Acknowledgements

We would like to thank our project guide, Prof. Jayant Umale & co-guide Prof. Rashmi Gugnani who have guided us throughtout the making of this project. Prof. Neepa Shah, our project co-ordinator has been very helpful and supportive of our efforts.

We would like to specially thank The Head of the Department of Computer Engineering, Prof. Deshpande, for granting approval for this project as a BE project. The Principle, DJ Sanghvi College of Engineering, Dr. DJ Shah also merits a mention, because he has been generous enough to grant permission for using the college resources for the project efforts till date.

Our partners in crime who are responsible for working on the remaining 2 modules have been a pleasure and lots of fun to work with, Sandy and Pascii with their scrutiny filled vision have spotted many a mistake that we have made during the course of designing & implementing the data storage engine, and hope they will continue to. Anu, Shetty & Prakash have been a pleasure to work with, and we hope we can continue like this even in the future.

Mr. Sridhar Ganti's help has also been much appreciated since he was the one who first made us realize the basic concept of the project, and the complexity we were about to set foot to. He was the one to suggest separating the distribution & replication part from the rest of the project, since it is logically independent from the rest of the Data Storage Engine.

Last but not the least, we would like to thank all our friends & family members who have supported us throughout this ordeal, and have backed our effort and have had faith in us.

Contents

Abstract

There are many relational databases in the market, both commercial and open source. However, we have identified what we believe to be major drawbacks in their architectures. Having done this, we propose to build a new relational database system, that will, from the ground-up, address these deficiencies. Our group will implement a Storage Engine for such a database.

We are implementing storage engine for a larger Distributed Database project. Though our project is independent in itself, it will be meaningless if the perspective of the whole Distributed Database is not taken into consideration. Therefore, we will, both in this report, and the final thesis, refer to TDDB (*T*he *D*istributed *D*ata*B*ase) as a whole.

1 Existing system

We realize that existing systems do not have provision for automatic distribution of data, and support for amalgamating that distributed data automatically on issue of a SELECT statement.

They merely support full and partial replication which does have it's advantages and disadvantages. There are a few commercially available systems that do have some *manual* support for distribution of data but nothing that does this automatically. This means that the user has to be aware of the site(s) containing the tables, or fragments of the tables. Another limitation of replication is that there is a limit beyond which a system can scale-up vertically[1] so we need to devise means by which we can scale-out horizontally.[2]

2 Problem Definition with scope of the project

2.1 Problem Definition

Current commercial implementations allow users to scale vertically by means of adding more hardware, or implementing replication by means of replicated copies of the data. However, keeping in mind the current trends in Data management including Data Warehousing, and Data Mining[which are data intensive operations], we need a solution which allows the data to scale horizontally. Scaling horizontally simply means distributing the data on various machines, and hence distributing the computational as well as storage burden on the various machines.

[1]Vertical scale-up means scaling-up by adding more hardware. In fact, Scale-up servers are large SMP systems with more than four CPUs and one instance of the operating system (OS) that covers all processors, memory, and I/O components. Generally, these resources are housed within a single chassis or "box," and resources are added to the box via system boards. Memory is shared in SMP systems so all processor and I/O connections have equal access to all memory. These vertical systems are also "cache coherent," meaning information is maintained on location of all data regardless of cache or memory location.

[2]The alternative to vertical scaling is horizontal scaling, which works by networking racks or clusters of volume servers. Typically, scale-out systems are linked together via standard network interconnects such as Fast Ethernet, Gigabit Ethernet (GBE), and InfiniBand (IB). Resources are contained within "nodes" — small servers with only one to four CPUs. Each node has its own processor, memory, and operating system. Resources are added by putting more nodes on the rack, not by adding more resources within a node. Memory in a horizontal architecture is distributed, meaning it's typically accessed by each node's CPU and isn't shared across the system.

Another aspect with respect to current implementations is the method of enforcing the ACID properties on the database, and the efficiency concerns which may not be quite up to the mark.

Hence, we would like to develop a system that allows the data and computational overhead to be scaled horizontally by means of distribution. And re-implementing the storage engine to try and iron out the defects mentioned above.

2.2 Scope of the Project

This project would be limited to data storage and retrieval, and may be used for Data Warehousing and Mining applications too. However, expecting utmost efficiency from it would not be wise, because it has many integrity constraints to take care of while processing the data. Hence, data correctness, and preserving the integrity of the data and at the same time ensuring reasonable efficiency would be the primary goals of this venture.

3 Proposed Solution

The solution to the above problems lies in the automatic distribution of data, which would allow:

1. Dynamic selection of a query plan based on which machines contain the required tuples,
2. Distributed Join, and Select.
3. Completely automatic distribution[vertical and horizontal fragmentation] of the data[tuples]

A distributed database consists of features like data distribution, data replication and options for choosing Storage Nodes for these activities. However, there are no commercially available databases which have features which allow automatic distribution of data and automatic Distributed Query execution. None have the provision for automatic fragmentation. In a non-distributed database, the failure of any one site may lead to the failure of the whole database, which would lead to the global unavailability of data.

Many databases are overloaded because of excess data stored in them. So, if one site[the server] goes down all clients suffer from unavailability of data.

We are trying to implement a distributed database system to counter the above mentioned problems & shortcomings. There will be many data storage nodes and

single server site. These data storage nodes are controlled by the single server site. May it be any task, the server handles it by making decisions on which data storage node to be used.

3.1 Distribution

The primary function of TDDB is the distribution of the data as required. For this purpose we will be using Replication & Fragmentation. Both of this can be used together so as to store the data as required and as convenient for the database to act on.

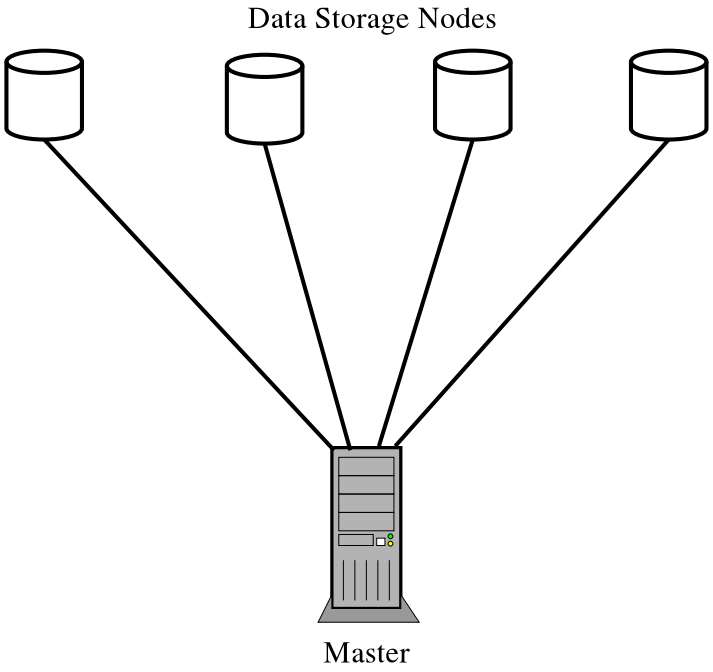

Figure 1: Relationship between the Master & the Data Storage Nodes

Figure 1 shows the relationship between the *Master* and the *Data Storage Nodes* in *TDDB*. A single *Master* is responsible for many *Data Storage Nodes*, and directly controls them. The interface between them is that of *SQL*, so any other database can seemlesly be plugged in place of the *TDDB Data Storage Engine* on the *Data Storage Nodes*, thus making it a kind of *Heterogeneous Distributed Database System*.

The Master in figure 1 can be one responsible for either *Distribution* or *Replication*. The structure and logical structure remains same for both. This is the

reason why we can combine both these techniques[*Distribution & Replication*] in man possible ways. For eg. the *Data Storage Node* can actually be a *Master* which is connected to many other *Data Storage Nodes*.

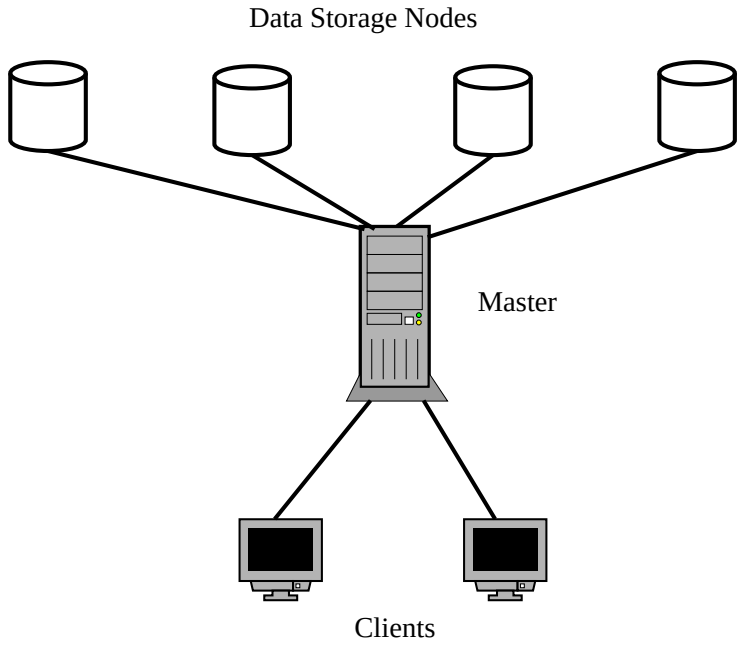

Figure 2: Relationship between a single TDDB Master & Clients

Figure 2 shows the relationship between a single *TDDB Master* & Clients. Here, there is just a single *TDDB Master* which is responsible for all Transaction processing. The system is pretty straight forward, and works as it there was a single *local DBMS* running at the *Master site*.

