A File System for Accessing MySQL Tables as CSV Files

by

Nizameddin Ordulu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2010

Author ....
Department of Electrical Engineering and Computer Science
August 20, 2010

Certified by ....
Samuel Madden
Associate Professor
Thesis Supervisor

Accepted by ....
Cristhopher J. Terman
Chairman, Department Committee on Graduate Theses

A File System for Accessing MySQL Tables as CSV Files

by

Nizameddin Ordulu

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, we describe the design and implementation of a userspace file system that represents MySQL tables as comma-separated values (CSV) files. The users can access and modify the data through both the file system and MySQL's query interface. In order to transform read and write operations to SQL queries, we maintain a reverse index from file offsets to line numbers. Changes to the database outside of the file system are reflected on the file system by means of MySQL's master-slave replication feature. We evaluate our system by comparing its performance to a regular file system's performance using popular command line tools(grep, sed, awk) and user applications (OpenOffice.org spreadsheets). The choice of the database for this system was MySQL because of its popularity and the availability of its source code, however, the same ideas can be applied to any relational database with replication to create a similar system.

Thesis Supervisor: Samuel Madden
Title: Associate Professor

Acknowledgments

I am indebted to my advisor, Samuel Madden, for suggesting me this idea, and helping me transform the idea into this project. Without his generous help, it would be a lot more difficult for me to survive at the times of frustration, uncertainty and doubt.

I am also indebted to Carlo Curino, for his suggestions for the tools that I used to build this system and his contributions for the overall design of the system. Thank you Carlo, I truly appreciate your help.

Finally I would like to thank my wife, Fatma Ordulu, for being supportive and for encouraging me to work on my thesis throughout the entire process. Thank you for keeping me going when I thought I didn't have the energy to do so.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The system described in this thesis allows users to mount a MySQL database as a file system. After the file system is mounted, the user can perform changes on the tables as if they were CSV files. This section talks about the motivation, background and the structure of the rest of this work.

1.1 Motivation

Relational databases are widely used for storing and organizing tabular data, however, they are not easily accessible for non-technical users. Both for technical and non-technical users, the fastest and easiest way to create and manage small sized tables is to use spreadsheets such as Microsoft Excel or OpenOffice.org. For example, most researchers in Compuational Biology use spreadsheets to manage large collections of genome data. Most spreadsheets provide tools to run simple aggregate queries over the data. The problem occurs when users want to run complex queries, or want to run a query on two different tables in a nontrivial way. At this point, an engineer has to import the data into a database and run the queries that the actual data gatherers demand.

There are two main problems with importing a spreadsheet into a database in the previous scenario. Firstly, a non-technical user may not be able to perform this operation; even if she can do so, it's a distraction from the actual work of managing the data. Secondly, even after importing the data, if she still wants to continue to add more data or edit the data, she has to either learn how to manage the database using more technical tools, or keep the data in spreadsheets and rerun the import script every time she wants to execute complex queries over the data. Eliminating the need to import the data into the database would be convenient for the user.

Another difficulty with regards to managing data occurs when a small company grows over time and it becomes necessary to switch from using spreadsheets to database-backed applications for departments such as human resources. When this happens, the employees have to learn the new software, and they are forced to give up on the spreadsheets that they had used so far. With our file system, a company can continue using spreadsheets to manage their internal data even after switching to a database-backed application.

Another use case for the file system interface for a database is the desire to use file system utilities on tables. Sometimes, it might be convenient for a developer to use UNIX tools such as `grep` or `sed` to find a particular string in a table or change all the occurrences of a string in a table quickly. He can also his favorite file editing scripts by mounting the database as a file system.

Our system addresses the above issues by representing tables as CSV files in the local file system. We chose CSV as the format to represent the tables because it is supported by mainstream spreadsheet editors, it's human-readable, and it is a convenient format for UNIX file system utilities.

1.2 Background

In this section we briefly talk about file system in user space (FUSE), and MySQL replication. It is important to have a basic understanding of these systems in order to understand how our system works.

1.2.1 FUSE

FUSE is a kernel module for UNIX systems that allows developers to write file systems in user space without directly interacting with the kernel[1]. It's used to create a vast variety of file systems, some of them solving interesting problems. Examples of FUSE-based file systems include NTFS-3g (Tuxera, commercial version) – an NTFS wrapper for UNIX file systems[9] and WikipediaFS [10] – a file system that allows wikipedia entries to be read and edited as text files. The MS Windows equivalent of FUSE is Dokan library [11]. Dokan is still under active development and not ready for serious development.

1.2.2 MySQL Replication

MySQL replication enables the changes on the master MySQL database server to be replicated on slave MySQL servers. MySQL replication is asynchronous, meaning that the slaves don't need to be connected to the master permanently in order to receive updates. A slave can connect to the master server at any time to receive the updates that happened since the last time it connected. In our system there is a thread that runs on the client that acts as a slave MySQL server from the perspective of the actual MySQL database server. There is no database server that needs to be run locally. The slave thread is started automatically when the file system is mounted. The updates that are received by this slave thread are used to update internal data structures that are used to read and write the tables as files.

The rest of this document is structured as follows: In Chapter 2 we discuss the overall design of the system and explain how it works at a higher level. We explain the internal data structures that we use to transform file offsets into SQL queries in Chapter 3. Chapter 4 explains what happens in the system when user applications read and write files; it also explains the slave thread that runs locally to receive the updates that happen to the database. In Chapter 5 we talk about the details of the implementation of the system. In Chapter 6 we describe the tests we crafted to test the consistency and the performance of our system. We talk about the related work in Chapter 7 and finally we talk about future work and conclude in Chapter 8.

Chapter 2

System Architecture

The architecture of our system can be seen in Figure 2-1. Our system is composed of two main modules: One of them is FUSE Callbacks – the set of callbacks for the FUSE kernel module and the other is the MySQL Slave Thread – the set of functions that update the file system when the database is changed externally. FUSE Callbacks and the MySQL Slave Thread use the Reverse Index Trees to transform between file offsets and row ids. This structure is discussed in Chapter 3.

