GOOGLE DATASTORE ON MULTI-CORE ARCHITECTURES

Project Background Report
COMP60990 – Research Methods and Professional Skills
2011

MOON MOON NATH

School of Computer Science
University of Manchester
# Table of Contents

List of Figures ................................................................. iii  
List of Tables ................................................................. iii  

Abstract iv  

1 Introduction 1  
  1.1 Shared Memory Multi-Core Systems and Google DataStore ............. 2  
  1.2 Aims and Objectives .................................................... 3  
  1.3 Organization of the Report ............................................ 3  

2 Background 5  
  2.1 The BigTable ............................................................ 6  
  2.2 HBase ........................................................................ 7  
  2.3 The Google File System .................................................. 9  
  2.4 Hadoop Distributed File System ........................................ 11  
  2.5 Data Retrieval in a Cluster Environment: MapReduce .................. 11  
  2.6 In-Memory Database Systems .......................................... 12  
  2.7 Cache-Oblivious Data Structures ....................................... 13  

3 Research Methodology 15  
  3.1 System Overview ......................................................... 15  
  3.2 System Model .............................................................. 15  
  3.3 Data Manipulation and Retrieval ....................................... 17  
  3.4 Evaluation .................................................................... 17  
  3.5 Project Execution Plan .................................................. 18  
  3.6 Project Delivery ............................................................ 18  

4 Risks and Challenges 19  

5 Conclusion 20  

References ................................................................. 21  
Appendix 1 ................................................................. 23  
Appendix 2 ................................................................. 24
List of Figures

Figure 1: NUMA architecture ............................................................... 1
Figure 2: UMA architecture ............................................................... 1
Figure 3: Example table storing web pages ........................................... 6
Figure 4: To illustrate the concept of ‘rows’, ‘column families’ and ‘columns’ in BigTable ............................................................... 8
Figure 5: To illustrate timestamps ......................................................... 8
Figure 6: GFS architecture ................................................................. 10
Figure 7: The memory hierarchy ......................................................... 13
Figure 8: The RAM model ................................................................. 13
Figure 9: Proposed cache-oblivious data structure ............................... 16

List of Tables

Table 1: Production system configurations for performance evaluation .......... 18
Abstract

Google provides its users an assortment of applications and services. In the process, it requires to store and manage huge volumes of user data. To accomplish this, all the data is distributed and stored across thousands of servers, in a distributed storage system. This approach is beneficial since it exploits parallelism in a cluster environment to achieve an optimal system performance, in terms of throughput and response time.

The advent of multi-core architectures has resulted in a lot of research to find effective software solutions that will take complete advantage of the parallel hardware. This project also deals with investigating the possibilities and developing a Google datastore-like system for shared memory multi-core machines, that is scalable, fast, and efficient.

This background report discusses the motivation, relevant literature, scope, design decisions, implementation techniques, evaluation strategies, and the deliverables of the project.

The literature survey provides all the essential background knowledge, necessary to understand the idea behind this research. The proposed system design comprises primarily of an underlying data structure and a set of APIs (Application Programming Interface) to manipulate the database system. The design decisions are aimed at obtaining high system throughput in a multi-core environment.

The implementation mechanisms proposed, based on Java 6 include the parallelization of searches made to the database, apart from developing the data structure. An apposite data structure design is critical to achieve the desired performance and scalability criteria. Support for several database operations like insert, delete, append and search query, similar to that of Google DataStore exist in this system. The query execution is parallelized on several multi-core machines to capture and evaluate performance and scalability of the design, based on certain performance evaluation metrics. This is followed by an analysis of this evaluation, to arrive at the final conclusion.

The designed database is a subset of the Google system (datastore) and hence, supports only the core features of the datastore. Given, the stringent time frame, enhancements like fault tolerance and security is avoided.
1 Introduction

Processor architecture has evolved considerably over the years. From being steered primarily by Moore’s Law [37] to exploiting multi-core parallelism nowadays, it has travelled a long way. The direct correlation between processor frequency and its performance is threatening to vanish owing to certain limiting factors. The most prominent one among them is the transistor size, which cannot be reduced beyond a certain degree [35]. In fact, smaller size transistors also require a lot of complex design effort. This is a physical limitation on the reduction of a transistor size. However, the direct impact of increasing the number of transistors in a chip is the increased power consumption. Apart from this, there is also the problem of physical memory bandwidth. We know that the speed of the main memory is much slower than that of the processor. In fact, the rate at which the processor frequency has amplified in the past decade, the memory speed has not. This memory bottleneck will always restrict the system performance, despite the high clock speed of the processor. This is due to the fact that a fast processor with a slow memory only increases the processor idle time. Wulf et al. called this bottleneck the ‘Memory Wall’ [40].

Hardware designers have now incorporated multi-core technology into the processors. Processors instead of having a single CPU have multiple CPUs built onto the same chip, called ‘cores’. The existence of multiple cores creates an opportunity for improvements in performance and speed of the processor, provided there exists parallel software that can utilize the cores available to it. This is because a task can now be executed on several cores simultaneously as threads.

A modern multi-core processor is usually a NUMA (Non Uniform Memory access) shared memory multi-processor [42]. However, the ones with fewer cores are still SMPs (Symmetric Multi-Processor) having UMA (Uniform Memory Access) [42]. UMA is where a single memory is shared by all processor cores, such that each core takes the same amount of time to access that memory. However, in case of a NUMA, pools of memory exists that are shared by a set of cores (multiple cores grouped together to form a ‘socket’). This implies that each socket (group of cores) is connected directly to one RAM and indirectly to all the others (see figure 1), which results in some sockets accessing a particular RAM faster than the others. The following figures 1 and 2 illustrate the two multi-core architectures.

![NUMA Architecture](image1)

![UMA Architecture](image2)

Here, we can see that a single operation can get divided among 16 and 4 CPUs in the NUMA and UMA architectures respectively and if parallel software is available can utilize the existence of these multiple cores for performance enhancements. However, some latency exists in case of NUMA due the different access times.
Although the hardware industry has found an effective technique in the form of multi-core, the software industry still needs to evolve accordingly to exploit this hardware. It is extremely vital to write software and design frameworks that can efficiently scale and utilize the underlying multi-core architecture. Also, one should bear in mind the hierarchical memory structure involving CPU caches to yield optimal system performance. This is especially true when processing terabytes of data.

