteQuery Method for SQLite DB

```

//Initialize the DB Connection
DataConnector* pDataConnector = new DataConnector();
pDataConnector->Init();

//Execute the DB Query
int i_UniqueID = 0;
String zQuery = "Select ticker_id,source_id from tickers where ticker_serial = 1421";

ContextDataQuery* pContextDataQuery = new ContextDataQuery(i_UniqueID,zQuery);
pDataConnector->ExecuteQuery(pContextDataQuery);

//Get the result set
DBTable mDBTable = pContextDataQuery->GetTable();

//Iterate through the result set
DBTable::Iterator itr = mDBTable.Begin();
DBTable::Iterator itrEnd = mDBTable.End();

int iFieldIdExchange = -1, iFieldIdSymbol = -1;

iFieldIdExchange = mDBTable.GetColumnIndex("TICKER_ID");
iFieldIdSymbol = mDBTable.GetColumnIndex("SOURCE_ID");

while(itr != itrEnd)
{
    DBRecord& rRecord = *itr;

    String zExchange,zSymbol;
    if(iFieldIdExchange > -1)
        zExchange = rRecord[iFieldIdExchange];
    if(iFieldIdSymbol > -1)
        zSymbol = rRecord[iFieldIdSymbol];

    m_LogFile.Log(M_LOG_LEVEL_3,M_LOG_CATEGORY_DB, "Exchange %s|%s",zExchange.GetCString(),zSymbol.GetCString());
    ++itr;
}

```

Figure 39 : Example usage of Framework

4.4 Performance Analysis of Framework

Abstraction versus performance is one of the major design consideration which should be considered when developing such a data access layer. As discussed in the previous section, there is an abstraction layer, which helps developers transparently connect to the currently configured store. The information regarding the database and provider is generally specified in a configuration file. While this approach is very flexible, it can become a performance overhead if not designed appropriately. So after implementing the framework, same benchmarks carried out in **Section 3.2** are carried out again with the framework. The results for insert operations which are measured with Oracle and SQLite databases are as given in **Figure 39** and **Figure 40**. The results for the select operation is given in **Figure 41**.

During this analysis, the direct database API calling method which is given in section 3.3 and the database operations through the Data Connector API is compared. For both scenarios, how the transactions per second varies when the number of transactions are increasing is plotted. As we can see in the graphs, the TPS difference between the two scenarios are not significant. So we can conclude that adding an extra layer in between the business logic layer and the data layer does not degrade the application performance.

