

FatWorm Project: Database Management System

Final Project for ~~03 ACM~~ Database Course

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

Designed By

Zhang Yaodong {zhydong@sjtu.edu.cn}

Li Lei {cosmos_l@sjtu.edu.cn}

Yang Linji {dragonflylj@sjtu.edu.cn}

8 November, 2005

这是李磊等学长以前所写的文档，其中大部分内容是可以借鉴的，我们对部分章节进行了修订，使之符合实际需要，其第八、第九、第十章因为改动很大，所以原稿作废，我们会把新的要求放在网页上这一点请大家务必注意。

Contents

1	Project Overview	3
2	Architecture	3
3	Storage Layer	4
3.1	Overview	4
3.2	Functions	4
3.3	Buffer Strategy	4
4	Indexing Layer	5
4.1	Overview	5
4.2	Functions	5
4.2.1	Search	5
4.2.2	Insert	5
4.2.3	Delete	6
5	SQL Parsing and Engine Layer	6
5.1	Overview	6
5.2	SQL	6
5.3	Engine	8
5.3.1	Query	8
5.3.2	Insert	8
5.3.3	Delete	8
5.3.4	Update	8
5.3.5	Transaction	9
5.4	Concurrency Control	9
6	Database Manager Layer	9
6.1	Overview	9
6.2	Functions	10
7	Access Interface Layer	10
7.1	Overview	10
7.2	Internal Driver	10
7.3	Remote Driver	11
7.4	Database Server	11
8	Grading Policy	11
8.1	Required-65%	11
8.2	Recommended-25%	12
8.3	Advanced-10%	13
8.4	Addition-10%	13
8.5	Summary	13
9	Schedule	13
10	Document Requirement	13

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.

Compared to other educational projects, we have special grading policy, based on functional points. There will be a detailed description in Section Grading Policy. 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 students, formed by free will.

Finally, some suggestions for doing well in this course project:

- Read the specifications carefully until you pretty much understand it.
- Do not code until you understand what you are doing. Design is the first and most important thing you should consider.
- 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.

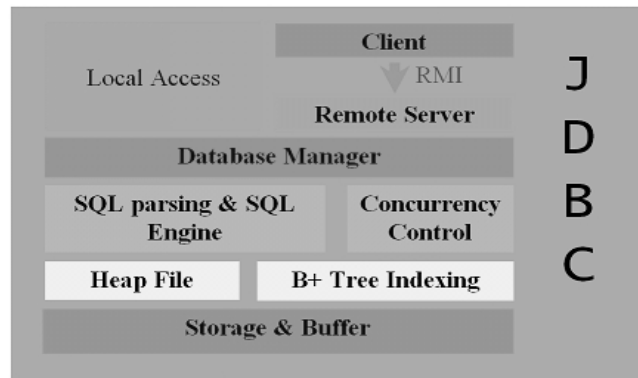


Figure 1: FatWorm Database Management System

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, configuration 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 used in FatWorm DBMS. Eight bits are a group in our bitmap table, and there are a lot of groups in the table. When we read 10000000 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
- do buffering

3.3 Buffer Strategy

Here is an example of buffer strategy in FatWorm DBMS.

We construct a single buffer pool for all of the databases which has already opened. In the 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.

Our buffer also support pre-fetch technology. When we require a page but find the page is not in the buffer pool, this will lead to a page fault. When a page fault is occurred, the system need to read the page from disk into the buffer pool. During this procedure, the buffer not only reads the required page into buffer pool but also reads pages around it because of the locality property of the data.

Another strategy which highly affects the performance of the buffer is batch-free. When we want to release a page, we only give a mark on this page instead of cleaning the content

of the page, since the cleaning needs a lot work on hard disk which will consume much time. ~~However, considering the safety of the data, the real clean is the necessary.~~ So we accumulate the released pages, when it reach a threshold, we do cleaning.