There are some frameworks for parallel programming on multi-core like OpenMP [47] for C and Fortan, Java Fork-Join [49], Phoenix [13] and MR-J [17]. However, there still exists a lot of instability in the applications written for multi-core architectures. The complexity involved in the appropriate utilization of thread-level parallelism is magnified by the existence of multiple cache levels, cache-sharing, memory page sizes and so forth [44]. Therefore, software designers can achieve greater performance from the multi-core systems if they consider these factors and design structures and algorithms that are tailored accordingly.

1.1 Shared Memory Multi-Core Systems and Google DataStore

The industry today requires managing huge amounts of data, in the order of petabytes. To process such large computations, a distributed cluster computing environment or a shared memory multi-core architecture can be used. Again, apart from improving the hardware, the software should also be rewritten to be able to exploit the hardware, as mentioned earlier.

Google has devised a mechanism based on the distributed computing environment to process and manage petabytes of user data. BigTable [3] is a high performance, scalable proprietary database system from Google. It is a distributed storage system that supervises large amounts of data across several thousand commodity servers. It is built atop other Google services like the Google File System (GFS) [5], MapReduce [14], Chubby Locking Service [6], and so on.

The GFS is a distributed file system that runs on several thousand inexpensive commodity Linux servers. It provides the usual file system operations with special fault tolerance, scalability and reliability features. The database operations are designed such that they can utilise the distributed nature of the environment and run in parallel. However, it does not utilize the individual cores within a single system to gain performance benefits; in other words, it does not support execution on a multi-core architecture. The distributed parallelism is achieved by using MapReduce, which is a framework that requires a programmer to write only two special functions, while the complex parallel activities are handled by its underlying run-time features.

Many Google projects make use of BigTable like, Google Earth, Google Finance, the web indexing operation, Gmail, YouTube, and so on.

Several similar distributed systems exist as open source. The most commonly used are from Hadoop [12]. Hadoop’s HBase [7], HDFS [12] and MapReduce [16] are similar in most ways to Google’s BigTable, GFS and MapReduce respectively. Hadoop is extensively used by services like Facebook, Twitter, Adobe, EBay, LinkedIn and Yahoo to name a few.

The applications supported by these distributed systems give us a fair idea about the enormity of the data handled by them. These systems are robust and have a low response time in most situations. However, if concurrent activities increase manifold, owing to a large number of simultaneous users or if the amount of data increases by many times, over the next few years, the performance parameters might not produce as good a result as they do now. The computations are bound to become large requiring more power in the future. Therefore, with the advent of multi-core architectures, it is only natural to try and extract the additional computational power required, from the multi-core systems itself. In fact, BigTable and
HBase make use of inexpensive commodity machines for their clusters. The multi-core nature of these individual systems, that constitute the cluster, can actually be exploited to gain improvements in performance and speed-up.

1.2 Aims and Objectives

This project aims at investigating the possibility of implementing a subset of the BigTable [3] functionality on multi-core architectures. The database to be designed will be cache-oblivious [27] and will always reside in memory [18, 19], eliminating entirely the access to a secondary storage for its operations. The research will be carried out in three basic phases. The initial phase will involve conducting a survey of the Google database system, its underlying infrastructure, the GFS [5] and a research of other similar non-SQL (unconventional or non-relational DBMS) database technologies. It will also constitute looking at the various in-memory and cache-oblivious data structures to identify the suitability of these structures for this research. The existing multi-core frameworks will also be examined to identify a suitable implementation that will be used to achieve parallel activity. Also, in this phase, we will explore and then decide on the programming environment for developing this implementation.

The next phase is to design and develop a version of the DataStore system for shared-memory multi-core machines, based on the decisions taken in the previous phase. This will involve designing and implementing a suitable data structure that has the capability to perform operations similar to Google BigTable. This structure will be used to perform simple operations like create, populate data, append new data and delete on the database. In addition to this, thread security features will be incorporated to allow multiple users concurrent access to a single database. Next, a data retrieval operation will be performed on it, exploiting the parallelism of the processor cores.

The final objective of this research is to evaluate the performance and usability of this multi-core implementation. Also, the efficiency of the implementation as well as its scalability on various multi-core systems will be examined. The parallelized query will be used for this evaluation. The results obtained from various multi-core systems will be analyzed to arrive at a formal conclusion. Moreover, if time permits, the performance of this database will be compared against the performance of Google DataStore to obtain a comparative analysis of the two systems to support research results.

1.3 Organization of the Report

This background report is organized into a Background (Section 2), Research Methodology (Section 3), Risks and Challenges (Section 4) and Conclusion (Section 5) sections, apart from the Introduction (Section 1). The Background section contains an overview of the entire research activity. It presents the primary motivation behind this project – Google DataStore (BigTable) [3]. The concept, architecture and salient features are discussed briefly. It is then compared with its open source counterpart HBase, from Hadoop [7]. Next GFS [5] and its open source version from Hadoop, HDFS [12] are discussed, exploring the architecture of these systems. The database querying mechanism, MapReduce [14, 16], used by these distributed systems is then examined. Further, we look at the alternative database technologies like In-Memory Databases (IMDB) [18, 19] to investigate
the feasibility of using them for this implementation. We also look at cache-oblivious [20] data structures to explore their suitability and at the same time, identify an appropriate structure for development.

The **Research Methodology** section contains a brief description of the design decisions made based upon the survey conducted. The data structure to be used, including its architecture is discussed briefly. The kind of operations supported by the database and other related characteristics are next touched upon. It also contains information about the programming environment and the infrastructure to be used to execute queries in parallel (as threads) on the multi-core architecture. This section further includes the evaluation techniques, the various multi-core configurations to be used, the benchmarks, the development strategy and plan and finally the deliverables. The risks and challenges are highlighted in the next section, followed by a summary of the report.
2 Background

Google provides its users with a Platform-as-a-Service (PaaS) commercial cloud technology, in the form of the Google App Engine [1]. App Engine allows users to build, maintain and run their web applications on Google’s infrastructure by means of a huge storage structure called the DataStore [2]. It comprises of several APIs, required for its services, one of which is the Datastore API [2], which is available in both Java and Python versions and accesses a query engine and some atomic transactions. This API provides users with a stable storage that is both reliable and consistent.