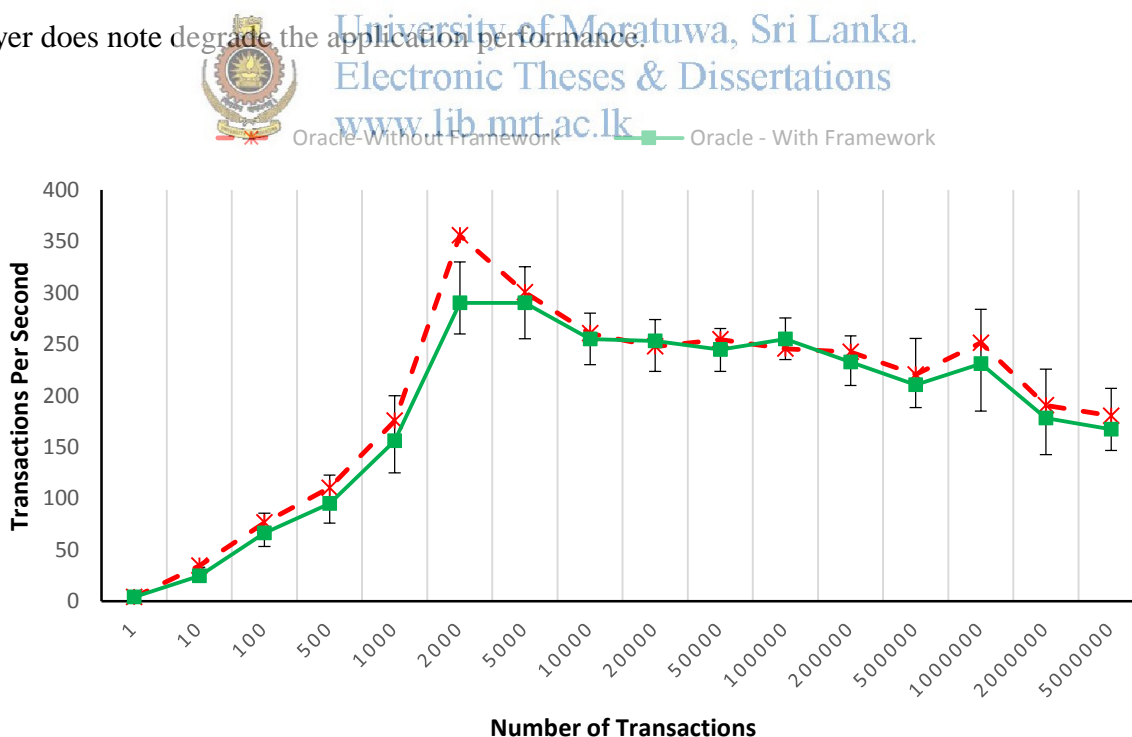
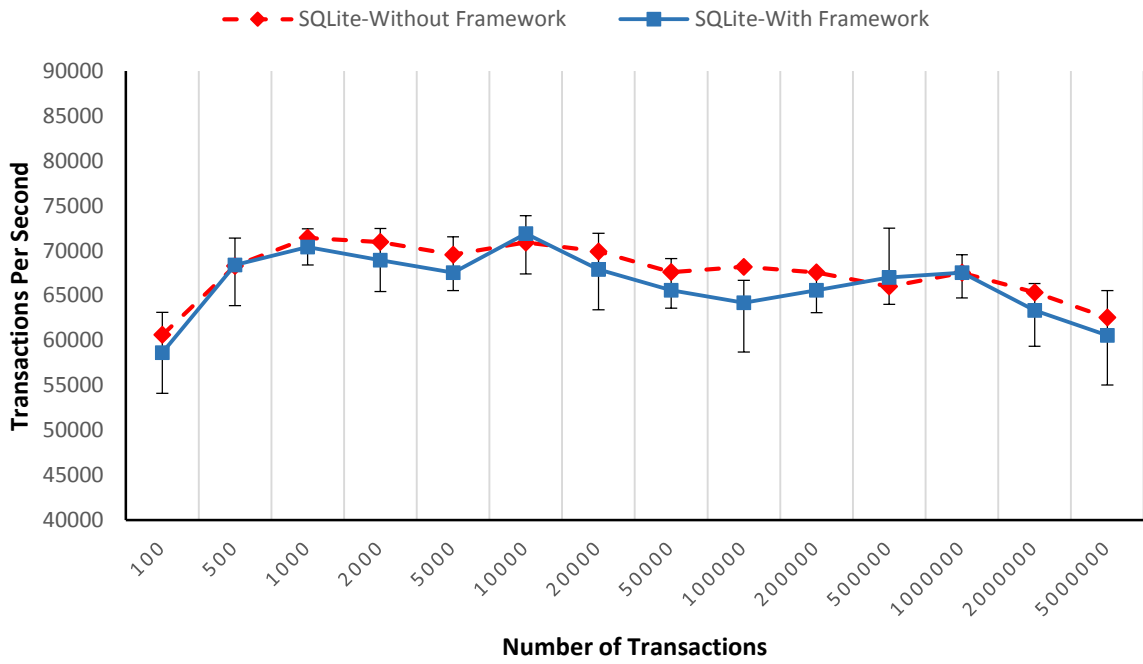


Figure 40 : Insert Operation Performance of Framework - With Oracle



University of Moratuwa, Sri Lanka.
 Electronic Theses & Dissertations
 www.lib.mrt.ac.lk