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 $\langle \text{key}, \text{rid} \rangle$. Key can be integer, string, date, time or float. Indexing has five functions: create an index, insert an entry, delete an entry, search the index and clear the 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

Please adopt the linked directory page method to implement the heap file, which is widely used in Operation System such as Microsoft Windows. 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: BTHdrPage, BTIndexPage and BTLeafPage. In BTHdrPage, some important information about the B+ tree is recorded in the BT-HeaderPage, 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 following characters:

4.2.1 Search

The Search algorithm is used to seek the rid of the given key. To search a specific key, first begin with the root and search top-down along the tree path. The algorithm is as follows: To search a key K from node N , if N is leaf, find entry with the key value K and returns the rid. If N is index page with m keys $K_1..K_m$ and $m + 1$ indic $I_0..I_m + 1$, if $K < K_1$, then search K in I_0 , else find the first j such that $K \geq K_j$, then search K in I_j .

4.2.2 Insert

Insert the given index entry $\langle \text{key}, \text{rid} \rangle$, first should find where to put the new entry and then add it. During the process, a B+ tree page may be full and should split some entries.

You must consider three scenarios when you add a record to a B+ tree. Each scenario causes a different action in the insert algorithm.

Insertion is a recursive procedure, to add an entry to a node:

- the node is leaf node, if the node still has extra space, then add the entry to the leaf node; otherwise if the node is already full, equally split the node into two leaf nodes and add the entry to one, return the newly splited node's pointer
- the node is index node, find the proper child node to add the entry, if there is any entry returned, perform insertion on the index page. If the index page still has extra space, then add the entry it, if not, split the index page, insert the entry to the proper one and return the newly split node's pointer.

4.2.3 Delete

Delete the specified index entry. The essence of the deletion is that after deleting the entry, you should keep the B+ tree balanced, which force us to merge some scarce pages.

To delete an entry in the B+ tree at the specific node:

- if the node is leaf page, then the entry is deleted from the page. If this page only uses less than half of the page space, then if the sibling of this page has extra entries, then the entries in these pages are equally reallocate and with the help of their father node; if the sibling is also scarce, then merge the two pages and delete one entry in their father node page.
- if the node is index page, then first try to delete the entry in the proper child page. if the deletion in child page involves in deletion in this page, try to delete the entry in this page and then balance the tree by reallocating and merging method similar to leaf.

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 fat worm SQL grammar parser as well as an algebra tree transformer and a simple optimizer (optimizer is optional). It is a convenience that lexical 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. There is an alternative and simple way to build up the AGT without tedious SQL grammar specification. That is to write a naive parser by yourself.

Secondly, the Engine's responsibility is to execute the SQL statement. Given a SQL statement, the SLEngine 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 some interfaces of indexing layer to make your engine executed easily.

5.2 SQL

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 database should support at least the following SQL sentences.

SELECT fields FROM tables [WHERE Wclause]
[GROUP BY fields] [HAVING fexpression] [ORDER BY fields]



fields ::= field , fields
field ::= name | name AS nickname | function (field) AS nickname | function (*)
name ::= fieldname.tablename | fieldname
function ::= AVG | COUNT | MIN | MAX | SUM
table ::= tablename , table
fexpression ::= function (field) COP rvalue
OP ::= + | - | * | /
Wclause ::= clause | Wclause WOP Wclause
WOP ::= AND | OR
clause ::= name COP rvalue
COP ::= > | = | < | >= | <= | <> | IN
rvalue ::= expression | subquery
expression ::= expression OP expression | value | (expression OP expression)
value ::= [-](1 | 2 | ... | 9)digit⁺ | 'string'
digit ::= 0 | 1 | 2 | ... | 9

CREATE TABLE tablename (ctclauses)
ctclauses ::= ctclause , ctclauses
ctclause ::= fieldname TYPE ([constrain]) [NOT NULL] [DEFAULT value] | KEYTYPE
(fields)
~~constrain~~ ::= digit⁺ | digit⁺, digit⁺
TYPE ::= CHAR | VARCHAR | DATE | TIME | REAL | DOUBLE | FLOAT | INTEGER
| LONG | DECIMAL
KEYTYPE ::= PRIMARY KEY | FOREIGN KEY

INSERT INTO tablename (fields) VALUE(values)
values ::= value , values

DELETE FROM tablename [WHERE Wclause]

UPDATE tablename SET uclauses [WHERE Wclause]
uclauses ::= uclause , uclauses | uclause
uclause ::= field ASSIGN rvalue
ASSIGN ::= =



The design of your SQL Engine is left BLANK here and should be completed all by yourselves. Your Engine should at least execute all SQL statements above. 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.