The read and write operations by the user applications are handled by the FUSE Callbacks. The goal of this module is to transform read and write operations on CSV files into `SELECT`, `UPDATE` and `DELETE` queries on tables. This module uses the Reverse Index Trees in order to convert from file offsets and sizes into row ids.

The MySQL Slave Thread is responsible for monitoring the changes on the database that are caused by the applications operating on the database other than our file system. MySQL server is configured to log all the queries that change the database to the binary log. This module periodically asks for the latest entries in the binary log and modifies the Reverse Index Trees accordingly. We give the details for FUSE Callbacks and the MySQL Slave Thread below.

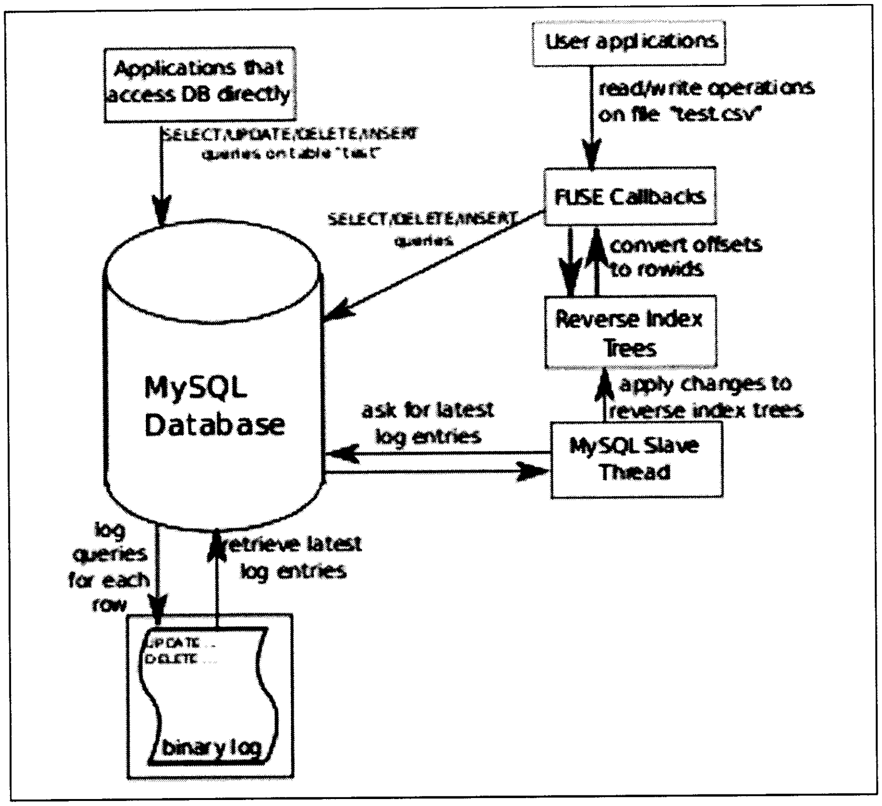

Figure 2-1: Overview of the different modules in our system

2.1 FUSE Callbacks

These functions are called by the UNIX kernel when user applications open, read or write files in the mounted file system. The main FUSE callbacks are given in Figure 2-2 below. We give a detailed explanation of what happens for each FUSE callback in Chapter 4.

When the file system is mounted, the `init()` function is called. Inside this function, we scan all of the tables in the database sequentially and build our internal reverse index that will be used for reads and writes to or from the tables. These indexes store the mapping between CSV file offsets and line numbers. The file names in the directory are the table names in the selected database appended with ".csv" extension. We don't physically store the tables on the disk in CSV format, instead, every time a `read()` is performed we craft the appropriate `SELECT` query to fetch the corresponding rows and then we fill the read buffer with the data in CSV format. When `write()` is called, a similar process is applied – the bytes written are

```
void* init(struct fuse_conn_info *conn);
int open(const char* path, struct fuse_file_info* fi);
int readdir(const char* path, void* buf,
            fuse_fill_dir_t filler, off_t offset,
            struct fuse_file_info* fi);
int getattr(const char* path, struct stat* stbuf);
int read(const char* path, char *buf, size_t size,
         off_t offset, struct fuse_file_info* fi);
int truncate(const char* path, off_t size);
int write(const char* path, char *buf, size_t size,
          off_t offset, struct fuse_file_info* fi);
int flush(const char* path, struct fuse_file_info* fi);
```

Figure 2-2: Main FUSE callbacks

transformed into an `INSERT` query. All writes are assumed to be append-only, therefore `write()`s aren't transformed into `UPDATE` queries. The justification for this assumption is given in Chapter 4.

Most user applications use temporary files for their operations, so we support the creation of these files. Temporary files are created and saved in a directory other than the mounted directory. The operations on temporary files under the original mounted directory are redirected to the files under the temporary directory. We assume that any file name not ending with ".csv" represents a temporary file. This can cause issues if the application tries to create a temporary file that ends with ".csv", however, this didn't happen for any of the applications that we used to test our system.

2.2 The MySQL Slave Thread

The MySQL Slave Thread queries the server for the binary log size every second. If the log size has changed, then the changes are requested from the server by issuing the MySQL-specific command `"SHOW MASTER STATUS"`.

The binary log on the MySQL server is configured to store the changes on a per-row basis. Each log entry is either an insertion or an update or a deletion of exactly one row. You can see a sample part from a binary log in Figure 2-3. The changes reflect `INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` operations on the table named

```
INSERT INTO sqlfs.test SET @1=11 @2='apple' @3=123
UPDATE sqlfs.test WHERE  @1=11 @2='apple' @3=123
                  SET @1=11 @2='orange' @3=12345
DELETE FROM sqlfs.test WHERE @1=11 @2='orange' @3=12345
TRUNCATE TABLE test
```

Figure 2-3: Four entries from a MySQL binary log

`test` in the database `sqlfs`. Note that the first column is `rowid` so the actual contents of the file are formed by the second and third columns.

