The Guide and Specification of Fatworm Project

(the third version)

Department of Computer Science and Engineering,
Shanghai Jiao Tong University,
Shanghai 200240, P. R. China

Designed By

02-ACMer:	Li Lei { lilei@cs.cmu.edu }
	Yang Linji {dragonylj@sjtu.edu.cn}
	Zhang Yaodong {zhydong@sjtu.edu.cn}
03-ACMer:	Ma Rong {marong1204@gmail.com }
	Zhou Muxin {muxin.zhou@gmail.com }
	Qian Feng { fengqian@umich.edu }
	Qu Wentao { thomas.qu@gamil.com }
04-ACMer:	Li Mu {limu.cn@gmail.com }
	Lian Xiaochen {lianxiaochen@gmail.com}
	Sun Xiaorui {sunsirius@hotmail.com }

March 9, 2008

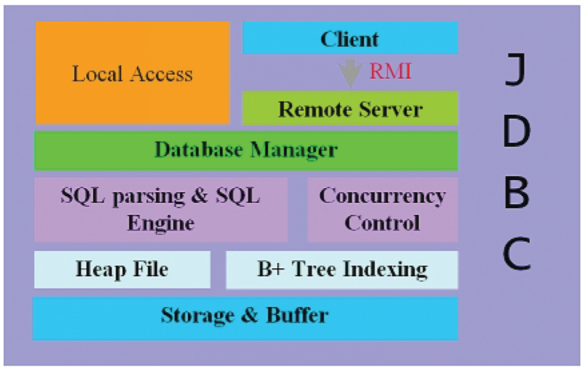
1 Project Overview

The project for this course is to build a database management system(DBMS), which would be the best way for you to understand the basic concepts of DBMS. As we know, a database system for business use would be extremely large and hard to implement, therefore, we give some restrictions and specifications to help you get started. However, any additions or improvement are encouraged, and certainly will make you receive a higher score. Although we do not carry out a whole featured DBMS, the project is still too large for only one student's work. Due to this reason, we will do teamwork. Each team should contain 3 or 2 students, formed by free will. Finally, some suggestions for doing well in this course project:

1. Read the specifications carefully until you pretty much understand it.
2. Do not code until you understand what you are doing. *Design* is the first and most important thing you should consider.
3. If you get stuck, feel free to talk to others or TAs.

2 Architecture

In this section, we give a brief impression of the entire system. Java Database Connectivity(JDBC) API is a standard SQL database access interface. This API provides programmers with universal access to a wide range of database management systems, including our Fatworm. In this way, the whole system should be built on the JDBC standards, which is illustrated in Figure 1.



The entire system can be divided into five layers: storage, indexing, SQL parsing and engine, database manager, access interfaces. Also, there are some assistant components in these layers, such as buffer, heap file, concurrency control and so on. The following sections will give you a detailed description on these layers.

3 Storage Layer

3.1 Overview

The storage part of Fatworm is implemented as a single file on hard disk. Similar to the virtual memory in operating systems, we divide the single file with pages, which can be recognized as the simply page-sized blocks of bytes within this single file. The higher-level structures, such as B+ tree index, do not know how their data are recorded on the hard disk but only know a page id as the entry point of accessing their data. Due to the fixed size of a page, some operations would require a collection of pages. How to load the needed pages or do buffering is the core problem that the buffer part should focus on. In summary, the storage layer should take care of the allocation and deallocation of pages within a database, which means an allocation strategy is required, such as using bitmap. It also performs reads and writes of pages to and from hard disk, and provides a logical and friendly interface of reading or writing data within the context of a database management system. Bitmap can be used to decide which page should be allocated or deallocated. There are various ways of designing a bitmap strategy. Here, we give an example: suppose eight bits are a group in the bitmap table, and there are a lot of groups in the table. When we read $(10000000)_2$ from a group 3 in bitmap, it means that except page $(3-1)\times 8+1$, the pages from $(3-1)\times 8+2$ to $(3-1)\times 8+8$ are free.

3.2 Functions

The storage layer should provide the following functions:

- create or delete a database
- open or close a database
- page allocation strategy
- buffer management

3.3 Buffer Strategy

Here is an example of buffer strategy. Suppose we construct a single buffer pool for all of the databases which has already opened. In this buffer pool, a globe hash table is used to store all the key-value pairs. Since all opened databases share one buffer pool, we use database id(DID) and page id(PID) as their key and use the whole page content as the value of the hash table. When the system starts, there are 1024 entries in this hash table. For this reason, when a database contains more than 1024 pages or more than one databases are opened at the same time, the buffer should use LFU algorithm to replace the unused page.