The huge amount of user data present in the DataStore is in reality, stored across thousands of servers and managed by a distributed storage system called BigTable [3]. In other words, the DataStore of App Engine is built on top of BigTable. BigTable, earlier a single-master, distributed storage system, consists of three main components – a library linked to all clients, a master server and several tablet servers [3]. It is a non-SQL (non traditional DBMS) database management system in that it does not conform to a specific schema – the number of columns in different rows of the same table can vary, thus sharing characteristics of both row-oriented and column-oriented databases. Typically, a column-oriented database serializes (stores values internally in file etc.) the contents of the database in a column-wise fashion, in that all data from one column gets stored together and then the same for the next column and so forth. The biggest advantage of such a storage mechanism is the quick execution of aggregation operations (like sum, count, etc. that are performed over specific columns) since now entire rows need not be read. Instead the required column, a much smaller subset of the database, can be accessed directly, giving faster query results. Also, since column data is usually of the same type, compression techniques can be employed to achieve storage size optimizations, which is not possible in row-oriented stores.

BigTable uses an underlying Google File System (GFS) [5] to store data and is based on the shared-nothing architecture [4]. BigTable also relies on a distributed locking service called Chubby [6] to ensure consistency and synchronization of all client activities in a loosely-coupled distributed system. It provides its clients with a highly reliable and consistent environment.

The open source counterparts from Hadoop [12] also have similarities in terms of architecture. One of the primary objectives of this research work is to conduct a survey of these distributed systems to understand their functionality, architecture and the structures employed. Additionally, we will examine in detail different types of data structures especially the ones that utilize the cache (cache-aware and cache-oblivious) for performance improvements. Their study will provide us with necessary understanding and thus allow us to decide on the data structure to implement. This decision will be guided primarily by the fact that the structure should be similar to that of BigTable’s; data should be stored in a column-oriented manner. Efficient memory and cache utilization, performance etc. are the other criteria. We will also look at in-memory data bases [18, 19] as they have very low response times and decide on their suitability for this project.

This section will deal with the above discussed aspects of my research and thus provide an understanding of the background and the system in general.
2.1 The BigTable

BigTable is defined as “a sparse, distributed, persistent multidimensional sorted map” by Ghemawat *et al.* [3]. It is “sparse” because each row in a table can have an arbitrary number of columns, very different from the other rows in that table. This is possible due to the fact that BigTable is not a conventional relational database management system that is strictly row-oriented. It is instead a non-SQL, column-oriented database system. A BigTable row contains only those columns which contain some value. Contrary to an RDBMS, there are no NULL values and no JOINs. The tables are also unlike the traditional RDBMS ones. A table here is a “map”, indexed by a row key, column key and a timestamp. In other words, a cell in a BigTable table is identified by 3 dimensions – row, column and timestamp. The timestamp facilitates versioning of the data. Each cell can have multiple values at different points in time and each value, an array of bytes, is maintained separately with its associated timestamp. These are 64 bit integers and can be used to store actual time in microseconds as well.

The unique row key is a maximum of 64 KB in size and is an arbitrary string. All data is maintained in lexicographic order of the row key. A table can be huge and is therefore split at row boundaries to manage them. These partitioned tables are called tablets. Each tablet is around 100 – 200 MB in size, allowing several hundred to be stored on each machine. This sort of an arrangement allows for fine grained load balancing.

Several column keys are combined to form a set called a *column family*, which is the basic access control unit. Any number of column keys can be part of a single *column family*, but these columns are usually of the same data type. The number of *column families* is restricted to a few hundred in contrast to the unbounded number of columns in a table. A column key has the following syntax: `-family:qualifier`, where ‘family’ and ‘qualifier’ refer to *column family* and *column key* respectively.

The following diagram, redrawn from the original paper [3] illustrates the structure of a single row in BigTable.

![Diagram of a BigTable row](image)

**Figure 3**: Example table storing web pages. It is redrawn from the original BigTable paper [3]. The diagram contains 2 *column families* namely ‘contents’ and ‘anchor’. ‘contents’ has a single column while ‘anchor’ has 2. Again while ‘anchor’ column values have a single timestamp (t8 and t9), ‘contents’ has 3 timestamps for a single value (t3, t5 and t6), where t3 is the oldest and t6 is the most recent value. The next row can have a different number of columns for these two *column families*.

As already mentioned, Google uses the distributed GFS to store all data and maintain log records. Internally however, an immutable file format called *SSTable* is used to store the BigTable tablets (data). It is sequence of blocks typically 64 KB in size. An index is stored at the end of each *SSTable*, to locate its blocks.

A BigTable realization comprises of three major constituents:- master server, several tablet servers and a library attached to every client machine. The master server assigns tablets
to various tablet servers, performs load balancing and garbage collection as well as detects any alterations in the tablet servers. Tablet servers manage the tablets assigned to it including reads and writes by the client. Additionally it is also responsible for partitioning tablets that have exceeded their size limit.

BigTable uses Chubby [6] as a locking service for synchronization, tablet location information, tablet server expirations, store schema information and so forth. A three-tier B+ tree-like structure is used to store tablet location information. Here, the first level is a file stored in Chubby which holds the location of the root tablet, which in turn holds locations of all other tables in separate tablets called METADATA. Each METADATA tablet stores the location of the user tablets that includes a list of SSTables. SSTables are loaded into memory using their index into a memtable. All updates are also made to a memtable. As its size increases, due to updates, to reach a threshold, a new memtable is created. The old memtable is turned into an SSTable and sent to the GFS. This is termed as “minor compaction”. Minor compactions create new SSTables which results in several of them after a time. Therefore, to curb the creation of numerous such SSTables, another merge operation is performed at regular intervals, called “major compaction”. This involves rewriting all existing SSTables into a single one.