A caveat is that our system doesn't support concurrent changes by the user applications editing CSV files and other applications connected directly to the database. The user must make sure that the table is not changing in the database while she is editing the corresponding CSV file. This prevents the changes on the database from interfering with the changes from the user applications. This is a restriction of the FUSE interface which we discuss in more detail in Chapter 5.

Chapter 3

Data Structures

The file system calls `read()`, `write()`, and `truncate()` deal with file offsets and file size. The database, however, doesn't know about the offsets and file sizes, so these operations must be transformed into `SELECT`, `INSERT`, `DELETE`, and `UPDATE` queries before executing them on the database.

Most databases, including MySQL, provide no guarantees on the order of the rows that are fetched from the output of a `SELECT` query. This causes a problem as new rows are introduced, because the order of the lines in the CSV file may change every time the user opens the file. In order to make sure that the order of the lines are fixed, we introduce an auto-incremented `rowid` column to all of the tables. The `rowid`s can be seen as the line number of each row, however, they are only used for sorting the rows. The particular `rowid` of a row may not exactly correspond to the line number that the row appears when the file is read, because certain rows may be deleted from the database. However, it is always the case that for two consecutive lines in a CSV file, the lower line corresponds to a row with a higher `rowid` than the higher line.

3.1 Reverse Index Trees

The data structures that help us determine the `rowid`s from file offsets are called reverse index trees, because they form a tree and they perform the reverse of what

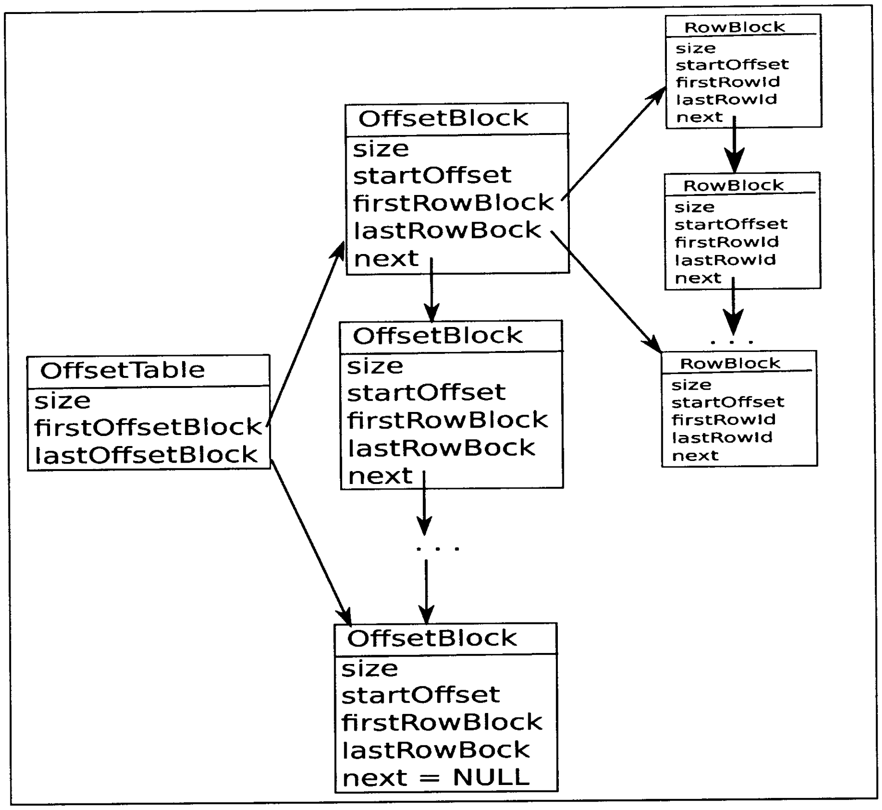

Figure 3-1: The tree of reverse indexes

regular indexes do. A visualization for these data structures is given in Figure 3-1. We have a different tree for each table in the database. Each tree has depth 3. Each node of the tree corresponds to a number of rows with contigious `rowids`. The root node is of type `OffsetTable` and it corresponds to the entire table. The first level nodes are `OffsetBlocks` and the leaves are `RowBlocks`. This tree structure is similar to a B-tree except that the branching factor is not bounded from above and the leaf nodes do not contain any data nor a pointer to the data.

The `size` in each node represents the size in bytes of the rows of that node in CSV format. Calculation of `size` takes the commas and the newlines into account. The `startOffset` field represents the byte-offset of each node relative to the beginning of its parent node. So for example, if an `OffsetBlock`'s `startOffset` is 100 and one of its children `RowBlock`'s `startOffset` is 50, then the absolute offset of the beginning of the child `RowBlock` is 150. The maximum `size` for a `RowBlock` is `ROW_BLOCK_SIZE` and the maximum `size` for an `OffsetBlock` is

rowid	a	b
1	Yonkers disco evictions	917815
2	soother consented grouts	244026
3	scramming Wong Lear	767339
4	ingrates airsicknesss soberer	841790
5	reproachfully Avogadro agendas	628429
6	arsenic refectorys snarls	484099
7	yeasty years nonprescription	748860
8	Amadeuss mixtures televisions	980288
9	havoc Blatzs clinchers	646756
10	gymnosperm homestretches inoculations	959294

Figure 3-2: Table named "test"

```
Yonkers disco evictions,917815
soother consented grouts,244026
scramming Wong Lear,767339
ingrates airsicknesss soberer,841790
reproachfully Avogadro agendas,628429
arsenic refectorys snarls,484099
yeasty years nonprescription,748860
Amadeuss mixtures televisions,980288
havoc Blatzs clinchers,646756
gymnosperm homestretches inoculations,959294
```

Figure 3-3: Contents of the file "test.csv"

`OFFSET_BLOCK_SIZE`. We discuss the optimal values for `ROW_BLOCK_SIZE` and `OFFSET_BLOCK_SIZE` in Chapter 6.

3.2 A Table and Its Reverse Index Tree

In order to illustrate the data structures and the operations on them, we give a simple example of a randomly generated table in Figure 3-2, the contents of the corresponding CSV file in Figure 3-3, and its corresponding `OffsetTable` in Figure 3-4.