Advantages:

1. All the data in the database is distributed by proper means. The data is distributed evenly on all the *Data Storage nodes*. So none of the sites suffers from overloading of data, or lack of sufficient data.[3]

2. Consider only one site executing a query. If a single machine is searching entire database on its own it would take a considerable anount of time. Now consiter *n* number of data storage nodes in a database. Now the work load

[3]This is also knows as data *skew*. We have assumed *Round-robin* distribution of data in this discussion of distributed databases.

is evenly distributed among the *n* sites. So the Query processing becomes approximately *n* times faster.

Disadvantages:

1. The only disadvantage in Distribution is that managing all the above processes is quite complex. So maitaining the consistency of the database is also quite complex.

3.1.1 Fragmentation

Fragmentation is a process in which the data is divided into parts as required by the database. It depends on what part of data is to be stored at what site.

3.1.2 Replication

Replication is the process of making copies of the data in the database. These copies maybe stored on all or on some server sites as requitred by the user. Also the number of copies of the data may depend on the importance of the data. Data that is used more frequently, or is more critical in nature may be replicated at all sites, while other data may be replicated at just a few sites. According to the importance of the data, replication is done.

Figure 3 shows the relationship between multiple *TDDB Masters* & Clients. Here, there are 2 *Masters* to which each *Client* is connected. A *Client* can request any of the *Masters* it is connected to to provide it with results of a Query, and can also fire *Insert* and *Update* commands at *any* of the *Masters*. It is the responsibility of the *Masters* to perform synchronization between themselves, and ensire the *ACID* properties of the data.

Advantages:

1. Full replication gives high availabilty of data. Since data is copied at all the server sites, data is available everywhere. Now even if any of the sites are down or loose all the data, it can be made available from the other sites.

2. A query failure in a replicated database is of a very low probability. Even if one site is down, other sites can fulfill the request. This gives the data protection from system failures.

3. At a time many non-blocking concurrent reads can be performed on the same data. This results in no loss of time. Instead of waiting for a site performing a read action, another site can be queried, where the data has been replicated and there is no activity being performed.

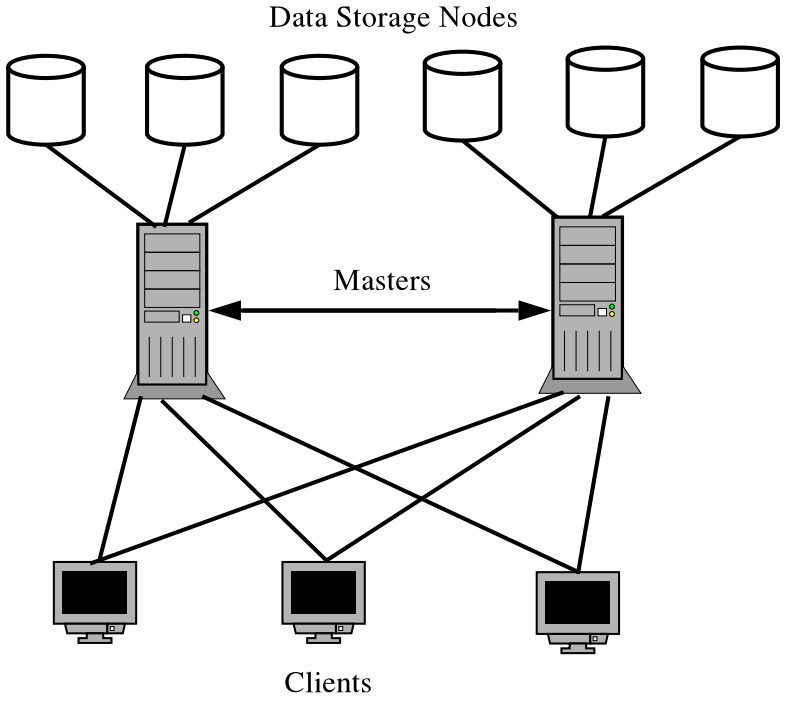

Figure 3: Relationship between multiple TDDB Masters & Clients

4. Also the geographical distance between the sites makes the query processing slow. A site requesting some data from a geographically distant site would slow down the data transfer speed. So, it can be made to approach a nearby site. This lessens the time for transfer of data. So, the query can be executed faster.

Disadvantages:

1. To update a single copy of data all the copies on the other site need to be updated. It means many copies has to be updated at a single time. This results in lot of latency, which results in the system getting generally unresponsive and sluggish.

2. Updation of a copy requires it to be updated it on all sites. If any of the site fails to update it is means no other site can update the copies, even though other sites may be ready for update. So it means failure in updation of the data, which ultimately boils down to the failure of the transacion as a whole.[4]

[4]We have assumed here that we are using a protocol in which if any one site fails to perform

3.2 DDL Commit

Data-Definition Language also known as *DDL* is a special language that is used to define a database schema using a set of data definitions. *DDL* provides commands for defining relational schemas, deleting relations, and also modifying relation schemas. This means *DDL* statements are the ones that provide the features for constructing a database.

In a distributed database, *DDL* statements are given to all sites to do the work of defining the schemas. The effect is seen only on the site where the statements are received and executed. Other sites, which didn't receive the statements, remain unaffected. Later when the *DML*[5] statements to input data or change data on the sites are given, the sites which didn't receive the previous *DDL* statement(s) return an error.

To avoid the above problem from occurring, commit for *DDL* statements is implemented, as is done for *DML* statements. `COMMIT` is a statement which allows a *Transaction* to be executed atomically. If all the sites commit the data successfully, then the commit statement is said to have succeeded. On the other hand, even if one site fails, then the commit is said to have failed.

A similar concept is developed to the *DDL* statements. This is required for use by the 2-phase commit protocol. We call this the `DDLCOMMIT` statement.

4 Requirement Analysis

4.1 Why distributed database?

Distributed database carries out its work using fragmentation and replication. In normal databases, if any server is not working it may make a task to be carried out on entire database impossible to perform due to it's unavailability. Also the databases are overloaded due to unnessecary reasons. Now since the data is distributed, there are many *Data Storage Nodes* to perform this work. So overloading of a particular site can be avoided.

The main purpose of distributed databases is distribution of the data as required by the user.[6]. Replication and Fragmentation are used for this purpose.

the update, the whole transaction is deemed to have failed. However, this is not always the case, and better protocols such as the *Majority Protocol* employ bettwer methods, and allow the transaction to continue as long as a majority of the sistes are up. These are however more complex to implement.

[5] *DML* stands for Data Manipulation Language

[6] In our case, *TDDB* itself can make this decision on its own.

Also, a combination of both can be used with good results. This will store the data as required and handling with it will be convenient.

The need for distributed databases is going to increase in the future. This is because a lot of data is being digitized or being stored in a digital form these days because:

- It is cheaper to store data this way
- It is easier to Search the data
- It is more convenient to manage this data
- It takes up lesser space as compared to voluminous paper files
- Is a more environmentally friendly option

A part of this also arises from the fact that many different kinds of computing devices will be on the network. As mentioned earlier, in distributed database none of the site will suffer from overloading and the query execution will also speed up.

4.2 Why is Replication needed?

Replication is the process of replicating copies of the data in the database. These copies may be stored as and how wished by the server or the user. Replication will also depended on the importance of the data. The more is useful data, more are the sites at which it will be replicated. Regional based data maybe stored only at the site required. Thus replication depends on the data and also the choice of the user.

If full replication is done, it gives high availabilty of data. Even if any of the sites are down or any sites loose all the data, it will be available from any other site.

A query failure in a replicated database has a very low probability. On failure of one site the data request can be fulfilled by another site.

Multiple reads can be performed on same data without loss of time. If there is a read request at a certain site, and a read is being performed at that site, then the request can be transferred to another site.

4.3 Independence from one single machine

In a distributed database, all the data is distributed on many machines, that is the *Data Storage Nodes*. These *Data Storage Nodes* are controlled by the a server,

which is in charge for any and all activities taking place at these *Data Storage Nodes*. The server controls process ranging from construction of database schema to the execution of queries and returning the output. The server can request any of the *Data Storage Nodes* for data that it may require. But it does this on the basis which results in a low cost of executing the query.

Thus a distributed database is independent of a single machine, since data is distributed all over the place.

4.4 Choosing *SQL*

SQL is referred to as *query language*. But it can do just more than just a query language that queries a database. It can define the structure of the data, modify data in the database and also specify security constraints.

SQL is the most influential commercialy marketed query language. SQL uses a combination of relational-algebra and relational-calculus constructs.

SQL has several parts which are very useful to construct a database.

- Data-definition language *DDL*
- Data-manipulation language *DML*
- View definition
- Transaction control
- Integrity

In this project, we will be using SQL as the interface for communication to and from the database. This is because SQL is one of the most widely used language in databases. Almost all people working on databases are well acquainted with the use of SQL. Since project will be implementing SQL, no additional training will be required to work on our system language.

SQL uses realtional-algebra. It is generally considered good practice to create relational databases. Relational querying is easy to understand and implement.

Due to all these features of our system, people and institutions working on our database system won't face problems while shifting/migrating to our system. Thus shifting from the older system to our system won't be hard and will be a seamless transistion for them. This will reduce the initial training cost, and also no time will be wasted training the employees after the transition is made.