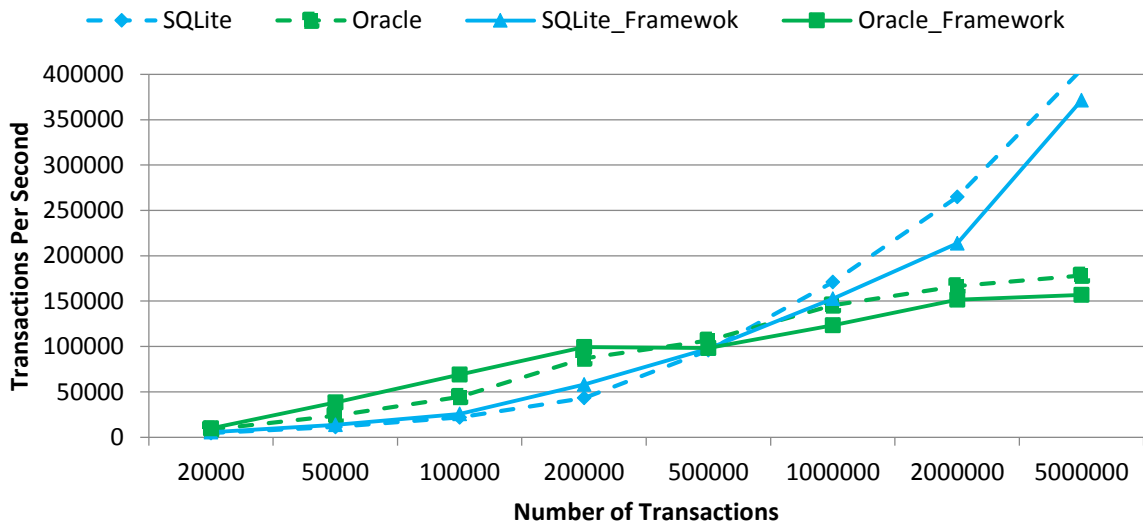


Figure 42 : Select Operation Performance of Framework

5. CONCLUSION AND FUTURE WORK

This chapter is intended to discuss the summary of benchmark results, final conclusion about the project and the remaining works of this project. While **Section 5.1** gives a brief summary of the results, **Section 4.2** gives details about future progress and remaining tasks.

5.1 Conclusion

Data growth is one of the major challenge that enterprise applications are facing today. As data accumulates, there is a corresponding burden on software developers to maintain acceptable levels of performance, whether that is measured by the speed with which an application responds, the ability to aggregate and deliver data, or the business value of information. Organizations are recognizing that their growing data stores bring massive, and largely untapped, potential to improve business intelligence. So researches on scalable data management solutions has gained more popularity within the industry now a days. During this research basically two major aspects of this problem domain is covered.

- Develop an unbiased benchmark for different In-Memory databases by comparing them with disk resident databases, In-Memory Data Caches and flat file database systems.
- Develop a framework for Data Source Management so that enterprise applications can be designed without concerning the underline data source.

To address the first problem, comprehensive performance evaluation was carried out for insert update, delete and select operations of different data sources. System throughput and the response time was taken as the performance metrics and the tests were carried out by varying the number of transactions and number of concurrent connections. For this benchmark, SQLite, MemSQL, and H2 in-memory databases, Oracle disk resident database, in-memory data cache and in-memory database are used.

According to the benchmark results obtained in Section 3.3, it is clear that In-Memory databases performs well than the disk resident databases. For insert operation IMDBs are around 200 times faster than the Oracle. For update operation IMDBs are around 80 times faster than the Oracle and for delete operation they are around 250 times faster. But for insert operation, oracle also performs comparatively well than the other operations and IMDBs are only 2 times faster than the Oracle. The disk I/O overhead and the network delay has become the biggest factors for delays in disk resident databases like Oracle.

Out of the three in-memory databases selected, MemSQL has the highest TPS for insert, select and delete operations. But for update operation, MemSQL has the lowest performance when compared to the other two databases. For insert, delete and select operations, SQLite has the next highest TPS than the other two IMDBs and H2 has the lowest. For update operation H2 database has the highest TPS. A key capability of the MemSQL platform is fast deletes. Users need to be able to delete data even faster than they can insert it so the system is not overwhelmed. When the data ingest rate is faster than the system can delete, users are forced to limit the amount of data they retain for real-time analytics. A system that can delete large volumes of data quickly can increase the amount of data that can be retained for real-time analytics.

Flat File Database has less run time for insert operation than the Oracle database as well. Non availability of transaction recovery mechanisms such as transactions logs and not having any constraint checking are the reasons for fast insert operation of file database. But for select operation flat file database become very slow when number of transactions increased. The reason for this poor performance of flat file DB is, to access the structure of the data and manipulate it, the file must be read in its entirety into the computer's memory.

For both insert and select operations, in-memory data cache is performing much better than the other databases. Cache is around 500 times faster than Oracle and 2 times faster than IMDBs for insert operations. For select operation it is around 5 times faster than Oracle and 1.5 times faster than IMDBs. Although in-memory data cache is not a good data persistent mechanism due to its volatile nature, it is good for enterprise applications which required high data processing rate such as complex event processing systems.