The table `test` originally consisted of two columns: `a` and `b` which are of type `VARCHAR(256)` and `INT`. The `rowid` column is of type `INT AUTO_INCREMENT` and it was automatically added when our file system was mounted. The contents of the file `test.csv` are shown in Figure 3-3. In order to illustrate all of the data struc-

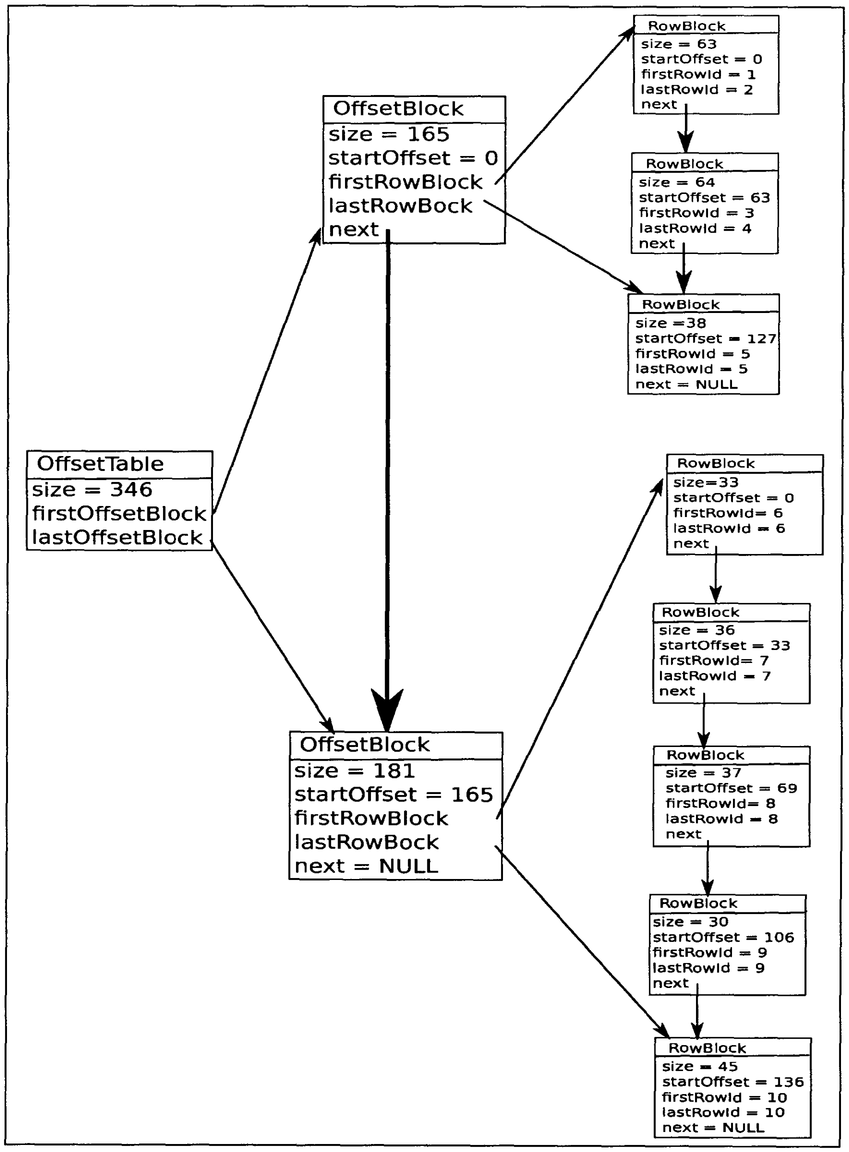

Figure 3-4: The OffsetTable tree for the table "test"

tures properly, we set `ROW_BLOCK_SIZE=64` and `OFFSET_BLOCK_SIZE=192`. The resulting `OffsetTable` is given in Figure 3-4

3.3 Tree Operations

In this section we describe the interface that supports the file system operations described in Chapter 4. For the sake of precision we use C++ notation.

```
struct RowOffset {
  long long int offset;
  long long int firstRowId;
  long long int lastRowId;
  long long int size;
}
RowOffset OffsetTable::findRows(long long int offset,
                                long long int size);
```

Figure 3-5: The findRows() method calculates the range of row ids for a given chunk of the file.

3.3.1 Offset Search

Offset search is the process of finding the range of `rowids` that correspond to a particular byte range in the file. This functionality is implemented as a method of `OffsetTable`. Its signature is given in Figure 3-5. `OffsetTable::findRows()` finds the range of rows that contain the part of the CSV file that starts at offset `offset` and spans `size` bytes. The return value is of type `RowOffset` and it contains the first and last `rowids` of the rows that need to be fetched from the database. The `offset` field of the return value represents the number of bytes that need to be skipped from the beginning of serialization of the fetched rows. The `size` of `RowOffset` is the total size of the serialization of the rows; it is the same as the `size` argument passed to the method, unless the chunk described by the arguments is past the end of file.

The implementation of this method works as follows: For each node in the reverse index tree, we maintain a map from offsets to the child nodes. Each child node is the image of its `startOffset` in this map. For each `offsetBlock` with parent `offsetTable`, we have

```
offsetTable->offsetMap[offsetBlock->startOffset] = offsetBlock.
```

Similarly, for each child `rowBlock` of an `offsetBlock`, we have

```
offsetBlock->offsetMap[rowBlock->startOffset] = rowBlock.
```

```
void OffsetTable::appendRow(long long int rowid,
                            long long int size);
```

Figure 3-6: The appendRow() method adds a row to the end of an OffsetTable.

```
void OffsetTable::updateRowSize(size_t rowid,
                                size_t old_size,
                                size_t new_size);
```

Figure 3-7: The updateRowSize() method updates the size of a row.

In both cases `offsetMap` is of the type `map` of the standard C++ library, which itself is a self balancing tree. This allows us to regard the `offset` values as keys to the `offsetMap` and find the in-order predecessor of `offset`. Using `offsetTable->offsetMap`, we first find the `OffsetBlock` that contains the given `offset`. Then we find the `RowBlock` that contains the given `offset` in a similar fashion. After finding the initial `RowBlock`, we keep scanning the successor `RowBlocks` and summing their sizes until the total size reaches the given `size`.