4.5 Choosing *C++* as the language of implementation

C++ is one of the most commonly used language for systems programming, and produces High Quality code. C++ is a language which is neither a low level language nor a high level language. C++ has the low level power of C, but at the same time it allows us to abstract our ideas similar to other high level languages.

There are many inbuilt functions and templates which can be used easily. Also defining and using of new functions is easy. There are in-built data-types and also provides us with the facility of creating user-defined data-types. So data-types and the functions can be used as required and needed by the user.

C++ supports *generic programming* by means of a feature called *templates*. *Templates* allow you to write code which is independent of the underlying data type you are using, and allows you to code *data structures and algoritms* such as *sorting, searching, etc...* which are data-type independent.

Finally, the icing on the cake is the *STL[Standard Template Library]* which C++ is well known for. The *STL* is a collection of highly efficient pre-coded templates, which allow you to accomplish routine tasks very easily. The *C++ STL* comprises a large number of data structures and algorithms which are used very commonly.

4.6 Choosing *GNU/LINUX*

GNU/Linux is a very stable Operating System. It works efficiently on many kind of machines, be it one with older or newer hardware configuration. Crashes on linux are rare. Generally most of the servers and machines used for high productivity or tough tasks work on linux.

GNU/Linux quite is secure in comparision with other Operating Systems. Since the servers and other clients are going to store important data, it is necessary to provide a secure environment. The system files which are only visible in root user*[administrator of the system]* can't be modified by normal users. So this ensures Linux provides security to important files, data, documentsm and devices.

Also since GNU/Linux is freely and readily available, it would be easy for anyone to procure a copy. This does not increase the cost of the set-up.

5 Project Design

5.1 Proposed Architcture

The Data Storage Engine which we will be working on will have a design similar to that described below:

5.1.1 Unix File Abstraction

The lowest Level of the Data Storage Engine would be the UNIX File Abstraction layer. This Layer provides a device independent interface to access just about any device on the system using a simple *open()* system call. You can read from & write to this file using the *read() & write()* system calls.

The database engine would operate on just a single file, which could be just Raw disk space for fast access, thus avoiding the OS File System Layer. This would mean that we can format this Raw disk space in any way we wish, and make the data storage/retrieval as efficient as we wish to.

5.1.2 Buffer Cache

The Buffer Cache is a *Page Cache*[7] which stores all the *Most Recently & Frequently* used pages in memory. The algorithm used is a mix of both these algorithms, and is implemented using logic similar to the *2-handed clock algorithm* for page eviction.

The buffer cache will be constructed as a collection of hash-tables with chaining for handling overflows with each bucket provided by it's own mutex.

The Page Evicition function will be called only when all memory locations are exhausted and there is no more room for housing new pages in memory.

The details of the page replacement algorithm are as follows:

- Every time the page is referenced, the 13-bit counter is incremented by one.
- The daemon thread for a particular page_cache is run only if the number of free pages for that page_cache is less than 10% of the total number of pages on that page_cache.
- In the daemon thread, the value page_age is checked against new_lb. If it is $<$ new_lb, then new_lb is set to the new value, else, new_lb is left untouched.

[7] A Page is defined by the TDDB_PAGE_SIZE macro, and is typically 8kB in size. A Page is the smallest unit of data which will be read off the disk at one shot.

- In the daemon thread, the age for every page which has an age count of $\geq$ 1% of the age_max is decremented by one whenever the daemon thread runs for that particular page_cache and encounters that buffer.
- While referencing a page, if the age for a page reaches age_max, it is made $\frac{age\_max}{2}$.
- New pages entering the system are given an age of age_min.
- The pages to be evicted are evicted not by the 2^{nd} pass of the clock but by the thread itself which finds that the number of pages has dropped below min_free. It then tries to free max_free pages.
- The pages to be selected are those which have an age $< lb\_age + 1$. If in the first pass, not enough pages can be evicted, then the alsorithm makes another pass, but this time uses $(lb\_age + 2) * 2$ as the quantity to check against. If this too fails, then, the the quantity is set to $\frac{age\_max}{2}$, and if this too fails, then pages are evicted sequentially as and when they are encountered.

5.1.3 Bitmap Lock Manager

The *Bitmap Lock Manager* is a Page level bitmap lock manager. It has 1-bit reserved for ech page in the database. This ensures reasonable space efficiency of the implementation. All dictionary operations such as:

- Locking a Page[8]
- Unlocking a Page
- Checking if a page is locked

take constant time.

For a database of size *400GB*, assuming the page granularity[9] to be *8kB*, and overhead per page to be 1-bit, the total space overhead is: $\frac{400GB}{8kB}$
$= 52428800bits$
$= \frac{52428800}{8}bytes$
$= 6553600bytes$
$= 6.25MB$

In terms of percentage, this is an overhead of 0.0000152588% which is quite negligible.

[8]Locking a page make take more time if that page has already been locked by some other Transaction, because the current page has to then wait for that Transaction to unlock the page.

[9]also called as Page Size

A Transaction Locks a Page using the bit-map Lock when it is *reading from or writing to* it. This is done to ensure that no other transaction can read from or write to that page while the current Transaction is reading from or writing to it. It also ensures that concurrent transactions do not modify the page at the same time. The reason we have this lock in place while reading from OR writing to the page is that certain *Isolation Levels* such as *Dirty Read* can read from the table while it is in the process of being modified, so such transactions *must* lock the page before reading from it.

5.1.4 Dirty Page Manager

The *Dirty Page Manager* is similar in structure to the *Bitmap Lock Manager* and has similar space & time complexity, but the reason for it's existence is entirely different from that of the *Bitmap Lock Manager*.

The *Dirty Page Manager* is used to mark pages as *Dirty*. A page is called as *Dirty* when it has been modified by a Transaction, but it has not been written back to disk. When a page is *dirty*, it may be:

- Entirely in memory
- Partially in memory, and partially on the Redo Log
- Fully on the Redo Log

Thus, it would be an error for another Transaction to read data off that page, because it would undoubtedly contain *stale* data.

Another reason for marking pages as *dirty* is that *Control Information* in the header needs to be updated, and Transactions can not afford to read *stale Control Information* from these page headers. If such a situation occurs, then it may lead to problems such as Transactions using the same free space to write user data.

So, from the definition of a *dirty* page, it follows clearly that only the originating transaction[10] can read from or write to that page again. Other transactions must thus *always query* the *Dirty Page Manager* to check if the page they are trying to read from or write to has been marked as dirty by an earlier transaction. If so, then, they can not rely on the information they read from that page. These transaction can then either:

- Try to read data from another page, or
- Wait for the current page to be marked as *clean*

[10] *Orignating Transaction* here means that one that marked the page as dirty.

5.1.5 Redo Logger

How Consistency is to be maintained?

The value of the attributes when being updated, care is taken that if update doesn't take place then the old values are not lost and replaced in the place of the unchanged attributes. This is the property of consistency of a Database.

There are two ways to implement consistency.

1. Undo
2. Redo

1. *Undo:*
   In this method, the attribute values which have to be changed, are stored in a temporary location before being changed. After storing the values in the location, the new values are written in the location of the attributes. If the process of writing of new values takes place completely, then the old values stored are completely discarded. Or else if they are not written in the attribute location, then the old values from the temporary location are copied back to their respective location. This undo method is used where success rate of transaction is high.

   Since the system roll backs to its old state if any of the transaction did not succeed, this method is called *Undo*. Also the temporary location used to store the attribute values is called as *Undo Log*.

   Advantages:

   - Undo method is used when most the transactions in the database succeded.

   Disadvantages:

   - During updating the attribute values, other concurrent readers have to refer to the temporary location. This may be time consuming, since an extra level of indeirection is needed.

2. *Redo:*
   In this method the attribute values which have to be changed are stored in a temporary location before being changed. After storing the values in the location, the new values are written in the location of the attributes. If the process of writing of new values takes place completely, then the new

values stored are completely discarded. Or else if they are not written in the attribute location, then the new values from the temporary location are still present in the temporary location. This redo method is used where success rate of transaction is low. That is the transactions tend to fail.

Since the system proceeds to its new state if the transaction suceeds the method is called *Redo*. Also the temporary location used to store the attribute values can be called as *Redo Log*.

Advantages:

- Redo method is used when most the transactions in the database fail.
- During updating the attribute values, concurrent readers can refer to the original location on disk of the attributes. This means that concurrent readers do not suffer a performance hit when another transaction writes to data in the tables.

Disadvantages:

- The redo process may be time consuming.
- The transaction which writes data to the tables must now refer to the *Undo Log* to read the new *uncommitted* values in the table. This means an extra level of indirection is needed, which slows things down a bit for the transaction which made the changes.

Comparision of Undo and Redo

Generally the *redo method* is preferred over the *undo method*. This is because the new attribute values are stored in final database only if the transaction succeeded. Whereas in the *undo method* whatever may be the fate of transaction, new attribute values are written. This leads to complication w.r.t multiple transactions reading off from the same table. If the *Undo method* is being used, then the concurrent transactions need to read the un-committed values from the undo-log. This inderection can be the reason of some inefficiency while accessing the database. Also, this method increases the internal complexity of the database implementation. However, if the *Redo* method is being used, we need not be worried about any such problems, because new values are written to the database ONLY if the transaction succeeded.

TDDB uses the *Redo Method* to ensure consistency of data in the database, and for this purpose, a special region on disk called as the *Logging Segment*is reserved to house the *Redo Log*. When any transaction wishes to modify any data, it is written out to the *Redo Log*, and then the data in the *Redo Log* is played. If it were to fail, then it can be replayed when the system is re-started. Thus, consistency of data is ensureed.