### 2.2 HBase

HBase is an open source BigTable-like structured storage built on the Hadoop Distributed File System (HDFS) [12]. Source [7] defines HBase as “an open-source, distributed, versioned, column-oriented store modelled after Google’s BigTable”. Here too, a table is “sparse” in that rows in the same table can have variable number of columns. The rows again are lexicographically sorted on a unique row-key. It is a multi-dimensional map like BigTable, with the data being identified by the 3 dimensions namely, row, column and timestamp. A row contains only those columns which hold some data value; no NULL values are used. Columns like in BigTable are grouped together to constitute column families and are denoted by a column qualifier or label. A column needs to be identified therefore, by the <family:qualifier> notation. Figure 4 below illustrates rows and columns. It is a JSON example created based on examples from source [8].
This figure clearly explains the arrangement of rows, column families and columns in HBase (and BigTable). Here, ‘aaaaa’ and ‘aaaab’ are the two rows in an HBase table arranged in ascending lexicographic order. The table contains 2 column families: ‘A’ and ‘B’. Note that column families in a table are usually static unlike the columns constituting them. Therefore, the first row ‘aaaaa’ has 2 columns from only 1 family, A:foo and A:bar, whereas the second row ‘aaaab’ has 2 very different columns belonging to 2 different families, A:check and B:test. Each of these data values can also have several versions as stated earlier, thus allowing the database to store historical data as well. This can be illustrated using JSON as shown below.

```
"aaaaa" : {
    "A" : {
        "foo" : { 20 : "y", 8 : "x" },
        "bar" : "d"
    },

    "aaaab" : {
        "A" : { "check" : "world" },
        "B" : { "test" : "ocean"
    }
}
```

The figure above illustrates the use of timestamps in HBase/BigTable. The most recent data is stored first. For instance, to access the data ‘y’ (most recent value) HBase will use the path “aaaaa/A:foo/20” while “aaaaa/A:foo/8” for ‘x’. Also, when responding to a query HBase accesses the timestamp that is “less than or equal to” the queried time. For instance, if we want to access all values with timestamp 10, we will receive the cell value ‘x’ since its timestamp is less than 10.

An HBase table comprises of several regions, each of which is marked by a ‘startkey’ and ‘endKey’. Regions are made of several HDFS blocks. There are two types of nodes namely, Master server, Region servers attached to numerous client machines. These servers
are similar to master and tablet servers in BigTable. Master server monitors the region servers as well as assign and balance load on them. Region servers hold multiple regions. Contrary to a Chubby lock service in BigTable, HBase uses a ZooKeeper [9], a centralized service, for distributed synchronization. It has an extremely simple interface that is itself distributed and highly reliable. The clients connect to a specific cluster by seeking information from the ZooKeeper since it holds the locations of all Region servers hosting the root locations of all tables.

HBase uses an internal file format called HFile [11], analogous to BigTable’s SSTable. It uses a 64 KB block size, containing data and identified by a block magic number. HBase like BigTable is extremely efficient when managing huge amounts of data in the order of petabytes over equally large number machines widely distributed all across the globe. It allows data replication for reliability, availability and fault tolerance. It also facilitates distributed reads and writes on the data that are very fast.

2.3 The Google File System

The Google File System (GFS) [5] is a proprietary, scalable and distributed file system designed specifically for large, distributed and data-intensive applications. It is fault-tolerant and reliable, providing a high aggregate performance to its clients. The GFS design is primarily motivated by the observations on the technological environment as well as, the application workloads, where component failures are inevitable. The file system runs on thousands of inexpensive, commodity Linux systems and is accessed by an equivalent number of client machines. Unlike many file systems, it is not built into the OS kernel, but supported as a library.

GFS is simple and provides the users with the basic file commands like open, close, create, read, write, append and snapshot. Append and snapshot are special commands; while append allows multiple clients to add information to files (even concurrently) without overwriting existing data, snapshot creates a copy of a file/directory tree at minimal system cost.

Google organizes its resources into distributed clusters of computers, with each cluster comprising of thousands of machines, classified as either master server, chunk servers and client servers. Client files tend to be very large (order of multi GB), so they are divided into fixed sized chunks of 64 MB each and stored on various chunk servers. For reliability, chunks are replicated on multiple chunk servers with a default of 3 replicas. At the time of creation, each chunk is assigned a globally unique 64 bit chunk handle. The master acts as cluster-coordinator, maintaining an operation log for its cluster, stores all file system metadata including namespaces, access control information, mappings of files to chunks and current chunk locations.

The master server does not persistently store any chunk location information instead, upon start-up, it polls the chunk servers, which respond with the contents present in it. Also, periodically it communicates with the chunk servers via HeartBeat messages to give instructions and collect their state.
The client code is linked into each application (Figure 6 above) and it communicates with the master and chunk servers to read/write data. Figure 6 illustrates the architecture in terms of a single read reference. The application sends a filename and byte offset to the client, which converts this information into a chunk index and sends it (along with filename) to the master. The master replies with the corresponding chunk handle and replica locations. The client then sends a request to the closest replica. Also, it caches the chunk replica locations so that future interactions need not involve the master.

All metadata on the master are stored as in-memory data structures and hence master operations are fast and efficient. The operation log mentioned earlier is critical to the GFS in that, it contains all vital changes to the system metadata. The system is designed to have minimal master involvement in all operations. To this end, the master assigns lease to any one of the replicas and calls it the primary replica (chunk server) for an initial duration of 60 seconds. All mutations (alterations to file content and/or namespace) are now managed by the primary, including secondary replica management.

Another crucial feature of the GFS is garbage collection. This mechanism is unique in that, the physical storage released (due to a file deletion) is not reclaimed immediately. Instead, the file is renamed with a special (hidden) name along with a timestamp. The master performs scans at scheduled times, during which it deletes permanently all ‘hidden’ files that have existed for more than 3 days (using timestamp).

GFS uses a very important principle of autonomic computing [45, 46], which means that a system can detect and correct its problems without any human intervention. It incorporates ‘stale replica detection’ (where using the replica timestamp, master can identify outdated replicas), various fault-tolerance techniques like fast recovery (all servers to restart and restore stable state in seconds irrespective of how it terminated), chunk replication, master replication (copies of master maintained, including ‘shadow masters’ – slightly outdated read-only master replicas) and so forth.