3.3.2 Adding A Row to OffsetTable

The signature of the function that adds a row to the `OffsetTable` is given in Figure 3-6. This is a rather straightforward operation; we update the size of the last `RowBlock` if it has capacity for more rows, otherwise we add a new `RowBlock` or a new `OffsetBlock` to accommodate for the new row.

3.3.3 Updating or Deleting a Row from an OffsetTable

The signature of this function is given in Figure 3-7. This method works by first doing a binary search on `OffsetBlocks` to find the `OffsetBlock` that contains the `row` with id `rowid`. Then a binary search is performed on the `RowBlocks` of that `offsetBlock`. When the `RowBlock` that contains the given `rowid` is found, its size is updated. Furthermore, the `startOffsets` of the succeeding siblings of the found `OffsetBlock` and the `RowBlock` are updated. Deleting a row means updating its size to 0.

Chapter 4

FUSE Operations and Binary Log Entries

In this chapter we give a thorough explanation of how FUSE Callbacks handle the changes on CSV files and how the MySQL Slave Thread handles the binary log entries. We often refer to the example given in Section 3.2 in the sections below. For the purposes of clarity and consistency, we assume that we are mounting our file system on the directory `/tmp/fs` and the database that we are mounting is `sqlfs`.

4.1 Table Class

The `Table` class acts as a wrapper for `OffsetTable` and maintains a write buffer for the CSV file. The contents of the buffer are transformed into an `INSERT` query and executed every time a newline character is written to the file. The interface

```
class Table {
  Table(string table_name);
  OffsetTable* offsetTable;
  size_t read(char *buf, size_t size, off_t offset);
  void truncate(off_t size);
  void write(const char* buf, int size, off_t offset);
}
```

Figure 4-1: The interface for the class "Table" for the class Table is given in Figure 4-1. The methods of this class are used in performing the FUSE operations described in this chapter.

The Table::read() method is used for reading the part of a file of size size starting at offset offset. It calls offsetTable->findRows(size, offset) to determine the corresponding range of rows that need to be fetched from the database. It fetches those rows, serializes them, and discards the first ro.offset bytes of the serialized data. Finally, it copies the string into buf.

The Table::truncate() method is used to truncate the file to a certain size. This means deleting the rows whose rowids are greater than the rowid that corresponds to the line of the file which ends at byte size of the file.This is accomplished by calling RowOffset ro = offsetTable->findRows(size, 0) and issuing the appropriate DELETE query to delete the rows with rowids greater than ro.lastRowId.

The Table::write() method is used to write contents to the CSV file. All writes are assumed to be append-only, because this holds for all the end-user applications that we tested with. The updates in the middle of the file are ignored. The contents of the write buffer are parsed and a new row is inserted into the database every time a newline character is found in the write buffer. The crafted INSERT query does not set any rowid columns, because the rowid column is auto-incremented by the database.

4.2 FUSE Operations

The FUSE callbacks given in Figure 2-2 are called by the FUSE kernel module when the user applications want to access or modify a file. FUSE is multi-threaded by default so our system employs a basic read-write locking mechanism to prevent race conditions and deadlocks. We discuss locking more in Chapter 5. All the FUSE callbacks that return an int return 0 or a positive integer on success and a negative value chosen from errno.h on error.

```
void* init(struct fuse_conn_info *conn)
```

Figure 4-2: FUSE init callback

4.2.1 Initialization of the File System

The FUSE callback `init()`'s signature is given in Figure 4-2. The argument `conn` contains information about what features are supported by FUSE and it is ignored.

When this function is called, the list of tables are obtained by executing the query `"SHOW TABLES"` on the database. We go through all of the tables and fetch all of their rows sorted by `rowid` in order to build the `OffsetTables` discussed in Chapter 3. For the table given in Section 3.2, the SQL query executed is `"SELECT * FROM test ORDER BY rowid"`.

Rows are added by making successive calls to `OffsetTable::appendRow()`. A global dictionary from the table names to the `Table` objects is created.

We also create the directory for the non-CSV files if it doesn't exist during the execution of this callback. The path for this directory is `/tmp/.fs` for our example and in general it is derived similarly from the path of the original mount directory. In the subsections below, we generally omit the file system operations on non-CSV files, because these are accomplished by calling their counterpart system calls on the regular file system under `/tmp/.fs`. So, for instance, if a user application creates a file called `file.txt` under `/tmp/fs`, our file system creates it under `/tmp/.fs`. Similarly, if an application wants to read `/tmp/fs/file.txt`, then, our system reads the contents of the file `/tmp/.fs/file.txt` provided that it exists.

4.2.2 Listing the Directory Contents

The signature of the `readdir()` callback is given in Figure 4-3. Despite the somewhat complex signature, this function is actually simple; for each file under the directory, it calls the callback `filler` and passes it `buffer` and the file name. The file names that are passed to `filler` are the table names concatenated the string ".csv" and the files under the non-CSV directory `/tmp/.fs`.

```
int readdir(const char* path, void* buf,
            fuse_fill_dir_t filler, off_t offset,
            struct fuse_file_info* fi)
```

Figure 4-3: FUSE readdir callback

```
int getattr(const char* path, struct stat* stbuf)
```

Figure 4-4: FUSE getattr callback

4.2.3 Getting the Attributes of a File

The signature for the callback `getattr()` is given in Figure 4-4. The `getattr` callback is passed the path of the file and a parameter of type `struct stat` to be filled with file's attributes. The only field of `stbuf` that matters for the purposes of our system is the `size` field. We set `st_buf->size` to the size of the `OffsetTable` associated with the file. The access permissions are also stored in `stbuf`; we set them so that the file is readable and writable by everyone. Adding mappings between the database and the UNIX access control systems, while possible, is beyond the scope of this work.