5.1.6 Table Lock manager

The *Table Lock manager* is responsible for performing locking on the tables[11] and thus ensuring consistency of data in the database.

There are currently 5 modes in which a table can be locked. They are:

1. *Read Mode:* A table is locked by a transaction in this mode if it wishes to *read* data from it.

2. *Insert Mode:* A table is locked by a transaction in this mode if it wishes to *indert* data into it.

3. *Update Mode:* A table is locked by a transaction in this mode if it wishes to *update* data in it. *Updation* also includes *deletion* of data from the tables.

4. *Serializable Mode:* A table is locked by a transaction in this mode if it wishes to lock the table in *Serializable* mode. This is a special mode in which other *Serializable* transaction are not allowed to perform *dirty or phantom reads* from the table, and must wait for the locking transaction[12] to unlock the table. This mode corresponds directly to the *Serializable Mode* mentioned in the *SQL92 Standard*.

5. *Critical Mode:* A table which is being modified[data is being written to the database from the *Redo Log*] has to be locked in this mode before any modification is possible. This mode ensures that *no other* transaction can write to the table while some other transaction is writing to it. It thus enforces *mutual exclusion* in the true sense.

5.1.7 Division of Disk Space

How is Raw disk space to formatted for use by TDDB?

TDDB creates managable units of disk space by which it logocally groups these units and makes sure that this dsk space management is made possible in an easy and efficient manner.

Starting from the highest to the lowest granularity, the fllowing units of disk spacs are present in *TDDB*:

1. *Group Block:* A group block is the most granulary unit of disk space in *TDDB*. The Raw disk space obtained from the Disk is partitioned into many *Group Blocks*.

[11] This means table-level locking.

[12] One that has currently locked the table.

2. *Chunk Block:* Each group block contains many *Chunk Blocks*. These chunk blocks have a fixed size just like the group blocks they belong to. Each chunk block has pages of *only* a specific type.

3. *Disk Page:* Each *Chunk Block* contains many *Disk Pages*. These *Disk Pages* are of many types, a few of them being:

   - *Fixed-size Tuple Page:* which holds each tuple's fixed-size portion.
   - *Overflow Page:* which holds overflow entries for many types of fields in a table.
   - *Blank Page:* which generally is used to store *CLOB* entries.

5.1.8 Indexes

Indices are commonly used to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. However, indexes also add overhead to the database system as a whole, so they should not be overused.

1. *B-tree Index:*
   TDDB uses a B-tree structure to organize index information. A fully developed B-tree index is composed of the following three different types of index pages or nodes:

   - One root node. A root node contains node pointers to branch nodes.
   - Two or more branch nodes. A branch node contains pointers to leaf nodes or other branch nodes.
   - Many leaf nodes. A leaf node contains index items and horizontal pointers to other leaf nodes.

   The fundamental unit of an index is the index item. An index item contains a key value that represents the value of the indexed column for a particular row. An index item also contains rowid information that the database server uses to locate the row in a data page.

   Nodes: A node is an index page that stores a group of index items.

   How does TDDB create and fill a B-tree index?

   Creation of Root and Leaf Nodes: When you create an index for an empty table, the database server allocates a single index page. This page represents the root node and remains empty until you insert data in the table.

At first, the root node functions in the same way as a leaf node. For each row that you insert into the table, the database server creates and inserts an index item in the root node.

When the root node becomes full of index items, the database server splits the root node by performing the following steps:

- Creates two leaf nodes
- Moves approximately half of the root-node entries to each of the newly created leaf nodes
- Puts pointers to leaf nodes in the root node

As you add new rows to a table, the database server adds index items to the leaf nodes. When a leaf node fills, the database server creates a new leaf node, moves part of the contents of the full index node to the new node, and adds a node pointer to the new leaf node in the root node.

Eventually, as you add rows to the table, the database server fills the root node with node pointers to all the existing leaf nodes. When the database server splits yet another leaf node, and the root node has no room for an additional node pointer, the following process occurs.

The database server splits the root node and divides its contents among two newly created branch nodes. As index items are added, more and more leaf nodes are split, causing the database server to add more branch nodes. Eventually, the root node fills with pointers to these branch nodes. When this situation occurs, the database server splits the root node again. The database server then creates yet another branch level between the root node and the lower branch level. This process results in a four-level tree, with one root node, two branch levels, and one leaf level. The B-tree structure can continue to grow in this way.

2. *Hash Index:*
   Hash Indexes use *Hash tables* to implement indexes. A B-tree Index can not be used for multi-attribute indexes, but is well suited for single-attribute indexes. Hence, Hash Indexes are needed in such situations. Hash Indexes can not be used is when relational operations other than *equality*, such as $<, >, \leq, \geq$ are to be used, because unlike B-trees, Hash Tables do not work on comparison between key values, but work on the principle of equating the *Hash Value* of the key with the *Hash Value* of other keys.

5.1.9 Join Processing

The efficient implementation of joins has been the goal of much work in database systems, because joins are both extremely common but rather difficult to execute efficiently. The difficulty results from the fact that joins are both commutative and associative. In practice, this means that the user merely supplies the list of tables to be joined and the join conditions to be used, and the database system has the task of determining the most efficient way to perform the operation. Determining how to execute a query containing joins is done by the query optimizer. It has two basic freedoms:

1. *join order:* because joins are commutative, the order in which tables are joined does not change the final result set of the query. However, join order does have an enormous impact on the cost of the join operation, so choosing the right join order is very important.

2. *join method:* given two tables and a join condition, there are multiple algorithms to produce the result set of the join. Which algorithm is most efficient depends on the sizes of the input tables, the number of rows from each table that match the join condition, and the operations required by the rest of the query.

Many join algorithms treat their input tables differently. The input tables are referred to as the outer and inner tables, or left and right, respectively. In the case of nested loops, for example, the entire inner table will be scanned for each row of the outer table.

Join algorithms

There are four fundamental algorithms to perform a join operation.

1. *Nested loops:*

   This is the simplest join algorithm. For each tuple in the outer join relation, the entire inner join relation is scanned, and any tuples that match the join condition are added to the result set. Naturally, this algorithm performs poorly if either the inner or outer join relation is very large.

   A refinement to this technique is called ”block nested loops”: for every block in the outer relation, the entire inner relation is scanned. For each match between the current inner tuple and one of the tuples in the current block of the outer relation, a tuple is added to the join result set. This variant means that more computation is done for each tuple of the inner relation, but far fewer scans of the inner relation are required.

2. *Merge join:*

   If both join relations are sorted by the join attribute, the join can be performed trivially:

   - For each tuple in the outer relation,
     - Consider the current ”group” of tuples from the inner relation; a group consists of a set of contiguous tuples in the inner relation with the same value in the join attribute.
     - For each matching tuple in the current inner group, add a tuple to the join result. Once the inner group has been exhausted, both the inner and outer scans can be advanced to the next group.

   This is one reason why many optimizers keep track of the sort order of query nodes — if one or both input relations to a merge join is already sorted on the join attribute, an additional sort is not required. Otherwise, the DBMS will need to perform the sort, usually using an external sort to avoid consuming too much memory.

3. *Hash join:*

   Applying the Hash Join algorithm on a inner join of two relations proceeds as follows: first prepare a hash table for the bigger relation, by applying a hash function to the join attribute of each row, and then scan the smaller relation and find the relevant rows by looking on the hash table.

4. *Semi join:*

   A semi join is an optimization technique for joins on distributed databases. The join predicates are applied in multiple phases, starting with the earliest possible. This can reduce the size of the intermediate results that must be exchanged with remote nodes. Thus reducing inter node network traffic. It can be improved with a Bloom-Filter (hashing).

5.1.10 Security Issues

Security is a major concern in any non-trivial database management system, especially a distributed database management one such as *TDDB*, which relies heavily on the *network* for transfer of data.

We have developed an access method which tries to reduce the risk of the security of the system being compromized, and confidential user data being leaked.

We have thought of a strategy which tries to minimize[not totally eliminate] the influence of the man in the middle. We’ll describe it below. However, before

we delve into the details of that, let's see why we considered the public/private key solution not good enough.

1. Who would have the private key? If the server were to have it, then the public key would have to be transferred to the client, which can be picked up by the man in the middle[referred to as *MIM* in the rest of this document], and then used to encrypt data. Ok, you say that the password would be encrypted within the public-key encrypted message that the client sends, but then, the *MIM* an just pick up that request that the client sends to the server, and keep firing it, and overloading the server[DOS attack].

2. The client can send the server encrypted information, but how can the server send the client encrypted information? Of course, it can not, because the client has only a public key. So, you say make another set of keys of whih the client has the private key, and the server has the public key. But then again, the *MIM* can cause havoc by intercepting packets and sending them for another query that the client fires. It[the client] will have no clue that they are junk packets. Ok, this can be solved using *TIDs* which must match. Meaning when the client sends a request, it also sends a *TID*, which the server must send back to indicate it is a valid response. This *TID* could be sent in an encrypted fashion. However, we are again stuck at the problem of encrypting the *TID!*

3. *PGP* has been cracked, and having a static hash key is always not advisable. This leaves a lot of room open for hacks.

So, we have decided to go with the apporach described below. Table 1 contains the list of symbols used.

P	→The password of the user.
R	→A string of random numbers.
Q	→The user's query string.
h(X)	→Hash function applied on string X.
P+R	→Concatenation of 2 strings P & R.