The Google File System is structured in a manner that systems as well as hardware memory can be upgraded with a lot of ease, making it truly scalable. This knowledge is vital since it enlightens us with the knowledge of in-memory data structures, fault tolerance and security techniques.
2.4 Hadoop Distributed File System

HDFS [12] is similar to the GFS [5] and partitions the large data files into fixed sized blocks called chunks and stores them across several machines in the cluster. It is designed to handle hardware failures and network congestion in a robust manner. It uses a large number of inexpensive commodity systems to construct the distributed cluster. It is fault tolerant, reliable and highly scalable. However, the design is restricted to a specific type of applications. It is assumed that the applications using HDFS are written only once and perform frequent sequential streaming reads with infrequent updates.

An HDFS cluster comprises of a NameNode, connected to numerous DataNodes and client machines. This is analogous to the Master and Cluster servers in GFS. The NameNode stores all the metadata information like namespace, file to chunk mappings etc. and also controls the DataNodes. All metadata is stored in-memory to facilitate faster access. The NameNode is accessed by a client to retrieve the location information of all chunks constituting the file required by it. This also includes the locations of all chunk replicas, created for greater reliability and fault-tolerance. The client then selects a DataNode nearest to it to start its operations.

DataNodes like the GFS cluster servers store the actual data chunks (blocks), with each chunk being replicated thrice by default. Also, replicas are housed on different machines, preferably on separate racks in the cluster. Moreover, apart from data replication, the NameNode is also copied so as to save the metadata in the event of a failure.

2.5 Data Retrieval in a Cluster Environment: MapReduce

Querying and data retrieval are an integral part of any database system and involves complex processing. As the amount of data increases, so does its processing complexity, in order to maintain a reasonably good response time. Distributed database systems have the advantage of utilizing parallelism to achieve this. The programming model available to exploit parallelism in both distributed and multi-core systems is MapReduce. The advantage of this model is that it abstracts away the complex parallel implementation from the programmer and yet achieves large scale parallelism. The programmer typically is involved in expressing the problem at hand as a functional programming model. Once this is done, the MapReduce runtime environment automatically parallelises it.

Google MapReduce [14] is a generic programming framework for processing and generating very large datasets in a cluster computing environment. The primary advantage of this paradigm is the simplicity it provides to a programmer; abstracting the underlying complexities and allowing the programmer to express the computation in a functional style. This implementation is highly scalable and easy to use, capable of processing terabytes of data across thousands of machines.

It requires programmers to specify two functions: map and reduce. Both functions accept key/value pairs as input. The map function processes the input key/value pair and generates an output consisting of a list of intermediate key/value pairs. The reduce function reads the sorted output of map and merges all intermediate values for a particular key to produce output for each unique key. Apart from these user-defined functions, it also has a runtime environment that manages data partitioning, scheduling, fault tolerance and automatic parallelism; all abstracted from the programmer. It uses GFS [5] as the underlying file system.
The open source counterpart of Google MapReduce is Hadoop MapReduce [16], also a framework for processing huge amount of data on large clusters of machines. This is based on the HDFS [12].

MapReduce implementation on multi-core systems is slightly different from that of distributed systems, although the underlying principle remains the same. A model called Phoenix, developed by Ranger et al. [13] and another, MR-J developed by Kovoor et al. [17] are examples of MapReduce architectures for shared memory multi-core systems.

2.6 In-Memory Database Systems

In-Memory Database (IMDB) [18, 19] systems also known as Main Memory Databases (MMDB) is a database management system that stores and manipulates its data in the main memory, eliminating disk access, unlike most database systems that use the disk for persistent storage.

The conventional disk-resident database (DRDB) systems support all the ACID (Atomicity, Consistency, Isolation and Durability) properties. Database transactions (operations) can fail due various hardware and software problems. The ACID properties ensure that these transactions are processed reliably, that is, even in the event of a failure the data stored will be consistent and reliable [38]. However, the DRDB systems have limitations in terms of their response time and throughput. Caching the disk data into memory for faster access does not completely eliminate disk accesses. Such accesses reduce the throughput while increasing the response time, thus rendering the system unsuitable for time-critical (hard real-time) applications.

On the other hand, IMDB systems were primarily designed to cater to time-critical applications by achieving very low response time and high throughput. They are faster because their performance is not dependent on disk I/Os. The data structures employed are also optimized to gain maximum performance benefits. Moreover, they usually have a strict memory-based architecture, which implies that data is stored and manipulated from memory in exactly the same form, in which it is used by the application. This completely eliminates all overheads associated with data translations as well as caching. This also results in minimal CPU usage. Another advantage of IMDB systems is that it can achieve multi-user concurrency on some shared data with consistent performance.

The main memory is volatile. This makes the IMDBs appear to lack the durability property of ACID, in case of a power off or server failure. This can be achieved by either of the following mechanisms:-

1. Creating checkpoints or snapshots. These periodically record the database state and provide the required persistence. However, in the event of a system failure, the most recent modifications will be lost (after the checkpoint), hence provides only partial durability.
2. Combining checkpoints and Transaction Logging. A transaction log records all modifications to a log/journal file that aids in complete system recovery.
3. Using a non-volatile RAM or an EEPROM (Electrically Erasable Programmable Read Only Memory).
4. Maintaining a Disk backup.

Another disadvantage appears to be the limited storage available to these systems due to the fact that all data is stored only in the main memory, which has less storage compared to a disk. IMDBs are primarily used for performance-critical embedded systems, which are usually devices that require applications and data to have a small footprint (size/memory requirement) and hence their being memory-resident (i.e. limited storage) no longer remains an issue.
Moreover, when used for systems handling large datasets, the virtual memory usually comes into play to hold the excess data. They are extremely important for this research because of their low response time and high throughput. Designing a system with lowest possible response time and optimal memory usage is one of the basic objectives of this project.

2.7 Cache-Oblivious Data Structures

Modern computers have a multi-level hierarchical storage that includes CPU registers, different levels of cache, main memory and disk, where the data oscillates between the processor registers and the rest. Figure 7 below illustrates this hierarchy.