4.2.4 Opening a File

The signature for the `open()` callback is given in Figure 4-5. The parameter `fi` is an object that can hold information about the file that's about to be opened, and the same object is passed to the subsequent FUSE calls. We store the pointer to the `Table` object in `fi->fh`.

```
Table* table = Table::Get(table_name);
fi->fh = (uint64_t)table;
```

```
int open(const char* path, struct fuse_file_info* fi)
```

Figure 4-5: FUSE open callback

```
int read(const char* path, char *buf, size_t size,
         off_t offset, struct fuse_file_info* fi);
```

Figure 4-6: FUSE read callback

```
int truncate(const char* path, off_t size)
```

Figure 4-7: FUSE truncate callback

4.2.5 Reading from a File

The signature for the `read()` callback is given in Figure 4-6. This function calls `Table::read()` if `path` specifies a CSV file, otherwise it reads from the corresponding regular file under `/tmp/.fs`.

4.2.6 Truncating a File

The signature for this callback is given in Figure 4-7. The expected outcome for this callback is that the file is truncated to the specified size. All text file editors truncate the file before writing the changes to the file.

Our system handles truncation by first looking at whether the `size` is zero. If it is, then we delete all of the rows by executing a `TRUNCATE` query. For our example in Section 3.2, this query is `TRUNCATE test`. We also remove all the nodes in the corresponding `OffsetTable` tree. If `size` is not zero, then `table->truncate()` is called with the argument `size`.

4.2.7 Writing to a File

The signature for this callback is given in Figure 4-8. This function just calls `table->write()` with the same arguments. All writes are assumed to be append-only. There are some common editors that requires this call to handle special cases.

```
int write(const char* path, char *buf, size_t size,
         off_t offset, struct fuse_file_info* fi);
```

Figure 4-8: FUSE write callback

These special cases are discussed in Section 5.3.

4.3 Handling MySQL Binary Log Entries

The FUSE module allows our system to propagate the changes on the file system onto the database, however, it can not detect the changes in the database in order to update the file system. The changes on the database are monitored by a thread on the client which makes use of MySQL's master-slave replication. Because we are not storing any serialized CSV versions of tables on the disk, we only update the reverse index trees when a change on the database is detected.

The MySQL Slave Thread works by polling the master for the binary log offset every second. The query for this is "`SHOW MASTER STATUS`". Once it is detected that the binary log offset has changed, the client requests the changes. MySQL server binary log is configured to be row-based rather-than statement-based (see [12] for an explanation of both formats). This is important, because if it were statement-based, then it would be impossible to update the reverse index trees appropriately. In order to make this clear, imagine a user executed the following query on the database: `DELETE FROM table1 WHERE column1 = 40`. This statement by itself does not describe how the reverse index tree should be changed, and it's impossible to determine with our data structure because the deleted row data has been lost; the only option is to recreate the entire reverse index tree from scratch. On the other hand, row-based replication allows us to update the reverse index trees because as we see in the following sections, both the old row data, and the new row data is preserved in row-based replication. The configuration parameter for row-based replication is `binlog_format` and it should be set as follows: `binlog_format=row`.

4.3.1 INSERT,DELETE and UPDATE Entries

`INSERT`,`DELETE` and `UPDATE` entries are handled by calling `offsetTable->updateRowSize()`, passing the `rowid`, the old size of the row and the new size of the row as parameters. For an inserted row the old size of the

```
INSERT INTO sqlfs.test
  SET @1=1 @2='Yonkers disco evictions' @3=917815
DELETE FROM sqlfs.test
  WHERE @1=7 @2='yeasty years nonprescription' @3=748860
UPDATE sqlfs.test
  WHERE @1=9 @2='havoc Blatzs clinchers' @3=646756
  SET @1=9 @2='CHANGED' @3=646756
```

Figure 4-9: INSERT, DELETE, and UPDATE entries from a binary log

row is zero; for a deleted row, the new size of the row is zero. Examples of parsed binary log entries are shown in Figure 4-9. We assume here that `rowid` is always the first column of a table and that the users do not modify the `rowid` column of any row. This is reasonable, because the `rowid` column is added by our system in order to keep the lines of the CSV file sorted.

4.3.2 TRUNCATE, DROP TABLE and CREATE TABLE Entries

`TRUNCATE` is handled by resetting the corresponding `OffsetTable` to an empty tree. `DROP TABLE` is handled by discarding the corresponding `OffsetTable` from memory. `CREATE TABLE` is handled by reconstructing the `OffsetTable`.

Chapter 5

Implementation Details

In this chapter, we talk about some of the interesting challenges that we faced during the implementation. We also talk about how we handled concurrency.

The system is implemented in C++, developed and tested on an Ubuntu Linux 10.04. We used Samba network file system to mount directories on MS Windows in order to test common MS Windows editors such as, MS Word, MS Excel and MS Notepad.

5.1 Preventing Duplicate Modifications to the Reverse Index Tree

In our first implementation of the system, when users modified the CSV files, our system modified the corresponding `OffsetTable` and executed a SQL query on the database. The SQL query caused new binary log entries to be recorded on the binary

```
void inc_truncate(const string& table);
bool shall_skip_truncate(const string& table);
void add_insert(const string& table, size_t rowSize);
bool shall_skip_insert(const string& table, size_t rowSize);
void add_delete(const string& table, size_t rowSize);
bool shall_skip_delete(const string& table, size_t rowSize);
```

Figure 5-1: Functions that determine which entries should be omitted log, and these were in turn picked up by the MySQL Slave Thread and the reverse index tree was modified for the second time. This caused every change to be applied twice, so if a line of size 15 was inserted, then the total file size would increase by 30, or if a line of size 10 was deleted, then the total file size would decrease by 20. If the entire file were truncated, and then changes were made, then the file would be truncated again when the MySQL slave thread encountered the `TRUNCATE` entry on the binary log.

In order to overcome this inconsistency, we keep track of the total size changes caused by `truncate()` and `write()` FUSE callbacks, and we ignore the binary log entries of type `DELETE` and `INSERT` until the total size of the ignored entries matches the size of the `DELETE`s and `INSERT`s caused by the FUSE callbacks. The signatures of the functions that are used to determine if a binary log entry should be omitted are given in Figure 5-1.