Table 1: Symbols used

1. The client tells the server that it wants to initiate a transaction.

2. The server sends sends the client a random string R. This is sent unencrypted.

3. The client constructs $h(P + R)$ with the P obtained from the user. Simultaneously, the server also computes $h(P + R)$ from the P stored in it's database.

4. The client now gets the user's query string Q, and computes $Q \oplus h(P + R)$. This is the string that is sent to the server.

5. The server XORs the received string with $h(P+R)$, and gets back the actual data that it is supposed to receive.

Thus, the *MIM* is rendered practically impotent! The problem can arise only when the server generates R which has already been generated before and the same user is using it, and the *MIM* has logged the previous R, and has been able to determine $h(P + R)$ for that $P\&R$, and is quick to enough to retrieve it. A very rare occurance if R is a pretty large number.

5.1.11 Multi-threaded Memory Allocator

TDDB is a multi-threaded application capable of running on an a multi-processor machine, and taking advantage of features like:

- Multiple Threads of exection
- Multiple Processors

It uses these features to produce the results to user queries more efficiently[13].

Theory of Design of the Allocator

This implementation involves maintaining 2 free lists for each thread, for each bin-size. One of the lists would be the allocation list, and the other would be the deallocation list. Whenever objects are to be allocated, they are given off from the allocation list, and whenever, they are freed, they are put into the free/deallocation-list. This is a truly multi-threaded data structure, which is not quite scalable, but multiple data structures of the same kind make it linearly scalable over a large domain. This means that while neither the allocation/deallocation lists interferes with each other, also no locking is required. This scheme also simultaneously solves the problem of locality, producer-consumer model programs, and false sharing to a large extent. There is also a global heap for each bin-size,

[13]By *efficiently*, here we mean faster, or in a lesser amount of time.

which is maintained as a linked list of super-blocks, which may not necessarily be from the same physical/virtual page, but are of size 1 or 2 pages, that is the total memory of the links in a logical super-block is 4K or 8K, depending on your choice! One thing to be noted here is that the blocks in the allocate-list may not necessarily be a multiple of the page size in the system. The reason for this will become clear shortly. Now, follows a description of the algorithm for the allocation of memory from the above-mentioned allocator.

Algorithms for the Allocate and Deallocate functions:

- Allocate(Size) Begin

    1. Search the allocate linked list for a free block of size Size. This operation just involves checking the correct bin for the existence of a non-zero link in the bin's linked list.
    2. If such an entry exists, then we just return it to the user. Otherwise, we ask the deallocate-free list to give us some memory, which is at least 1 block in size, so that the current request can be satisfied.
    3. The deallocate-free list will first lock itself, and then query itself to check if it contains at least 1 entry. If so, it will make the Size of the deallocate-list 0, unlock itself, and will return that entry to the requester, or else it will unlock itself and query the global heap for an entry that is 1 or 2 Pages in size.
    4. The global pool in turn will first lock itself, and query itself to check if it contains a valid super-block, which is nothing but a logical page or 2-logical pages. If they exist, they are removed from the free-list, the global heap is unlocked, and the super-block is given to the requester. If not, then the global-heap is unlocked, and a page or a couple of pages are allocated from the page memory allocator, the headers are written, and is returned.

  End.

- Deallocate(Pointer) Begin

    1. Lock the correct deallocate-list, add the entry Pointer to the deallocate-list, and increment the Size of the deallocate-list by 1. If the size of the deallocate-list is equal to a page, or 2-pages, whatever seems convenient, return it to the global-pool, and set the Size of the deallocate-list to 0. The global pool treats the memory list as a super-block, and adds it to the list of its super blocks. Finally, we unlock the deallocate-list.

End.

The addition of a logical page to the global-pool is made possible by first locking it, and then adding the logical page to the front of the super-block free-list. Finally, the global-pool is unlocked.

Thus, it can be seen that there is not strict need of locking at every stage of the allocator. Atomic operations can be implemented using the facilities provided by the underlying Operating System.

5.1.12 SQL processing Engine

The SQL processing Engine is the main interface of the TDDB Engine with the SQL Parser. The SQL Parser will parse SQL syntax, and create a helper object for each type of SQL statement, which will be handled appropriately by the SQL processing Engine. The SQL processing Engine currently handles the following SQL statements:

- CREATE TABLE
- CREATE DATABASE
- DROP TABLE
- DROP DATABASE
- SELECT
- INSERT
- UPDATE
- DELETE
- BEGIN WORK/TRANSACTION
- END WORK/TRANSACTION
- LOCK TABLE(S)
- UNLOCK TABLE(S)
- SET ISOLATION LEVEL
- COMMIT
- ROLLBACK

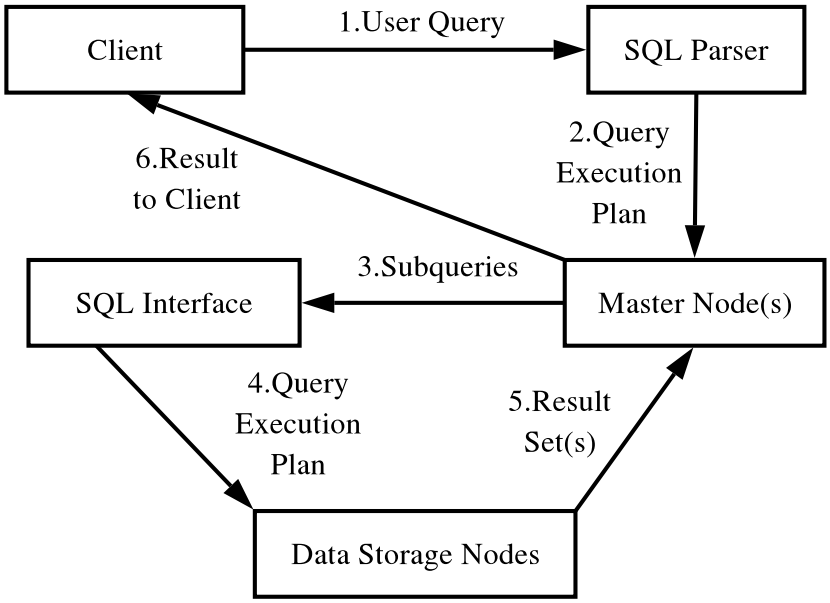

Figure 4: The Data Flow Diagram showing how a Query is processed

5.2 Data Flow Diagram

Figure 4 shows the Data Flow Diagram for how a Query is processed by *TDDB*. The following steps occur in processing of a Query:

1. *Step - 1:* The user fires a Query, and it is checked for syntactic correctness at the client side itself. Syntactic correctness includes things like omission of the *FROM Clause*, or omission of what to *SELECT* in a *SELECT* statement. These errors are independent of the schema of the table, or database, and can be detected at the client side itself. This saves some amount of network data transfers, and reduces the load on the network, and also generally speeds up Query processing.

2. *Step - 2:* The Query is sent to the *SQL Parser*, and is checked for other errors which depend on the schema of the table and database, such as entering a field name in a *SELECT* statement while a field with such a name does not exist in that table, etc.... If the *SQL Parser* detects that there is an error, it notifies the sender about such an occurance.

3. *Step - 3:* The *Master Node* generates *sub-queries* for each of the *Data Storage Nodes* connected to it, and sends these *sub-queries* to each of these *Data Storage Nodes*. Here, many optimizations are possible based on the

type of fragmentation or replication involved. The *Master Node* maintains meta data about all the tables it manages. Hence, it it able to perform such optimizations.

4. *Step - 4:* A process similar to *Steps 1 & 2* is performed here, except that this is at the *Master & Storage Node* interface. Here too, the *SQL* generated is processed by the *SQL Parser*, and a *Query Evaluation Plan* is generated for processing this query.

5. *Setp - 5:* The *Result Set(s)* are sent back to the *Master* after being generated by the *Data Storage Nodes*. The *union* of these constitutes the final result in *most cases*. However, many a time, an *ORDER BY or GROUP BY* clause in the original query needs to be processed at the *Master*, and hence, these tuples are further processed here.

6. *Step - 6:* The final result set is sent to the client by the *Master*. This step marks the end of the Query Processing cycle in *TDDB*.