5.3 Engine

The Engine's responsibility is to execute the SQL statement. Given a SQL statement, the SQL Engine first parses it with SQL parser and then according to the parsed result, execute either query or update operation.



5.3.1 Query

The parsed result of SELECT statement is in itself a logical tree whose node is logical operators of the query. The executed result of each logical operator is also a table, which is represented as the Inter Result, a two dimensional tables. There are nine query operators:

- **Project** Project a table into another table with some specific fields selected. Each Inter Result is reconstructed into a new Inter Result with the proper fields' value, and all the records forms a new table.
- **Product** If a table with m rows products with another with n rows, the result is a $m \times n$ table of a cardisen product of the two tables.
- **Order** We sort the table records using the quick-sort algorithm and the processing is all in memory.
- **Group** First sort the records according to the specific fields; second combine the records with the same group fields value into a group and calculate the aggregate functions.
- **Compare** Compare the records of a table and only remains the records suffice the compare-condition.
- **Indexed Compare** Use the B+ Index Tree to find the proper records.
- **Table** It is the simplest, only to find the corresponding table and load it to memory.

5.3.2 Insert

First find all the fields that have been indexed. Find the heap file of the table and make a record and insert it into the heap file. Then insert the indic into all the B+ trees of the fields' indic. During this procedure, the expression in the insertion will be evaluated first.

5.3.3 Delete

In the parsed delete, the where condition is first evaluated and the proper sets of rids. And then the records with the specific rids are deleted. The selection of the rids is done like the query.

5.3.4 Update

Update is just like a combine of delete and insert. First select the set of rids specified in the where clause. Second read the records from the heap file, delete the indic from the B+ trees of the indexed fields. Then update the values of the specified fields and insert the newly modified record into the heap file and insert the indic into the corresponding B+ trees.

5.3.5 Transaction

The essence of transaction is the concurrency control of different transactions. We have adopted the two-phased lock strategies. At the beginning of each transaction, the Engine tries to lock all the tables involved in the transaction. If it fails to lock a table, it releases all the tables it has locked and wait several minutes to try again. Once it succeeds to lock the tables, it processes the statements one by one. After it completes the processing, whether with success or failure, it releases all the tables.

5.4 Concurrency Control

You should carefully design a concurrency control unit which may include 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 users 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 does writing operations on it. So you can set up a table lock to solve this problem.

Table lock is the kind of lock which supports 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 wants to do some writing operations on it, he must wait until the previous read lock is released.

When a user requires a transaction, at this time all the tables contained in this transaction must be locked. However, a dead lock may be occurred. You can use the simple method here 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 core system. 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 request comes from the out connections, the database manager holds the session and sends the request to the specific database.

For each database, there will be some system tables constructed at the initialization of the database, because the information such as the table description, the index description and the relation description should be recorded. The database manager is the only one who has the right to modify these tables, when the users want to add a new table, delete a table and so on.

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 also includes the logger, the user manager and configuration manager. Logger 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 whole behavior and thresholds for tuning, we use a configuration manager to read the all the needed parameters and thresholds at the start of the whole system.

6.2 Functions

- hold the connected sessions
- manage every opened database in current system
- package the result and do response

7 Access Interface Layer

7.1 Overview

As we said above, we use JDBC standards version 1.0 as our common interface. We recommend you to implement two JDBC drivers. One is for internal connection and for testing, and the other is for remote connection like a client-server architecture.

7.2 Internal Driver

The driver should be a part of JDBC 1.0 specification, which means not all functions need to be implemented. All the classes should inherit JAVA Standard JDBC class, which make you code support standard JDBC access. This internal driver is suitable for you to debug the system.