The FUSE callback functions call `add_insert()`, `add_delete()` and `add_truncate()` functions whenever they make a change to an `offsetTable`. The MySQL Slave Thread checks if an incoming binary log entry was already applied to the `offsetTable` by calling `shall_skip_insert()`, `shall_skip_delete()` and `shall_skip_truncate()` functions. This solution assumes that the database isn't concurrently modified by both the FUSE system and an application executing queries directly on the database. This is a reasonable assumption, because we require the user to be the only party interacting with the database while he or she is editing the CSV file.

5.2 Concurrency

Even though most user applications use a single thread when editing a CSV file, FUSE is inherently multi-threaded and it uses threads to read parts of the file in parallel. We use a read-write locking mechanism to prevent deadlocks and race conditions. Every `OffsetTable` has a read-write lock; functions that modify the structure of the table acquire the write lock before their modification and functions that only read data acquire the read lock. At any point in time, there can only be one thread that has the write lock, and if a thread has the write lock, then no other thread can have either the write lock or the read lock for the same `OffsetTable`. `OffsetTable::findRows()` and `OffsetTable::getSize()` acquire the read lock; `OffsetTable::OffsetTable()`, `OffsetTable::updateRowSize()`, `OffsetTable::appendRow()`, `OffsetTable::truncate()`, `OffsetTable::clear()`, and `OffsetTable::~OffsetTable()` acquire the write lock before accessing the fields of the `OffsetTable`. We used pthreads library for threading and locking.

5.3 Corner Cases

There are two main corner cases that we needed to handle, both of which affect only MS Windows users: One is that MS Windows programs insert carriage return (CR) characters before the line feed characters. Our system ignores these characters when writing to the file. The second is that it is quite easy to accidentally insert invisible backspace characters when using MS Notepad. These characters are also ignored when writing to the file.

Chapter 6

Evaluation and Analysis

In this section, we present the performance of the system and compare it to the performance of the regular file system. Since the system is designed for use by the end users, high performance is of secondary importance, however, it needs to be fast enough to provide a smooth user experience. In addition to the performance tests, we stress tested the consistency of the system by reading and writing at different positions in the files by different sizes.

6.1 Testing Methods

Most end user editors operate on the moderate-sized files as follows:

1. Read the file into memory.
2. While the user modifies the file, new content is kept in memory or is saved in a temporary file.
3. When the user saves the file, the original file is truncated entirely or partially, then the new contents are written.

In step 3, most editors truncate the entire file (equivalent to deleting all the rows of the corresponding table) rather than truncating partially. Among the ones we tested, the only editor that did not completely truncate the file before writing was OpenOffice.org Spreadsheets.

We tested the file system using both command line tools such as `sed`, `grep`, `less`, python scripts as well as the graphical editors such as MS Word, MS Excel, notepad, and OpenOffice.org Spreadsheets. The performance results do not include metrics for the graphical editors, however, in our experience their performance on our file system was indistinguishable from their performance on a regular file system.

We chose `ROW_BLOCK_SIZE = 512KB`, `OFFSET_BLOCK_SIZE = 1024KB`, and they work well. The file system operations become significantly slower when `ROW_BLOCK_SIZE` is as large as the size of the files that are handled and when it is so small that only a couple of rows can fit in one `RowBlock`. Too small and too big values don't work well, because the former requires too many tree operations to fetch all the data to read, while the latter requires to fetch too many rows from the database even to read a small part of the file.

6.2 Performance Comparison

Table 6.1: Performance Comparison to The Regular File System

	small.csv		big.csv	
test	regular	FUSE	regular	FUSE
`read`	0.0	0.0	0.30	3.76
`write`	0.06	0.09	0.65	6.70
`grep`	0.01	0.02	0.23	4.36
`sed`	0.02	0.15	0.85	11.96

Table 6.2: SELECT and INSERT performances

test	`small`	`big`
`SELECT`	0.0	0.57
`INSERT`	0.03	1.52

The initalization (`init()`) of the file system with the files `small.csv` and `big.csv` takes 1.8 seconds. The results for the performance tests applied after initializing our file system and the regular file system are given in Table 6.1. The first column contains the names of the tests. The `read` test involves reading the file from the beginning to the end. The write test involves writing to an empty file. In the grep test, we execute

"grep apple <filename>.csv";

in the sed test, we execute

"sed -i'' -e 's/apple/orange/g' <filename>.csv".

small.csv is a CSV file that contains 1000 lines and its size is 135K. big.csv contains 150000 lines and its size is about 20M. We see that the read operations are at most 20 times slower and the write operations are at most 15 times slower. Although the performance of the operations is not particularly good for a file system, it is still usable and practical if the user wants to make changes to the table using command line utilities. For comparison, we provide the time that it takes to run SELECT and INSERT queries in 6.2. The SELECT test measures the time it takes to run the query SELECT * FROM <table> and then fetch all of the rows that belong to the table <table>. The INSERT test measures the time it takes to insert 1000 rows to small table and 150000 rows to big table. All rows are inserted by one INSERT query.

Table 6.3: Ratios of the time spent by different modules in our system

test	String Ops %	Reverse Index Tree %	MySQL Slave Thread %	Database Ops %	Other %
read	50	5	0	44	1
write	41	6	49	3	1
grep	52	0	0	47	1
sed	49	12	16	18	5

Using gprof, we also determined how much time different parts of our system consume. The results are given in Table 6.3. "String Ops" column represents the percentage of time spent by the string manipulation to transform the data fetched from the database into CSV and to transform CSV data from files into SQL queries. "Reverse Index Tree" column represents the percentage of time spent by tree operations, "MySQL Slave Thread" column represents the percentage of time spent by the MySQL Slave Thread to apply the changes on the database to the reverse index trees, and "Database Ops" column represents the percentage of time spent by executing database queries. We see that the overhead in read operations is almost equally contributed by the database queries and string manipulation. In write operations, the database queries do not contribute to the overhead as much as the string manipulation, because it takes more time to create database queries from CSV data than it takes to create CSV formatted strings from database query results. Database queries and string manipulation are the main bottlenecks of our system. The former can be improved by making database calls using MySQL C API– we currently use MySQL++ [13] as our database wrapper. The latter bottleneck can be improved by rewriting the string manipulation code in assembly.