6 Implementation

6.1 Schedule

6.1.1 Timeline Chart

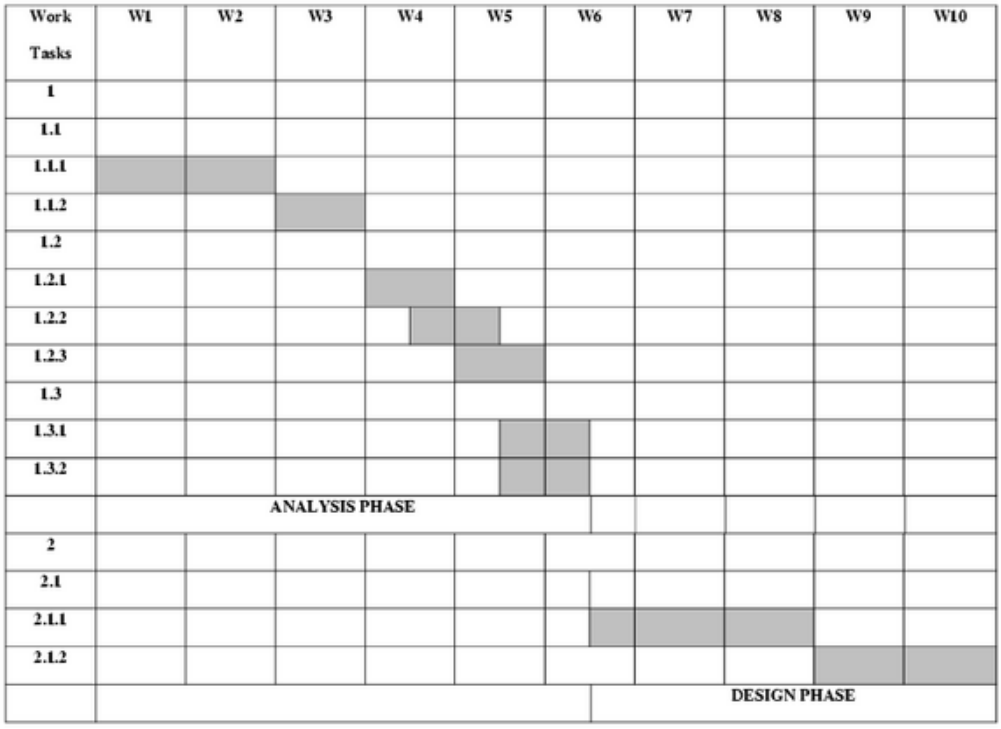

Figure 5: Time Chart: Week 1-10

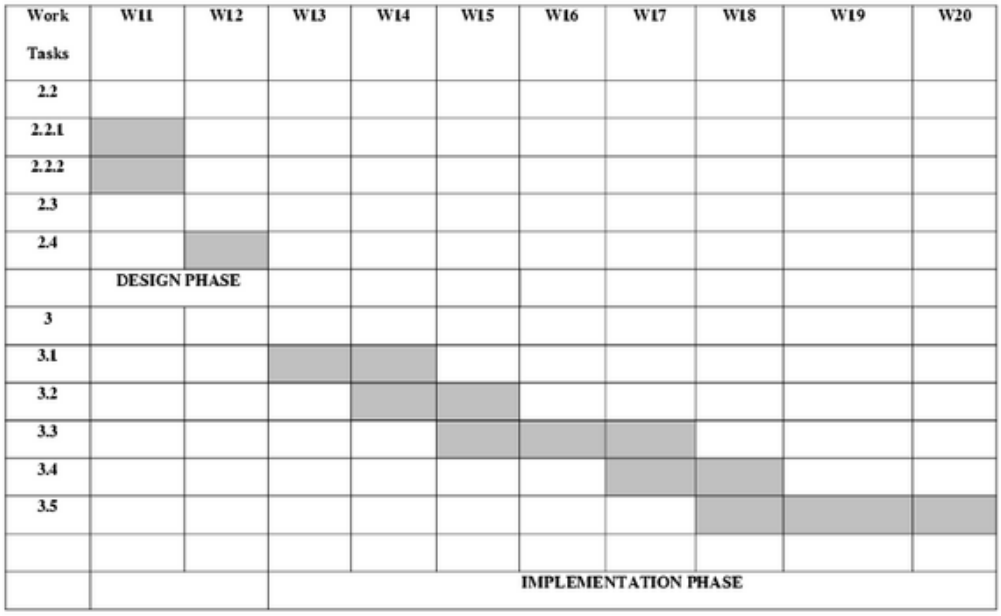

Figure 6: Time Chart: Week 11-20

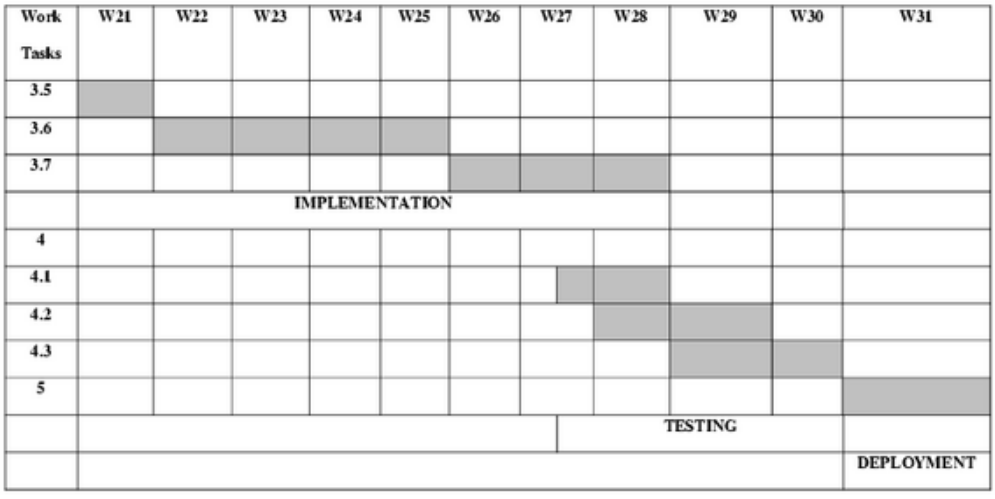

Figure 7: Time Chart: Week 21-30

6.2 Project Resources

6.2.1 Hardware

The hardware required for the project is:

- x86-64 architecture machine.
- Borad-band internet.
- Dual processor machine for testing concurrently executing code.
- Sufficient RAM [8GB] for executing large *Joins*, and *Sorts* on large tables.

6.2.2 Software

- GNU/Linux : Operating System

  We decided to go in for UNIX-like system to develop our project. UNIX is a very old, but highly effective, stable, and reliable, as proven by various systems around the world over the years. In fact, most, if not all, of the major databases you would trust your life with[14], run on various Unices[15].

  The GNU/Linux operating system is a very high-quality implementation of the UNIX standard. Moreover, it's free for use by developers (and businesses that have good in-house technical support). We still haven't decided which distribution of Linux shall be our development platform, but we shall spare no efforts to ensure that the system works on all major Linux distributions out there.

  The ones in reckoning for our attention are :

  - Fedora Core
  - Novell Suse Linux
  - Mandriva
  - Debian GNU/Linux
  - Gentoo

  For now, we are continuing our development efforts on Red Hat - 9.

[14] Hospitals, banks, airports ...

[15] Plural of Unix.

- g++ : A Standards compliant C++ Compiler

  A Standard compliant C++ compiler is required to compile the source code of the project. This compiler should support the full C++ standard, and also allow machine specific inline assembly and C code. Being an optimizing compiler, is an added advantage.

- emacs : A text-editor

  A syntax recognizing text-editor such as emacs which has in-biuld support for languages such as:

  - C
  - C++
  - Assembly
  - Latex

  Is useful, because it eliminates many of the syntax errors while writing the code.

- valgrind : a memory error detector for x86-linux

  One of the most common and worst errors while programming in languages which allow you to deal with pointers are:

  - Memory leaks and
  - Memory Access Violations.

  Valgrind is a memory leak detection tool, which does just that. It is a freely available tool licensed under the GNU/GLP. However, it works only on x86-linux.

- scons : a software construction tool

  scons is a tool which allows you to create builds for large projects in a very short amoyunt of time. It allows you to handle complexity in a very simple and elegant manner. *Scons* is a tool which provides functionality similar to the GNU build tools such as:

  - Autoconf
  - Automake
  - make

Scons is written is *Python*, and allows you to write *Python* code in the build script.

- LaTeX: structured text formatting and type-setting tool

  LaTeX is a tool which allows you to create professional looking documents in a jiffy. In fact this document has also been written using *LaTeX*.

- doxygen : Documentation system for C/C++, etc...

  Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors) and to some extent PHP, C#, and D.

  It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

- Flex : A lexical analyser

  flex is a tool for generating scanners: programs which recognise lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, 'lex.yy.c', which defines a routine 'yylex()'. This file is compiled and linked with the '-lfl' library to produce an executable. When the executable is run, it analyses its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

  flex can be used to generate a C++ scanner class, using the '-+' option, (or, equivalently, '%option c++'), which is automatically specified if the name of the flex executable ends in a '+', such as flex++. When using this option, flex defaults to generating the scanner to the file 'lex.yy.cc' instead of 'lex.yy.c'. The generated scanner includes the header file 'FlexLexer.h', which defines the interface to two C++ classes.

- Bison : A parser generator

  Choosing the tools for creating an interpreter was not simple. Lex and yacc are the traditional tools in the Unix environment, however, they are primarily 'C' language tools, and would not blend very well into the object-oriented nature of development that we intended to pursue. The other choices that we had were:

  - Lemon
  - ANTLR
  - BisonCPP

- Lemon

  Lemon itself has been used in SQLite, an embedded database engine. The lemon parser too is C based, but it is functionally different from the Yacc parser. For one, in most conventional parsers, it is the parser that calls the lexer for tokens. However, in lemon, it is the lexer that calls the parser. Also, the parser generated by lemon is thread-safe, i.e. it can be simply plugged into a multithreaded program without any problems. However, the grammar specification for a lemon differs from that of yacc.

- ANTLR

  ANTLR parser is a Java-based parser generator. Due to the lack of Java development tools on the Unix platform, this aspect weighed heavily against ANTLR. However, it is a parser with an n-level look-ahead [16]. We understand this to be an advantage, but our knowledge of compilers is presently too limited to judge whether this makes it a better choice.

- BisonCPP

  BisonCPP is a port of the bison[17] parser generator to C++. This seems to be the only possible advantage of BisonCPP. Unless we have very serious issues with Yacc, there seems to be no compelling reason to use it.