Buffer strategy may also support the pre-fetch technology. Requiring a page but in the buffer pool will lead to a page fault. When a page fault is occurred, the system need to load the page from disk into the buffer pool. During this procedure, the buffer-management not only loads the required page into buffer pool but also loads pages around it because of the locality assumption of the data.

4 Indexing Layer

4.1 Overview

Heap file structure is one of the common components in a database system, which provides the ability of sequential access of the records, as well as insertion, deletion, updating and etc basic features. Index provides fast access to the specific record when given a certain condition. In the implementation, please adopt the B+ Tree indexing. The index entry is formed as $\langle key, rid \rangle$. Key can be integer, string, date, time or float. Indexing should have five functions:

- create an index
- drop an index
- insert an entry by an index
- delete an entry by an index
- search though an index

The most important functions are insert, delete and search, among which delete is the most complex because you have to keep the tree balance after delete an entry.

4.2 Functions

We may adopt the linked directory page method to implement the heap file, which is widely used in Operation System such as Microsoft Windows. Suppose each directory page has 8K bytes space, the first 4 bytes of which are intentionally preserved to the next page pointer, and other bytes of which are all used to store the ID of content pages. The content page has a head area, which indicates the necessary information of this page, such as the length of each slot, the length of head and etc, and a body area, which stores the data of each slot, that is the data of each record. The B+ Tree is a balanced tree and its nodes are BTPages. For instance, in one specific design: there are three types of BTPages: BTHeaderPage, BTIndexPage and BTLeafPage. In BTHeaderPage, some important information about the B+ tree is recorded in the BTHeaderPage, such as the root page id of the tree and the key type. BTIndexPage is the internal node and only contains index to the child B+ tree page. BTLeafPage is the leaf node and contain index entries with rids. A typical B+ tree provides three operations: searching, inserting and deleting[1], see [1] for detail.

5 SQL Parsing and Engine Layer

5.1 Overview

In this part of design, you are first asked to implement a highly efficient and accurate fatworm SQL grammar parser as well as an algebra tree transformer and a simple optimizer (which is optional). It is a convenience that lexical

[1]deleting a B+ tree node with balance is not a basic required feature.

analysts such as JLEX or JFLEX should be used to build up a abstract grammar tree(AGT), and in succession an algebra tree transformer applied to the AGT will finish the stage of parser.

Secondly, the Engine's responsibility is to execute the SQL statement. Given a SQL statement, the Engine first parses it with SQL parser and then according to the parsed result, execute either query or update. Engine interacts with the indexing layer and parsing layer. You should design the interfaces of indexing layer carefully to make your engine called easily.

5.2 SQL Parser

It is NOT required to make your parser parse everything and reject wrong grammar. You can write a parser that can parse correctly when the input query is correct from the perspective of syntax. You can always assume that the input query is correct, because this is not a project of compiler. Your parser should support the SQL statements in the appendix at least.

5.3 Engine

The design of your SQL Engine is left BLANK here and should be completed all by yourselves. It is wise to transform the grammar tree to an algebra tree(for the definition of algebra tree, please refer to GOOGLE), and then transform the algebra tree to an execution tree (one may wish transform the grammar tree to execution tree directly, it's also OK). However, here are some design pattern you may consider to apply(see [2]):

- **Composite Pattern** to present a tree structure.
  For example, the boolean expression of the where-condition may be considered as a tree structure naturally. You may define an abstract class WhereNode which is the common interface and implements the default behavior. And certainly you should define the other composite class such as AndWhereNode(to present an binary conjunction operation), NotWhereNode(to present a negation operation) to implement child-related operations as their names. Finally, you may define some leaf class to implement the primitive behavior such as comparison to a literal, arithmetic operation and so on.

- **Visitor Pattern** to represent different operations to be performed on the elements of a fixed structure.
  As the example shown above, you may realize there are several visit operation will be performed on the WhereNode-tree for different propose. In most cases, the visitor traverses the tree in the propose of evaluating the condition, however, a SQL-optimizer may traverses the WhereNode-tree to reorder the execution plan. So you may define these different visitor class such as ExecutionVisitor, OptimizationVisitor, PrettyPrintVisitor or TypeCheckingVisitor to gather related traversal operations and separates unrelated ones.

- **Iterator Pattern** to provide a way to access the element of the heap file(or index) without exposing its underlying representation.
  For instance, to traverse a single table, we may choose SingleIterator, to

traverse a table which is jointed by other tables, we may choose JointIterator, to traverse a B+ tree index, we may choose BTreeIterator. These iterators' behavior are hidden to Engine, Engine only uses the interface—RowIterator to access the intended sequential file abstractly.

WARNING: The proper design will help you code clear and bug-free, but an over design will bring your team to the HELL!