As the memory levels move further from the CPU, their access times as well as storage capacity increases. In fact, there exists a sharp rise in these characteristics as we proceed from the main memory to the disk. This implies that for any algorithm executing on such a system, the cost of memory access (and hence system performance) entirely depends upon the storage level where the element being accessed is currently residing. Moreover, data travels between the storage hierarchies in blocks, of a certain size and different caches have different block size. So, the design of the algorithm, in terms of how it accesses memory, now has a major impact on its actual execution time and therefore, to achieve optimal performance, it should take into consideration the above mentioned storage hierarchy characteristics, especially the cache. Normally algorithms are analyzed by overlooking the existence of cache in between CPU and RAM (illustrated in Figure 8) which assumes all memory accesses consume the same amount of time. However, practically this is not so and therefore, data structures and algorithms that can exploit the cache suitably can achieve very higher performance.

Data structures and algorithms that are cache-aware [23] do just this. They contain parameters that can be tuned to gain optimal performance for a specific cache size. This advantage in turn results in an issue; they either need to be tuned for every system (with different cache size) for good performance or they perform well only on some systems (for which it is tuned) while not so well on others. This behaviour however, is not really an attractive one.
Caches in general are based on two basic principles of locality namely, temporal and spatial [23]. Temporal locality states that a program, which uses a particular data has a higher probability of using the same again in the near future. Spatial locality states that a program, which uses a particular data has a higher probability of using some adjacent data in the near future. So any optimal cache-aware algorithm should try and exploit both these properties to achieve optimality.

Harald Prokop in 1999 came up with the concept of cache-oblivious algorithms for this master’s thesis [27] that was later published by Frigo et al. [20]. This arrived as a solution to the cache-aware problem. It also exploits the cache size, however without requiring the tuning to achieve optimal performance. It works well for all cache block sizes since it optimizes the algorithm for one unknown memory level, which automatically optimizes it for all levels.

The basic idea is to recursively split a dataset such that its size reduces and at some point a single portion (split section) of the dataset will be small enough to fit into the cache and will fill at least half of it. This eliminates cache-misses. This idea also eliminates the requirement to know the cache block size. The data structures are designed in manner that a dataset (irrespective of its size) is split appropriately to make good use of caches of all sizes.

The memory model suggested by Prokop [27] considers an infinitely large external memory and an internal memory acting as cache of size M. Data moves in between these two, in blocks of size B. The algorithm cannot control the cache in that it does not explicitly manage the movement of data blocks between the two storage devices. It assumes the existence of a cache manager. This restriction is due to the fact that M and B values are unknown and hence, cannot be manipulated directly. A fixed page replacement policy is used and it is also assumed that the cache is ideal [27]. This means that the cache is fully-associative and the page replacement strategy is optimal. It also assumes that the cache is Tall [27]. A tall-cache is one where the number of blocks present in it (M / B) is much greater than the size of a single block (B). This assumption is represented by the following equation:

\[ M = \Omega (B^2) \] .......................... eqn. (1) [20, 30].

This constraint facilitates the cache-oblivious algorithms to have a large pool of values to guess the block size (B).

Demaine [30] in his paper introduces the various cache-oblivious algorithms and data structures available, explaining the techniques behind those designs. Also, Bender et al. [29] proposed a design for a cache-oblivious B Tree, which was later simplified by Wu et al. [28] while still preserving cache locality. All these designs make efficient use of the cache. Olsen and Skov [26] also analyzed and examined two cache-oblivious priority queues and designed an optimal cache-oblivious priority deque based on one of the priority queues. Also, in 2005, Bender, Fineman, Gilbert and Kuszmaul [31] proposed 3 different concurrent cache-oblivious algorithms that they proved made efficient use of the cache.

A very important aspect of this research is therefore, to analyse these data structures in order to identify the suitability of cache-oblivious structures for our purpose.
3 Research Methodology

This research project entails developing a Google BigTable-like database, for shared memory multi-core systems. This implementation will be a simplified structure of the database that will be cache-oblivious [27] and in-memory [18, 19] to try and achieve performance benefits like speed-up, as well as scalability in a multi-core environment. It will involve creating a concurrent cache-oblivious B-Tree, based on the design proposed by Bender et al. [31]. Another vital aspect of the research will be to perform data retrieval operations in parallel and then evaluate the efficiency and usability of the design. This is crucial as it will help us assess the suitability of a multi-core environment for such huge distributed database systems.

The functional and architectural details of the proposed system are described in the following sub sections.

3.1 System Overview

The proposed design for this database system comprises of two main parts, the underlying Data Structure to hold the data and the API to query the database with. The data structure is cache-oblivious and will reside in memory. The disk will be used to store backup of the data to ensure durability. The cache-oblivious model is primarily based on the Packed-Memory Concurrent Cache-Oblivious B-Tree model proposed by Bender et al. in 2005 [31], that contains both lock-based and a lock-free versions of the structure for concurrency control. Implementing this model will ensure that the data can be accessed concurrently. B-Trees minimise the number of disk accesses, which is critical to this design because of its in-memory nature.

An important point to note here, is that all data needs to be stored in the key/value format and sorted based on the unique row key. Every BigTable data is identified by a unique combination key (row, column, timestamp). It is therefore essential for the data structure to be able to support this and yield good performance.

The API will include a set of system functions to create and manipulate the data structure, maintain operation logs, provide thread-level security (for concurrency), and so forth. Also another set of user accessible APIs will allow users to create the database, populate it with data, add new data, to delete, to append and to retrieve data. The retrieval operations, primarily consisting of search queries, will be parallel in nature. These APIs will be designed conforming to the Google’s API model.

3.2 System Model

B-Trees have been one of the most predominant data structures that keep data sorted, allowing insertions, deletions, searches, sequential reads with very low response time. It is a generalized binary search tree [33], optimized to handle large data sets. The Packed-Memory Concurrent Cache-Oblivious B-Tree model [31] consists to two structures combined into one; a static cache-oblivious Binary Tree [27] and a packed memory data structure [29].

The static cache-oblivious binary tree is a static binary tree based on the van Emde Boas (cache-oblivious) layout. The tree, in memory is an array, and its nodes can be traversed in $O(1 + \log_b(N))$ memory transfers and is hence asymptotically optimal [29].
The packed memory data structure is ‘one-way packed’ and stores the data in sorted order in a loosely packed array. It is said to be loosely packed, since the elements are stored with a lot gaps in between to allow for insertions and deletions. One-way packing allows concurrency to be supported. The array is divided into ‘sections’, with gaps within each section to allow insertions, as mentioned above.