Here is an example code of this driver.

```
public class TestInternalJDBCdriver {
    private final static String PATH = "jdbc:fatworm:local:spring";
    public TestInternalJDBCdriver() {
    }
    public static void main(String[] args) {
        Connection conn = null;
        try {
            Class.forName("sjtu.acm.fatworm.jdbc.InternalJDBCdriver");
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            conn = DriverManager.getConnection(PATH, "smart", "blue");
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
```

```

        sql = "SELECT userID, password FROM spring ";
        ResultSet rs = st.executeQuery(sql);
        st.close();
        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

7.3 Remote Driver

This JDBC driver should support remote access, which leads to a client-server architecture of the whole system. When the database server is on, users can use this driver to connect to the remote server to operate their database. This driver will be the same with the internal JDBC driver in functions but has differences in connections. For instance, when we use internal JDBC driver to open a database named spring, we should use the protocol as **jdbc:fatworm:local:spring**. However, when we use Remote JDBC driver, we should use the protocol as **jdbc:fatworm:remote:rmi://192.**

168.0.1:1099/fatworm:spring. The text **rmi://192.168.0.1:1099/fatworm** is the ip address of the server and the name of the DBMS service running on the server. The text **spring** is name of the database. In this example, we use RMI protocol to manage our connections and data transfer. You can use any protocols you want to reach the same goal.

7.4 Database Server

When you want to build a client-server architecture, you need a server part, which accepts connections from the remote users and sends the results to the them. You can refer to the MySQL or SQL Server 2005's architecture.

~~8 Grading Policy~~

The whole score is divided into three parts, the required part, the recommended part and the advanced part, of which the ratio is 65:25:10. Also, we have the addition part for another 10 points. In the final review, you should prepare for a 10 minutes slides, which should describe the features of your system and brilliant points.

8.1 Required-65%

- virtual page based storage
 - write and read page
 - bitmap based page allocation and deallocation
- buffer strategy
 - buffer pool
 - replacement algorithm

- heap file
 - linked page construction
- B+ tree indexing
 - insert balanced B+ tree
 - insert and delete operation
- SQL parser
 - support basic SELECT statement on single table
 - support UPDATE and INSERT statement
 - support CREATE and DELETE operations on table
 - support expressions in WHERE clause
 - support ORDER BY, GROUP BY and HAVING operators
- SQL engine
 - system table maintenance
- database manager
 - support only one database opened in the system
 - support only one user logon into the system
 - user verification
- internal JDBC driver

8.2 Recommended-25%

- client-server architecture
 - database server
 - remote JDBC driver
 - support multiple users
 - support multiple databases opened in the system
 - session maintenance
- SQL optimizer
 - basic optimization
- concurrency control
 - read lock and write lock
 - support transaction
 - avoiding dead lock
- logging

8.3 Advanced-10%

- sub query support
- delete balanced B+ tree indexing
- more recovery strategy support
- advanced query optimizer

8.4 Addition-10%

- advanced buffer strategy
- full featured SQL parser
- full featured SQL engine
- GUI interface

8.5 Summary

Your final score consist of two parts. One is on how much you have been done, the other is got from your final review and document, the ratio of which is between 7:3 and 8:2.

~~9 Schedule~~

The project will be start in Week 10, and will last for 10 weeks. There will be several talks during the whole procedure, to give you some detailed information about each part of the DBMS.

- Week 10: team validation
- Week 11: project begins
- Week 12 - Week 19: coding, testing, tuning
- Week 14, Week 17: report the rate of progress(email)
- Week 20: final review

~~10 Document Requirement~~

The final document should contain the following sections:

- **Code Description** You should make a simple explanation of your final code, including the brief explanation of the structure of your codes, the clew of how to compile and run for the test cases and the result of your test cases. Here is an example: Structure of the codes: org.acm.parser - the package of the parser
org.acm.constant - the package of all constant classes
.....
org.acm.Main - the entry point of the whole compiler
How to Compile: Please run build.bat for compiling all codes related.

- **Brilliant Points** During the design and implementation of the project, if you think your idea is quite smart or fancy, please make a record here. Also, when you use some different methods from the directing instructions or have done some extensions beyond the basic request of project, mention them here. Of course, we'll consider adding some extra points for your excellent ideas.

The final document should be written in ENGLISH. And at last you should submit a single PDF file attached to your other files. We recommend you to use LaTeX for writing this document, which does help your future works.

Here is some materials about LaTeX. You can download all the related files from <http://www.ctex.org> and you will find a standard template in the files we give you. Then you can open this template.tex with WinEdt, click TeXify button on the tool bar and click dvi2pdf button. Firstly, you will see your document in DVI format, and then you can get your PDF result.