When number of concurrent connections are increasing, for all databases the TPS is initially increased gradually and then remains constant. But for SQLite database, TPS is gradually decreased when number of concurrent connections are high. According to the results, it can be seen that oracle database can support larger number of concurrent connections without degrading the performance.

To address the second problem of not having a standard framework to access data source layer, a data connector framework is developed in C++ language. By looking at the performance analysis results of the Framework given in Section 4.4, it can be concluded that adding an extra layer between the presentation layer and the data layer does not affect the performance of the application as there is no significant difference between the two curves.

5.2 Future Work


At this initial release of the data management framework, support for a stack of most useful data sources used in the enterprise application is provided. But sometimes these data sources will not perfectly match with the some of the enterprise applications, since there are large number of databases using in this domain. Since this framework is extensible solution, it is possible to enhance its features by providing support for more databases and data sources which are used in the applications. Hence in future, it is possible to provide a data source stack which contains almost all databases and data sources under each specific category and then it will be more flexible to developers, when managing their data sources.

Currently the data management framework is supported for Linux OS and GNU GCC C++ compiler only. So as future work we could add cross platform support for this so that it will be more usable for enterprise applications.

Cloud computing is quickly gaining popularity with companies in all industries. The cloud's on-demand elasticity, enabling it to expand its computational power as needed for peak loads, creates new and important benefits for enterprise computing. So in future we could research on how cloud based data sources can be integrated with this framework and how effective it would be for enterprise applications.

Security will be one of the major factors which impact greatly in software development. All the core business data and other organization data are stored in these data sources. So accessing and altering these data should be done in more secure manner. So the security aspects such as enable password protection for data source connections and add support for encrypted data can be integrated with this framework in future.

6. REFERENCES

- [1]. Gordon E. Moore., "Cramming More Components into Integrated Circuits ", in proceedings of the IEEE, vol. 86, No. 1, January 1998.
- [2]. Donald K. Burleson, "Oracle Tuning: The Definitive Reference", Rampant TechPress, 2nd Ed. New York: Wiley, 2010, pp. 483-485.
- [3]. IBM Inc." Applying new analytics tools to reveal new opportunities". Internet: http://www.ibm.com/smarterplanet/us/en/business_analytics/article/it_business_intelligence.html. [Accessed: 03-Jan-2014].
- [4]. F. Raja et al., "A Comparative Study of Main Memory Databases and Disk Resident Databases", in World Academy of Science, Engineering and Technology 14 , 2008
- [5]. H.O. Plattner and A. Zeier, "In-Memory Data Management: An Inflection Point for Enterprise Applications", Springer, Berlin Heidelberg, 2011.
- [6]. Manghul Tu et al., "Secure Data Objects Replication in Data Grid", in IEEE Transactions on Dependable and Secure Computing, Vol 7 No 1, Jan 2010
- [7]. Pierangelo Di Sanzo et al. "Auto-tuning of Cloud-based In-memory Transactional Data Grids via Machine Learning" in IEEE Second Symposium on Network Cloud Computing and Applications, 2012  University of Moratuwa, Sri Lanka. Electronic Theses & Dissertations www.lib.mrt.ac.lk
- [8]. InfoQ Articles."Jags Ramnarayan on In-Memory Data Grids". Internet: <http://www.infoq.com/articles/in-memory-data-grids>. [Accessed: 22-Jan-2014].
- [9]. Hector Gracia and Kenneth Salem., "Main Memory Database Systems: An overview", in IEEE Transactions on Knowledge and Data Engineering, Vol 4 No 6, Dec 1992
- [10]. Oracle TimesTen, "Oracle Times Ten In-Memory Database Architectural Review", Oracle Press, USA, 2006, pp. 10-11.
- [11]. Cha S.K. et al., "An extensible architecture for main-memory real-time storage systems", in IEEE Third International Workshop on Real-Time Computing Systems and Applications, 1996
- [12]. H.Garcia-Molina and K. Salem, "High performance transaction processing with memory resident data," in Proc. Int. Workshop OII High Performance Transaction Systems, Paris, Dec. 1987
- [13]. R.B. Hagmann, "A crash recovery scheme for a memory-resident database system," IEEE Transactions and Computing.. Vol. C-35, pp. 830-842. Sept. 1986.