Chapter 7

Related Work

7.1 MySQL CSV Engine

Most MySQL-server packages come with the CSV storage engine which stores the data in CSV files[3]. This provides a similar functionality to our system, but supports only reading, not writing the CSV files by user editors. If the user modifies the CSV file, the changes are not reflected on the database. A benefit of this storage engine is that it provides an easy way to move and replicate the tables between databases by just copying the CSV file. A disadvantage of this system is that CSV engine doesn't support indexing and it's much less efficient than MyISAM and InnoDB engines.

7.2 DBToy

DBToy [4] is a file system in user space that represents the tables in an XML format. This system allows database managers to view the data using browsers. This is a read-only file system.

7.3 RDB

RDB is a commercial database that consists of 131 shell commands[5]. RDB is inspired by the research paper "The UNIX Shell As a Fourth Generation Language"[6] which defines a fourth generation language to be a relational database system that leverages the power of piping UNIX commands. RDB is highly modular and has a light memory footprint. RDB saves its data in text files in human readable format.

7.4 Flat-Text Database for Shell Scripting

Flat-text database for shell scripting [7] is a database that efficiently handles medium amounts of data using flat ASCII databases allowing manipulation with shell scripts. The file format for FSDB is such that each row is on its own line, and the columns are separated by a space character. Instead of parsing SQL queries, FSDB provides its own language in which statements are constructed using binaries and UNIX pipes. It has a rich set of commands that computes statistics over the columns of a table. FSDB aims to be a lightweight toolbox for the researchers who want to keep moderate amounts of data in plain text files and run sophisticated queries over them. FSDB is highly influenced by RDB. It doesn't have some of the advanced feratures of the commercial RDB, and is not as fast, however FSDB is free and it has a richer set of tools for statisticians.

7.5 QueryCell, An MS Excel Plugin

QueryCell is not a database, but it is trying to solve a similar problem to what our system tries to solve. It is a plug-in to MS Excel, that uses the embedded database FireBird in order to run SQL queries. QueryCell operates by dumping all of the data on the MS Excel cells in to a table in FireBird, then executing the SQL query provided by the user. Although practical for small to medium sized files, this method is inefficient for spreadsheets that contain a large number of rows.

Chapter 8

Conclusion

In this thesis, we described a userspace file system that allows the users to view and modify MySQL tables as if they were CSV files. We examined other systems that try to solve similar problems. Some of those systems build database functionality using CSV files as their data storage. There are two major advantages in our approach to this problem: first of all, building and maintaining reverse indexes is simpler than building and maintaining the combination of indexes, SQL interface, transaction logic and the other common database features. Secondly, the applications that use the database interface have a higher performance-priority than the applications that use the file interface. For example, CSV files may be accessed by applications such as MS Excel, MS Word, or notepad each of which is used by only one user at a time where the performance is not of utmost importance. In contrast, the database interface may be used by a web server that is accessed by multiple users who carry out monetary transactions that require high degree of concurrency where the performance matters. Therefore it is more efficient to build a CSV file system interface for a database than building a database backed by CSV files.

On the other hand, the performance of our system is worse than an optimal file system. The main bottlenecks are the execution of the database queries and the string manipulation for transforming CSV data into SQL queries and query results into CSV data. None of these bottlenecks can be entirely eliminated. Nevertheless, this doesn't constitute a big problem for the end user applications and command line utilities as we discussed in Chapter 6.

In addition to being a good showcase for userspace file systems, our system has many practical uses. It makes it easy for non-technical users to manipulate database tables using user-friendly file editors. It also allows technical users to quickly search for a string in a table or to delete a row from a table using command line utilities. By representing the database tables as CSV files, our system allows the users to use any application that can operate on CSV files in order to access or modify the tables in a database.

8.1 Future Work

The two most useful improvements to the current system would be adding MS Windows support using Dokan [11] instead of FUSE and adding support for databases that have replication other than MySQL. As mentioned in Chapter 6, the slow parts of the system can be improved by using MySQL C API instead of MySQL++ and writing the string manipulation code in assembly, however, none of these improvements would result in a dramatic increase in performance.

Bibliography

[1] *FUSE project.* Home page.
http://fuse.sourceforge.net.

[2] *MySQL Replication.*
http://dev.mysql.com/doc/refman/5.0/en/replication.html.

[3] *MySQL CSV Storage Engine.*
http://dev.mysql.com/tech-resources/articles/csv-storage-engine.html.

[4] Domenico Rotiroti, *DBToy: Mapping Relational Databases to File Systems - User's Manual*
http://cloud.github.com/downloads/drotiro/dbtoy/DBToy_UserManual.pdf

[5] *RDB, Relational Database Management System for Unix*
http://www.rdb.com/man/rdbman.html

[6] Evan Schaffer, Mike Wolf, *The UNIX Shell As a Fourth Generation Language*
http://www.rdb.com/lib/4gl.pdf

[7] John Heidemann, *A flat-text database for shell scripting*
http://www.isi.edu/~johnh/SOFTWARE/FSDB/Fsdb-2.21_README.html

[8] *QueryCell, A SQL plugin to MS Excel*
http://www.querycell.com/GettingStarted.html

[9] *Tuxera NTFS solution*
http://www.tuxera.com/

[10] *A file system to edit Wikipedia articles*
http://wikipediafs.sourceforge.net/

[11] *A FUSE Equivalent Module for Windows*
http://dokan-dev.net/en/

[12] *Comparison of row-based and statement-based replication*
http://dev.mysql.com/doc/refman/5.1/en/replication-sbr-rbr.html

[13] *A C++ wrapper for MySQL C API*
`http://tangentsoft.net/mysql++/`