- Our choice

  We decided to go ahead with a flex + bison combination, which are the GNU versions of lex and yacc respectively. The decision factors that weighed in favourably in favour of these were:

  - Easy availability on modern distributions of GNU/Linux.
  - Plethora of documentation on line.
  - Well-tested, and have been stable for years.
  - Other successful databases like PostgreSQL and MySQL use the same tools.

[16] The number of tokens the parser reads without pushing them onto the stack

[17] GNU version of Yacc

  - Success in making the parser generated by these tools thread-safe[18].

- CVS - Concurrent Versions System

  CVS is a version control system, an important component of Source Configuration Management (SCM). Using it, you can record the history of sources files, and documents. It fills a similar role to the free software RCS, PRCS, and Aegis packages.

  CVS is a production quality system in wide use around the world, including many free software projects.

  While CVS stores individual file history in the same format as RCS, it offers the following significant advantages over RCS:

  - It can run scripts which you can supply to log CVS operations or enforce site-specific polices.
  - Client/server CVS enables developers scattered by geography or slow modems to function as a single team. The version history is stored on a single central server and the client machines have a copy of all the files that the developers are working on. Therefore, the network between the client and the server must be up to perform CVS operations (such as checkins or updates) but need not be up to edit or manipulate the current versions of the files. Clients can perform all the same operations which are available locally.
  - In cases where several developers or teams want to each maintain their own version of the files, because of geography and/or policy, CVS's vendor branches can import a version from another team (even if they don't use CVS), and then CVS can merge the changes from the vendor branch with the latest files if that is what is desired.
  - Unreserved checkouts, allowing more than one developer to work on the same files at the same time.
  - CVS provides a flexible modules database that provides a symbolic mapping of names to components of a larger software distribution. It applies names to collections of directories and files. A single command can manipulate the entire collection.
  - CVS servers run on most unix variants, and clients for Windows NT/95, OS/2 and VMS are also available. CVS will also operate in what is sometimes called server mode against local repositories on Windows 95/NT.

[18]This property was very much desired, both by us and the project in-charge. Details on how this was achieved will follow later.

6.2.3 Special Resources

The operating enviroment for both the Design & Build and the Execution phase is A GNU/Linux based enviroment running on x86 or x86-64 systems.

7 Testing

7.1 Test Plan

We indend using a variety of test methods for making sure that the correcness of the final product is not compromized in any way, and at the same time making sure that the efficiency is maintained at a decent enough and workable level. The various testing methods we indend using are listed below, with a broef description of each one followed by how we will be using it specifically in this project.

- *White Box Testing:*

  Also called as glass-box testing, is a design method that uses the control strusture of the procedural design to derive test cases. Using white box testing methods, there are test cases that

  1. All independent paths within a module have been exercised at least once.
  2. All logical decisions are exercised on their true and false sides.
  3. All loops are executed at their boundaries and within their operational bounds.
  4. Internal data structures are exercised to ensure their validity.

- *Basis Path Testing:*

  Basis path testing is one such white box testing which involves the following parts

  1. *Flow graph* - Flow graph depicts logical control flow using the graph notations. In a flow graph each node represents one or more procedural statements. The arrows on the flow graph, called edges or links, represents flow of control and are anologous to flowchart arrows.
  2. *Cyclomatic Complexity* - Cyclomatic complexity is a software metric that provides a quantative measures of the logical complexity of a program. The value computed here defines the number of independent

paths in the basis set. It provides with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed atleast once.

3. *Deriving Test Cases* - Following steps are applied to derive the basis set.
   (a) Using the design or code as a foundation, draw a corresponding flow graph.
   (b) Determine cyclomatic complexity of resultant flow grpah.
   (c) Determine a basis set of linearly independent paths.
   (d) Prepare test cases that will force execution of each path in the basis set.
4. *Graph Matrices* - A graph matrix is a software tool that assists in basis path testing. A grpah matrix is a square matrix whose size is equal to the number if nodes on the flow graph. Graph matrix can become a powerful tool for evaluating program control structure during testing.

Speaking in terms of a software engineer, white box testing is a good way to detect the software defects. But since the shortage of time faced as a student, we have not used white box testing for testing TDDB.

- *Black Box Testing:*

  Black-box testing, also called behavioural testing, focuses on the functional requirements of the software developed. Black-box testing enables to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing ia not a alternative to white-box techniques. Rather a complementary approach that is likely to uncover a different class of errors than white-box methods.

  Black-box testing attempts to find errors in the following categories:

  1. incorrect or missing functions,
  2. interface errors,
  3. errors in data structures or external data base access,
  4. behavior or performance errors, and
  5. initialization and termination errors.

  Black-box testing ia applied during later stages of testing. Different steps in back-box testing are :

1. *Graph-Based Testing Methods:*

   Testing here is to understand and test the objects and relationship that connect these objects. Software testing begins with creation of a graph of important objects and their relationship and devising a series of tests that will cover the graph so that object and relationship is exercised and errors are uncovered.

2. *Equivalence Partitioning:*

   Equivalence Partitioning is a Black-box testing method that divides the input domain of a program into classes of data from which test can be derived. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed. Test cases design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.

3. *Boundary Value Analysis:*

   The reason behind Boundary Value Analysis to be a popular testing technique is that a greater number if error tend to occur at the boundaries of the input domain rather than the center. Boundary Value Analysis leads to a selection of test cases that exercise bounding values

4. *Comparison Testing:*

   When redundant software is developed, seperate software engineering teams develop independent versions of an application using the same specification. Each version can be tested with the same test data to ensure that all provide identical output. All versions are executed in parallel with real-time comparision of results to ensure consistency. These independent versions form the basis ol a black-box testing technique called comparison testing or back-to-back testing.

5. *Orthogonal Array Testing:*

   Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding errors associated with region faults - an error category associated with faulty logic within a software component.

- *Unit Testing:*

  Unit testing focuses verification effort on the smallest unit of software design, the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors

within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity duringall steps in an algorithm‘s execution. Boundary conditons are tested to ensure that module operates properly at boundary established to limit or restrict processing. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Finally, all error handling paths are tested.

Among the more common errors in computation are

1. misunderstood or incorrect arithmetic precedence,
2. mixed mode operations,
3. incorrect initialization,
4. precision inaccuracy,
5. incorrect representation of an expression

Test cases should uncover errors such as

1. comparision of different data types,
2. incorrect locigal operators or precedence,
3. expectation of equality when precision error makes equality unlikely,
4. incorrect comparision of variables,
5. improper or nonexistent loop termination,
6. failure to exit when divergent iteration is encountered, and
7. improperly modified loop variables

Among the potential errors that should be tested when error handling is evaluated are

1. Error description is unintelligible.
2. Error noted does not correspod to error encountered.
3. Error condition causes system intervention prior to error handling.

4. Exception condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.

Unit testing is normally considered as an adjunct to the coding step. After source level code has ben developed, reviewed, and verified for correspondence to component level design, unit test cases that are likely to uncover errors in each of the categories. Each test case should be coupled with a set of expected results

A driver or a stub software has to be developed for each unit test. in most applications a driver is nothing more than a ”main program” that accepts data, passes such data to the component and prints relevant results. Stubs serve to replace modules that subordinate to the component to be tested. Drivers and stubs represents overhead. That is, both are software that must be written but that is not deliveres with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low.

- *Integration Testing:*

  After each unit is tested individually, we move to integration testing. Even after every unit is tested, they are combined stepwise n checked for errors. This is known as integration testing. There are many errors that come up when the units are interfaced together. This errors are hardly known when each unit is tested. The errors may be realted to the data, between the modules, and even the desired output may not be produced.

  Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and built a program strucutre that is required as per the design.

  There is often a tendency to attempt nonincremental integration, that is to construct the program using a ”big bang” approach. All components are combined and the entire program is tested as a whole. This always results in set of errors. Correction of errors is very difficult. Once the errors are corrected new ones appear. This takes a lot of time for the program to be error free.

  To avoid the problems faced in the big bang approach, incremental integration is used. The program is constructed and tested in small increments, where errors can be isolated and corrected, interfaces are likely to be tested completely.

  Following are different integration stratergies

– *Top-down Integration:*

Modules are integrated by moving downward through the control hierarchy, beginning with the main control module, main program. Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Depth-first integration would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.

The integration process is performed in a series of five steps

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate for all components directly subordinate to the main control module.
2. Depending on the integration approach selected subordinate stubs are replaced one ata atime with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real components.
5. Regression testing may be conducted to ensure that new errors have not been introduced. The process continues from step 2 until the entire program structure is built.

Top-down stratergy sounds relatively uncomplicated, but in practice, logistical problems can arise. The tester is left with three choices :

1. delay many tests until stubs are replaced with actual modules,
2. develop stubs that perform limited functions that stimulate the actual module, or
3. integrate the software from the bottom of the hierarchy upwards

The first approach causes us to loose some control over correspondence between specific tests and incorporates of specific modules. The second approach is workable but not feasible as it leads to significant overhead. The third approach, called as bottom-up testing is discussed as next stratergy.

– *Bottom-up Integration:*

Bottom-up integration testing, as it name implies, begins construction and testing with atomic modules(components at the lowest level in the program structure). Because components are integrated from the

bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters that perfiorm a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upwards in the program structure.

As integration moves upward, the need for seperate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

- *Regression Testing:*

  Each time a new module is added as a part of integration testing, the software changes. These changes are reflected by changes such that new data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test startergy, regression testing is the re-execution of some subset of tests.