- [14]. D. J. DeWitt et al., "Implementation techniques for main memory database systems", in Proceedings of ACM SIGMOD Conference, June. 1084.
- [15]. Krueger J. et al., "Data structures for mixed workloads in in-memory databases", in IEEE 5th International Conference on Computer Sciences and Convergence Information Technology (ICCIT), 2010
- [16]. T.J. Lehman and M. J. Carey, "Query processing in main memory database management systems," in Proc. ACM SIGMOD Conference, Washington, DC, May, 1986.
- [17]. M. H. Eich, "A classification and comparison of main memory database recovery techniques," in Proceedings of International Conference on Data Engineering, Feb. 1987, pp. 332-339.
- [18]. S. K. Cha et al., "Object-oriented design of main-memory DBMS for real-time applications," in Proceedings of 2nd International Workshop on Real-Time Computing Systems and Applications, Oct. 1995.
- [19]. H.Garcia Molina and K. Salem, "High performance transaction processing with memory resident data," in Proceedings of International Workshop on High Performance Transaction Systems, Paris, Dec.1987.
- [20]. M.Stonebraker, "Managing persistent objects in a multi-level store," in Proceedings of ACM SIGMOD Conference, Denver, CO, May 1991, pp.2-11.
- [21]. Elliot King., "The Growth And Expanding Application Of In-Memory Databases", for Information Value Loyola University Maryland, June 2011
- [22]. Liu Yang et al., "The Research of Embedded Linux and SQLite Database Application in the Intelligent Monitoring System", in IEEE International Conference on Intelligent Computation Technology and Automation (ICICTA), Vol 3, 2010
- [23]. Olson, M.A., "Selecting and implementing an embedded database system", in IEEE Computer Society, Volume 33 Issue 9, Sept 2000
- [24]. Jens Krueger et al., "Main Memory Databases for Enterprise Applications", in IEEE 18Th International Conference, Vol. 1 No 6, Sept 2011.
- [25]. Hasso Plattner and Alexander Zeier, "Introduction to IMDB," in In-Memory Data Management - An Inflection Point for Enterprise Applications, 2nd Ed. New York: Springer, 2011, pp. 3-5.
- [26]. David J. DeWitt, "The Wisconsin Benchmark: Past, Present, and Future," in The Benchmark Handbook, 2nd Ed. Morgan Kaufmann Publishers Inc, 1993.
- [27]. CSQL, "CSQL Wisconsin Benchmark Results" [Online]. Available: <http://csql.sourceforge.net/bresults.html> [Accessed On: 2014 February 14]

- [28]. “Oracle TimesTen In-Memory Database on Oracle Exalogic Elastic Cloud” , white paper, Oracle Corp., July. 2011.
- [29]. “Telecommunication Application Transaction Processing (TATP) Benchmark Description”, white paper, IBM Software Group Information Management., March. 2009.
- [30]. “Using the TATP Benchmark to Measure the Effect of Additional Memory Capacity on Database Performance”, white paper, IBM System x and Database Performance Analysis, June, 2011.
- [31]. Francois Raab, " TPC-C -- The Standard Benchmark for Online transaction Processing," in The Benchmark Handbook, 2nd Ed. Morgan Kaufmann Publishers Inc, 1993.
- [32]. Yao, S. Bing; Hevner, Alan R., "A Guide to Performance Evaluation of Database Systems," in The NBS Special Publication 500-188, 1984.
- [33]. SQLite.org, "SQLite Database ” [Online]. Available: <https://www.sqlite.org/> [Accessed On: 2015 February]
- [34]. H2database.org,"H2 Database ” [Online]. Available: <http://www.h2database.com/html/main.html> [Accessed On: 2015 February]
- [35]. “Comparison of Hibernate with H2 server vs Hibernate with SQLite embedded”, in JPA Performance Benchmark, 2010.
- [36]. Memsql.org, “MemSQL Documentation” [Online]. Available: <http://developers.memsql.com/docs/latest/> [Accessed On: 2015 February]
- [37]. Oracle Cooperation, “Oracle Database” [Online]. Available: <http://www.oracle.com/us/corporate/index.html> [Accessed On : 2015 March]
- [38]. Bitton, D., DeWitt, D. J., and C. Turbyfil, "Benchmarking Database Systems: A Systematic Approach," Computer Sciences Department Technical Report #526, Computer Sciences Department, University of Wisconsin, December 1983.
- [39]. Pierangelo Masahiko Tanaka et al., "Database Operation Using ODBC/JDBC in the KEK 8gev LINAC", in International Conference on Accelerator and Large Experimental Physics Control Systems, Italy, 1999