The combined structure is a binary tree whose leaves correspond to certain sections in the packed memory array. Each leaf of the tree maps onto each section as its first element. The other nodes adjust themselves in the sections accordingly. It will support insertions, deletions, searches as well as range queries. The figure below illustrates the design proposed for the database system.

![Diagram showing the proposed cache-oblivious data structure.](https://example.com/diagram.png)

Figure 9: Proposed cache-oblivious data structure. It is a modified design based on the work of Bender et al. [31]. The binary tree will contain the combination-key of (row, column, timestamp) as its nodes, as shown for the root node ‘55’. Thus, each number in the nodes (like 55) here is used to represent the combination-key. The tree will be created only once and hence static. The actual data will be stored in the packed array below the tree. This array contains in addition to the data, the combination-key as well. Thus each element in the array will be a complex data type comprising of a key and its data, as illustrated using the key value of ‘44’. The array again will be divided into sections; the black bold sections in the diagram. Each section will contain a few elements and some gaps for insertions. For e.g. the first section will contain only two key/value pairs, 1 and 5 (keys + their data). The remaining array locations in that section will be empty. Also the maximum value of a key, it will be able to hold is 8 (shown in grey). The next key (9) will be part of the next section, which can hold a maximum of 21 (shown in grey) and so on. Also, the leaves of the tree will map onto the first element of every section as shown above.

Figure 9 above is the proposed data structure based on the model described earlier. This structure can be effectively used to store the data in the key/value format, as required for this project. The choice of using the binary tree is important, since it ensures that the data elements added are stored in sorted order. The actual data, along with its combination key will be stored in the packed memory array, while only the keys of some of this data will be held in the static binary tree acting as the index. Thus, the nodes of the binary tree will contain the combination key (row, column, timestamp) information for only a few of the data elements. Also, since the tree is static, it will get created only once; the first time the table is created. The keys (present in the leaf nodes) will map onto the packed array (shown in the diagram)
that will contain the actual data along with their individual keys. Thus, this structure will ensure that any operation (add/delete/search) will traverse the tree and reach the appropriate section of array containing the data. Also, since this array is loosely packed (contains gaps), insertion operations to the table, will result in data getting added to the packed array, in the gaps of a section.

The proposed structure is also beneficial in terms of performance. It will remain in the main memory throughout, hence low response time. Moreover, mapping of key to value, both held in memory, will add to this speed. Also, its cache-oblivious nature will always ensure optimality.

3.3 Data Manipulation and Retrieval

Once the data structure is in place, it will be utilized to create the database. This will include inserting data into different columns. The system will handle the generation of the row key and timestamp, in case the latter is not specified by the user. The unique row key, the user specific column name and timestamp will together act as a unique combination key for the data.

The various operations supported will be insertion, deletion of both data and table, data append, range queries and search queries. There will be no random data write operations like BigTable. The supported operations will be made accessible to the user via a set of user APIs. However, a set of system APIs will also be designed to handle background operations. These will include an array rebalancing operation which is mandatory for data deletions and is optional in the case of insertions to the data structure. Rebalancing is the re-arrangement of the packed array to adjust the element density in its sections. Also, there will be a disk-write operation to save a database backup image onto the disk. A transactional log will be maintained to store updates. Its job will be to keep a track of all transactions to the database. Once the file size is about to exceed, all updates will be written to disk.

The most important runtime system operation will be to execute the user queries in parallel. Since all queries may not be completely parallelizable, they need to be created in a manner that they are able to make maximum use of the processor cores. In this project however, a single query will be designed and parallelized to be executed on several multi-core systems for performance evaluation.

3.4 Evaluation

The system will be evaluated on several multi-core machines. Specifically, a query will be executed in parallel on multi-core systems to analyse the database performance, its scalability and efficiency. Table 1 below lists the various production system hardware specifications on which the evaluation will be conducted.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>System Name</th>
<th>CPU Type</th>
<th>No. of Cores</th>
<th>No. of Threads (total)</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nehalem</td>
<td>Core i7</td>
<td>4</td>
<td>8</td>
<td>6 GB</td>
</tr>
<tr>
<td>2</td>
<td>Niagara II</td>
<td>Sun Solaris Ultra Sparc T2</td>
<td>8</td>
<td>64</td>
<td>32 GB</td>
</tr>
<tr>
<td>3</td>
<td>MCore 48</td>
<td>AMD</td>
<td>48</td>
<td>48</td>
<td>128 GB</td>
</tr>
</tbody>
</table>

Table 1: Production system configurations for performance evaluation.
Due to the complexity of the project and the limited time frame available, a comparative analysis between the developed system on multi-core and the distributed Google BigTable will be carried out only if time permits.

3.5 Project Execution Plan

The development of this project will be done in the Java Programming language (Java 6). Java is platform independent, supports concurrency in the form of the Fork/Join framework that is essential for this project. The environment will be Eclipse Helios. The Eclipse testing framework (TPTP – Test and Performance Tools Platform) will also be used for testing and debugging purposes along with JProfiler 6.1 (Java Profiler). The system will be executed and tested on both Windows and Unix based operating systems.

The project will be executed iteratively; each iteration involving design, development and unit testing. From now on the project is divided into three basic stages: - implementation, testing & evaluation and dissertation write-up. The implementation phase spans for 8 weeks between 01/06/2011 and 26/07/2011. This will include the design and implementation of the proposed cache-oblivious data structure illustrated in Figure 9 above for both lock-based and lock-free versions. This will be followed by a series of experiments to ascertain their performance before finalising a structure for query evaluation. This will be followed by a unit testing of the final implementation. Next, the APIs to support system and user operations on the database will be developed, followed by another series of unit tests on the code. Finally, the data retrieval operation for multi-core machines will be created and parallelized, for performance evaluation. Keeping in mind the complexity of the complete system and time constraints associated with this project, only the above mentioned functionalities will be implemented. These will enable us to build the basic structure with the core features, and allow us to evaluate this design. However, if time permits, the developed system will be compared and evaluated against the existing distributed databases (BigTable/HBase).

Although unit testing will be performed in parallel with the implementation phase, another 2 weeks for a complete system test and integration test, along with evaluation will be required. The period between 27/07/2011 and 11/08/2011 will be dedicated to this. The evaluation will include running the implementation and executing parallel queries on various production systems (shown in Table 1 above). The results obtained will be analysed for performance gains and other metrics.