  Successful tests result in the discovery of errors, and errors are corrected. Whenever software is corrected, some aspect of software configuration is changed . Regression testing is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors. Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

  The regression test suite contains three different classes of test cases:

  1. A representative sample of tests that will exercise all software functions.
  2. Additional tests that focus on software functions that are likely to be affected by the change.
  3. Test that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. It is impractical and inefficient to re-execute every test for every program function once a change has occured

- *System Testing:*

  Software is incorporated with other system elements, and a series of system integration and validation tests are conducted. Such tests related of execution of the software with respect to the system it is being executed are known as system testing.

  The potential problems faced are :

  1. design error handling paths thata test all information coming from other elements of the system,
  2. conduct a series of tests that stimulate bad data or other potential errors at the software interface,
  3. record the results of tests to use as evidence if finger-pointing does occur, and
  4. participate in planning and design of system tests to ensure that software is adequately tested.

  System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

  Following we will discuss the types of system tests that are worthwhile for software based systems.

  – *Recovery Testing:*

    Many computer based system must recover from faults and resume processing within a prescribed time. A system must be fault tolerant, that is, processing faults must not cause overall system function to cease. In other cases, a system failure must not be corrected within a specified period of time or severe economic damage will occur.

    Recovery testing is a system test that forces software to fail in a variety of ways and verifies that recovery is properly performed. Recovery can be automatic, by the system itself or by human intervention. If recovery is automatic, the system, reinitialization, checkpointing mechanisms, data recovery and restart are evaluated for correctness. If it requires human intervention, MTTR is evaluated to determine whether it is within acceptable limits.

– *Security Testing:*

Security testing attempts to verify that protection mechanisms built into system will, in fact, protect it from improper penetration. The system‘s security must,of course be tested for invulnerability from frontal attack but must also be tested for invulnerability from flank or rear attack.

During security testing, the tester plays the role of the individual who desire to penetrate the system. The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying services to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of information that will be obtained.

– *Stress Testing:*

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

1. Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
2. Input data rates may be increased by an order of magnitude to determane how input functions will respond.
3. Test cases thet require maximum memory or other resources are executed.
4. Test cases that may cause thrashing in a virtual operating system are designed.
5. Test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called sensitivity testing. In some situation, a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroenous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

– *Performance Testing:*

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unnacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in testing process. Even at the unit level, the performance of an individual module may be assessed. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure reource utilization in an exacting fashion. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

7.2 Test Cases & Methods Used

We have used a judicious use of the testing techniques described above for testing the working correctness & efficiency aspects of TDDB.

Black Box Testing & Unit Testing were used to test each of the individual components which can operate in isolation. These include:

- Lock Manager
- Buffer Cache
- File Interface
- Bitmap Lock
- Dirty Map
- Table Lock
- MT-Allocator
- MD5 Encryption
- External Parallel Merge Sort
- Threading Interface
- Socket Interface
- Primitive Data Types

- Disk Debugger

For the other modules which depended on the other modules to be working, we used *Integration Testing* where by we had assemble the whole system, and then begin testing, because of coupling between the individual modules. These include:

- SQL Parser
- Console Client
- Server
- SQL Execution Engine

Regression Testing helped us discover bugs which were introduced into the code tree because of the addition of a new feature, or because of some other bug-fix.

8 Maintenance

8.1 Installation

This is how you can install & run TDDB for the first time.

TDDB is broken up into a Server & a Client application. The following steps need to be carried out in order to compile & run TDDB:

1. Compile TDDB: Run 'scons opt=1 release=1' from within the tddb/ directory.
2. Set up the initial DB file. This must be at least 256MB in size. This can be done using 'dd' as such:
   'dd if=/dev/zero of=db_stub.dat bs=1048576 count=400'
   This will create a DB file of size 400MB. You can also use any RAW disk space or partition for the DB file.
3. Format the DB file for initial use:
   './src/client/server –db-file-name=db_stub.dat –create-db'
   Will format the file for subsequent use.

4. Start the TDDB server:
   ’./src/client/server –db-file-name=db_stub.dat’
   This will start the TDDB server, and will bring you to the ADMIN prompt where you can type ’help’ to get the list of commands supported.

5. Help on the server can be obtained using:
   ’./src/client/server –help’

6. Start the console based client:
   ’./src/client/console_client’
   As usual add a ’–help’ for help on the console client.

7. Congratulations! If you got to this point, and the console client connected to the TDDB server, then you can start executing SQL queries.

9 Conclusion & Future Scope

This project has been a great learning experience for us, and we have gotten to know many practical design & implementation level aspects of real systems. The experience co-corination & working in groups with people located at geographically distant locations has been something new, exciting & enjoyable for us. Needless to say, we have learnt a *lot* from it.

As far as the future scope of this project, there is lots that can be done. There are many features & optimizations that can be implemented. To mention some:

- Nested Query Support
- Support for non-field projections
- Support for *Aggregate* function in SQL
- Support for Indexes
- Query Optimization
- Support for Query Time-Out
- Process management administrative functions
- Process priority admistration
- Support for Users
- Security features such as database/table access rights on a per user basis.

As with any project, there are some limitations that *TDDB* has. Here are some major ones:

- The INTEGER/INT data type doesn't support negative numbers. Yes, only non-negative numbers can be used. Furthermore, decrementing an INT beyond 0 results in the INT getting the value NULL.
- You can view the list of tables, and the commands used to create them by selecting from the table 'global.tabtab' as: 'SELECT * FROM global.tabtab;'. This command is treated as a DDL statement for compatibility reasons.
- DDL Commands are also Transaction oriented!! Suppose you CREATE or DROP a Table by mistake, you can revert your action by doing a ROLLBACK. However, if you want to make the changes permanant, use COMMIT.
- You can't Mix DML & DDL commands in a single Transaction. Suppose you CREATE a table, you need to COMMIT or ROLLBACK before you issue a DML Command. You may however, issue another DDL Command in the same Transaction such as creating or dropping another table. Don't try to CREATE & DROP the same table in the same Transaction though.
- Maximum Row Size = 8100 Bytes. That is the Sum of all data types in the Row should be not greater than 8100 Bytes. The sizes of the various data types for this calculation are mentioned in Table 2 below:

Data Type	Size In Bytes
INTEGER	8
CHAR(N)	N + 4
VARCHAR(N)	32
CLOB	16

Table 2: Various data types and their sizes on disk

- There is currently no support for indices, and so each query processed needs to do a full table scan. Even JOINs are performed using the *nested loop join* algorithm. We are working on getting more efficient JOIN algorithms which don't need Indices or create the indices dynamically in place, so that JOIN processing can be made faster.
- There is currently no support for nested queries.

- The type checking of the parser is quite weak. That needs to be fixed. For example, certain type conversions are performed silently by the engine without the user ever being notified about them. If the user passes a string in case of an integer, then the string is parsed as an int, and the first few digit characters(if any) are parsed as the integer. For the reverse case, the integer parsed as the string is entered as a string with the representation of the integer.

List of Figures

List of Tables

References

[PatXX] Some Considerations of Locking to Prevent Phantoms, Patrick O'Neil, Alan Fekete, Elizabeth O'Neil1, and Dimitrios Liarokapis

[Tob94] The Design and Performance Evaluation of a Lock Manager for a memory-Resident Database System, Tobin. J. Lehman, Vibby Gottemukkala, January 14, 1994

[Don81] A History and Evaluation of System R, Donald D. Chamberlin, Thomas G. Price, Morton M. Astrahan, Franco Putzolu, Michael W. Blasgen, Patricia Griffiths Selinger, James N. Gray, Mario Schkolnick, W. Frank King, Donald R. Slutz, Bruce G. Lindsay, Irving L. Traiger, Raymond Lorie, Bradford W. Wade, James W. Mehl, Robert A. Yost, October 1981

[Sto80] Retrospection on a Database System, Michael Stonebraker, June 1980

[Tan87] Non-Stop SQL, A Distributed High-performance, High-availbility Implementation of SQL, Tandem Database Group, April 1987

[And84] Robustness to crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach, Andrea Borr, Tandem Database group, September 1984

[Avi96] Strategic Directions in Database Systems–Breaking Out of the Box, Avi Silberschatz, Stan Zdonil et al., December 1996

[Avi95] Database Research: Achievements and Opportunities Into the 21st Century, Avi Silberschatz, Michael Stonebraker, Jeff Ullman, editors, May 1995

[Sto91] The Postgres Next-Generation Database Management System, Michael Stonebraker, Greg Kemnitz, October 1991

[Dav92] Parallel Database Systems: The Future of High Performance Database Processing, David J. DeWitt, Jim Gray, January 1992

[Hon85] An Evaluation of Buffer Management Strategies for Relational Database Systems, Hong-Tai Chou, David J. Dewitt

[Sto87] The Design of the Postgres Storage System, Michael Stonebraker

[Jim96] The Dangers of Replication and a Solution, Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha

[DimXX] On Serializability of Multidatabase Transactions Through Forced Local Conflicts, Dimitrios Georgakopoulos and Marek Rusinkiewicz, Amit Sheth

[Knu98] Donald Knuth, The Art of Computer Programming Volume-3, 2^{nd} Edition, Pearson Education, 2004

[Hec04] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, Database Systems: The Complete Book, 1^{st} Edition, Pearson Education, 2004

[Ram04] Ramez Elmasri, Shamkant B. Navathe, Fundamentals of Database Systems, 4^{th} Edition, Pearson Education, 2004

[Sil02] Silberschatz, Korth, Sudarshan, Database System Concepts, 4^{th} Edition, Mc Graw Hill, 2002