5.3.1 Query Processing

Once the query plan is chosen, the Engine evaluated with that plan, and the result of the query can be seen as a table (or a relation). Generally speaking, you may process simple selection operations by performing a linear scan, by doing a binary search, or by making use of index. Further more, you may handle complex selections by computing unions and intersections of the results of simple selection. Some issues are listed as following(you may refer to [3] for details).

- **Product(Natural Join).** If a table with m rows products with another with n rows, the result is the Cartesian product of the two tables with $m \times n$ rows.
- **Sorting(Order By).** Sort the tuples in the table. You may sort the table larger than memory by the external sort-merge algorithm.
- **Duplicate Elimination.** This operator can be done by sorting or hashing easily.
- **Projection.** You may implement projection easily by performing projection on each tuple, which gives a relation that could have duplicate records, and then removing duplicate records.
- **Aggregation.** One way to handle the aggregation operation can be implemented in the same way as duplicate elimination. you may sort the relation based on the grouping attributes, then gather the tuples with the same value, and apply the aggregate operation on each group to get the result. On the other hand, instead of gathering all the tuples in a group and then applying the operation, we can implement some operations(such as sum, max, min, count and avg) on the fly as the groups are being constructed. see [3] for details.

5.3.2 Insert, Delete and Update

These three kind of statement will not cost much time to code if you followed a good design of Engine. We strongly recommended you to study some example like [4]. A mature programmer should spend 80% time to think and 20% time to code.

5.4 Concurrency Control

You should carefully designed a concurrency control unit which may includes a concurrency control manager, a session and a table lock. A session is given to each user who has successfully entered the database system and all of the operations of the user are related to his session. Different user can open the same database in the same time and they also can read from the same table concurrently. The only constraint is that they cannot access the same table when someone do writing operations on it. So you can set up a table lock to solve this problem. [2]

Table lock is the kind of lock which support two kinds of locking operation, read lock and write lock. For example, when a table has a write lock, no operations are allowed on it. When a table has a read lock, only read requirements are allowed. If a user want to do some writing operations on it, he must wait until the previous read lock is released.

When user requires a transaction,[3] at this time all the table involved in this transaction must be locked. Unfortunately, a dead lock may be occurred. You can use the simplest method to avoid this problem. For instance, there are three tables A, B and C which are required to be locked. If we get the locks of A and B but table C already has a write lock, we should wait for the table C. When a time interval(maybe 100ms) is passed, we release the locks of A and B, which means the first trying of locking is failed and we have to wait for another interval(50ms) to try again.

Certainly, you are encouraged to use a more reliable and efficient algorithm to avoid dead locking.

6 Database Manager Layer

6.1 Overview

The database manager is the abstraction of the whole system. It is the bridge between access interface and the Engine. Since we allow open more than one database at the same time, the database manager should also maintain all opened databases in current system. For example, when a require comes from the out connections, the database manager holds the session and send the require to the intended database.

There will be some meta-tables constructed at the initialization of a database, because the information such as the table description, the index description and the relation description should be recorded. Only the database manager has the right to modify these meta-tables, when the users ask for creating or dropping a table.

Certainly, the database manager will package the result from the low-level system, and send it to the right session.

In addition, the database manager may also include the logger, the user manager and configuration manager. Logger[4] records all the operation step by step, so that we can identify the user's behavior or do reconstruction when crash happens. The user manager manages all the verified users for each database, and the restrictions they have. Since there will be many parameters to control the behavior and the thresholds for tuning, we may use a configuration manager to handle these parameters and thresholds at the birth of the whole DBMS.

[2]Lock on table is a basic required feature, while lock on page is considered as an advanced feature.

[3]Transaction is not a basic required feature.

[4]Logger and rollback is not a basic required feature.

6.2 Functions

- hold the connected sessions.
- manage any database in current system.
- package the result from Engine and produce response to interface.

7 Access Interface Layer

to be continue.

8 Grading Policy

See the course website for detail.

Feature Requirement: Summary

Part I. Required

- Data types: INT, FLOAT, CHAR(), DATETIME [5]
- Components:
  - Storage
    - Buffering
  - Heap File
  - B+ Tree
    - Without balanced deleting
  - SQL Parser and Engine
    - Create / Drop databases
    - Create / Drop tables (without AUTO_INCREMENT column)
    - Insert / Delete / Update tuples in table
    - Basic SELECT on single or multiple table(s) [6]
  - Database manager
    - More than one database in system
    - Local-access drivers (JDBC)

[5] Make use of java.util.Date, here are some details: http://dev.mysql.com/doc/refman/5.1/en/storage-requirements.html

[6] Features that are not clearly mentioned in recommended and advanced section are all required.