The preparation for the dissertation report will be done alongside the implementation. The results of conducted experiments, research analyses and so forth will also be documented along with the development activity. However, the final write-up phase will start from 01/08/2011 and will extend up to the submission date of 09/09/2011; a duration of 40 days. The complete project schedule is present in Appendix 1, at the end of this report.

3.6 Project Deliverables

The deliverables for this research project include the complete source code of the implementation and a dissertation conforming to the University guidelines. The source code will also include the test codes and evaluation codes used. Similarly, the dissertation will consist of the background, experimental results, a detailed evaluation section describing the techniques used and the performance results obtained, supported by an analysis of the outcome.
4 Risks and Challenges

Having discussed the project details so far, it is essential to understand the challenges involved in the development activity as well. The first part of development involves creating the cache-oblivious data structure. The sheer complexity of the structure outweighs the other implementation challenges.

The next most crucial activity is parallelising a search query for the system. Parallelization on multi-core architectures is associated with several challenges, the key ones include the multi-core software triad [50]. The triad is, achieving high performance, ensuring reliability and lowering development time. These encompass risks like race conditions, load imbalance, thread-safety, synchronization and so forth. Also, the fact that all operations might not be completely parallel in nature makes the task of query parallelization much tougher.

To avoid and cope up with the challenges identified, it is essential to make the right design decisions. Appropriate analysis to create accurate partitions is crucial. Also, the use of locks or mutexes avoids race conditions. However, use of locks is highly detrimental to system performance. Again the use of appropriate parallel frameworks like OpenMP [47], Java Fork-Join [49], Cilk [48], etc. are a good way to handle such parallel tasks. Knowledge of involved risks and potential challenges will help us avoid such issues and allow us to create a system that attempts to overcome them.
5 Conclusion

This research involves the design and development of a subset of the Google BigTable [3] database system for multi-core machines. The system currently used by Google is distributed in nature and does not exploit thread-level parallelism of the individual machines in the cluster. This project aims to explore the possibility and assess the efficacy of thread-level parallelism for such huge databases.

This report has presented the Google system and its open source counterparts, the underlying file systems and the distributed parallelism techniques used. It has also described alternative database technologies (IMDBs) and cache-oblivious data structures to provide the readers the necessary background knowledge, before proceeding to explain the research methods utilized to build the proposed system. Presenting a wider context was essential due to the complexity of the system as a whole.

The proposed technique of development was presented. It includes building a cache-oblivious structure that resides in-memory, allows insertions, deletions, data appends and parallel queries, is secure, scalable and aims to provide near-optimal parallel performance and speed-up. Since the system being developed is a subset, features like fault tolerance and security will not be implemented. The strategy involved, programming environment to be used and development methodology to be followed were also put forth. The evaluation mechanism was discussed next.

Realization of this project and the achievement of the design goals will be an important contribution to this research area, given the risks and challenges involved. If the desired result is achieved, it will prove to be beneficial for huge cluster-based databases that will be able to further improve their performance and response time by exploiting thread-level parallelism in multi-core.
References

30. DEMAIN, E.D. “Cache-Oblivious Algorithms and Data Structures”, in Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, University of Aarhus, Denmark, June 27–July 1, 2002.
34. MING-YANG-KAO. Encyclopaedia of algorithms. Page 123.
40. DEMAIN, E.D. “Cache-Oblivious Algorithms and Data Structures”, in Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, University of Aarhus, Denmark, June 27–July 1, 2002.
44. MING-YANG-KAO. Encyclopaedia of algorithms. Page 123.
50. DEMAIN, E.D. “Cache-Oblivious Algorithms and Data Structures”, in Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, University of Aarhus, Denmark, June 27–July 1, 2002.
## Appendix 1

**PROJECT PLAN:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Design and Implementation Phase</td>
<td>01/06/11</td>
<td>58</td>
<td>26/07/11</td>
<td>01/06/11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cache-Oblivious B Tree design for DataStore</td>
<td>01/06/11</td>
<td>7</td>
<td>07/06/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Preliminary tests to finalize data structure</td>
<td>08/06/11</td>
<td>4</td>
<td>11/06/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Document the test results</td>
<td>12/06/11</td>
<td>3</td>
<td>14/06/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cache-Oblivious B Tree development</td>
<td>15/06/11</td>
<td>21</td>
<td>05/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Unit Testing</td>
<td>05/07/11</td>
<td>1</td>
<td>05/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Analysis and design of APIs (Insert, delete, append operations)</td>
<td>06/07/11</td>
<td>2</td>
<td>07/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Implementation of APIs</td>
<td>08/07/11</td>
<td>8</td>
<td>15/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Unit Testing</td>
<td>15/07/11</td>
<td>1</td>
<td>15/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Query design and implementation on multi-core machines</td>
<td>16/07/11</td>
<td>10</td>
<td>25/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Unit Testing</td>
<td>26/07/11</td>
<td>1</td>
<td>26/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Testing and Evaluation Phase</td>
<td>27/07/11</td>
<td>20</td>
<td>11/08/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Integration Testing</td>
<td>27/07/11</td>
<td>4</td>
<td>30/07/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>System Testing</td>
<td>31/07/11</td>
<td>2</td>
<td>01/08/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Execution of Query on production systems and analysis</td>
<td>02/08/11</td>
<td>7</td>
<td>08/08/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Performance Evaluation</td>
<td>05/08/11</td>
<td>7</td>
<td>11/08/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Dissertation Write-up</td>
<td>01/09/11</td>
<td>38</td>
<td>09/09/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Project Dissertation Drafting and Review</td>
<td>01/08/11</td>
<td>30</td>
<td>31/08/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Final Review</td>
<td>01/09/11</td>
<td>7</td>
<td>07/09/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Project Dissertation Submission</td>
<td>09/09/11</td>
<td>1</td>
<td>09/09/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix 2

IMPORTANT NOTE:

A project involving Google BigTable is also being designed by my colleague. However, the choice of the research area, scope of implementation, methodology/approach, supported features, and the choice of programming language vary completely. Hence, the two projects are separate and un-connected. The individual projects are thus being conducted independent of each other, with the approval and under the guidance by my supervisor.