Part II. Recommended

- Data types: BOOLEAN, TIMESTAMP, DECIMAL()
- Components:
  - B+ Tree
    - With balanced deleting
    - Without duplicated key support [7]
  - SQL Parser and Engine
    - Create / Drop tables (with AUTO_INCREMENT column)
    - Create / Drop indexes
    - More powerful SELECT
      - DISTINCT
      - Alias of tables
      - Subquery as Scalar Operand
      - [NOT] EXISTS
      - ORDER BY can accept more than one column as arguments
  - Database manager
    - Remote-access driver (JDBC)
    - Concurrency control (locks on table)
    - Deadlock solved

Part III. Advanced

- Data types: VARCHAR(), BLOB
- Components:
  - B+ Tree
    - With duplicated key support
  - SQL Parser and Engine
    - Even more powerful SELECT
      - GROUP BY / HAVING subclauses
      - Subquery as Relation Operand (ANY, ALL, IN)
    - Query Optimization
  - Support transaction
  - Logger and roll back
  - Database manager
    - Concurrency control (locks on pages)

[7] To simplify B+ tree, duplicated key support is moved to advanced section. But a unique index (just like primary key though without the title) is required.

Some Clarification

1. It's free for your team to decide which programming language to use, but the test case is totally based on the JDBC interface, so we strongly recommended Java as your programming language.
2. You should code very thing by yourself. However, taking advantage of some code-generation tool like JFLEX or JavaCup is allowed and encouraged.
3. It's NOT required to do arithmetic operation on DECIMAL column. In addition, you can set arbitrary restrict on CHAR(), DECIMAL() and VARCHAR(), as long as it's reasonable. For example, the length of VARCHAR can't exceed 4000. You should also write these restrict in document.
4. You are allowed to keep metadata of table / database (or something like that) in memory (outside buffer), since they are not very big. What you have to load into buffer are records (in heapfile) and B+ trees.

Appendix: Syntax of FwSQL

Note: the syntax of FwSQL (Fatworm-SQL) is a simplified version of MySQL 5.1. You can go to http://dev.mysql.com/doc/refman/5.1/en/ for more details.

1. Create or Drop Database

   `CREATE DATABASE` *db-name*
   `DROP DATABASE` *db-name*

2. Create or Drop table

   `CREATE TABLE` *tbl-name* (*create-definition*, ...)
   `DROP TABLE` *tbl-name* [, *tbl-name*]
   create-definition ::= *column-definition* | `PRIMARY KEY` (*col-name*)
   column-definition ::= *col-name data-type* [[`NOT`] `NULL`] [`DEFAULT` *default-value*] [`AUTO_INCREMENT`]
   date-type ::= `INT` | `FLOAT` | `CHAR(M)` | `DATETIME` | `BOOLEAN` | `DECIMAL(M[,D])` | `TIMESTAMP` | `VARCHAR(M)` | `BLOB`

3. Insert

   `INSERT INTO` *tbl-name* `VALUES` (*value*, ...)

4. Delete

   `DELETE FROM` *tbl-name* [`WHERE` *where-condition*]

5. Update

   `UPDATE` *tbl-name* `SET` *col-name1*=*value* [, *col-name2*=*value* ...] [`WHERE` *where-condition*]

6. Create or Drop index

```
CREATE [UNIQUE] INDEX index-name ON tbl-name (col-name)
DROP INDEX index-name ON tbl-name
```

7. Select

```
SELECT [DISTINCT] select-expr, ...
        [FROM tbl-ref [, tbl-ref] ... ]
        [WHERE where-condition]
        [GROUP BY col-name]
        [HAVING having-condition]
        [ORDER BY col-name [ASC | DESC], ...]
select-expr      ::= value | func(col-name) | *
func             ::= AVG | COUNT | MIN | MAX | SUM
col-name         ::= [tbl-name.] col-name
tbl-ref          ::= tbl-name [AS alias]
where-condition  ::= bool-expr
bool-expr        ::= value cop value
                   | bool-expr AND bool-expr
                   | bool-expr OR bool-expr
                   | [NOT] EXISTS (subquery)
                   | value cop ANY (subquery)
                   | value IN (subquery)
                   | value cop ALL (subquery)
cop              ::= < | > | = | <= | >= | <>
value            ::= value | col-name | const-value | value aop value | (subquery)
aop              ::= + | − | ∗ | / | %
const-value      ::= integer | 'string' | float | 'YYYY-MM-DD HH:MM:SS'
having-condition ::= having-value cop having-value
                   | having-condition AND having-condition
                   | having-condition OR having-condition
                   | bool-expr
having-value     ::= value | func(col-name)
```

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, The MIT Press, 2001.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Pearson Education, 1995.

[3] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Database System Concepts, the McGraw-Hill Companies, 2006.

[4] Source code of HSQLDB, see http://hsqldb.sourceforge.net.