Concrete Architecture
–
Recovery Manager of MySQL

EECS 4314 - Advanced Software Engineering
Fall 2017
York University

Team

1. Benjamin Zaganjori (212987533)
2. Dan Sheng (213587712)
3. Daiwei Zhou (209112905)
4. Hassaan Ahmed (213155700)
5. Matthew MacEachern (213960216)
6. Shagun Kazan (213741707)
7. Tushar Chand (213258009)

Table of Contents

Abstract

This report is prepared to provide the as-built (concrete) architecture based on the presentation of a conceptual architecture in the prior report on MySQL. The report reviews the conceptual architecture from our prior report and then presents our findings on the concrete architecture of MySQL. This document specifically focuses on one of the specific subsystems within the MySQL architecture, the Recovery Management.

We share our findings from the LSEdit software architecture extraction tool on how it was used to establish the dependencies between different subsystems within the Recovery Management as well the entire concrete architecture. The concrete architecture was compared to the conceptual to find similarities and discrepancies between different subsystems. Based on these results, we conclude the report with sharing our derivation process and the lessons learned throughout this activity.

Introduction

This report is designed to present the findings on the concrete architecture of MySQL v8.0 Relational Database Management System (RDBMS). The report is designed to inform our readers of how the concrete architecture compares with the conceptual architecture we presented in our earlier report as well as specifically going in detail over the Recovery Management subsystem within the Logical Layer of the design.

The report starts off by discussing the architecture style and design patterns applicable to the MySQL RDBMS. We then move forward with providing a Reflexion Analysis through the comparison of the presented conceptual architecture from Assignment #1 and presenting the concrete architecture as part of this assignment. We also go in depth with the Recovery Management subsystem particularly discussing the functionalities of the two main subsystems found within in it, the Log Manager and the Recovery Manager. As part of the reflexion analysis, we disclose the divergences and absences found between the conceptual and concrete architectures.

Lastly, as part of this activity we share the derivation process and the lessons learned to help fellow academics hoping to learn about the architecture of MySQL.

1. Architectural Styles MySQL

One architecture style applicable to MySQL is the Master-Slave architectural style where slave databases are created by replicating the master database, primarily to help with the load by spreading out the read responsibilities across the slaves while the master slave takes on the write tasks. Additionally, such an architecture helps support scalability, fault tolerance and failover. The slaves function as a copy of the master database receiving information from the master in near real-time. [7]

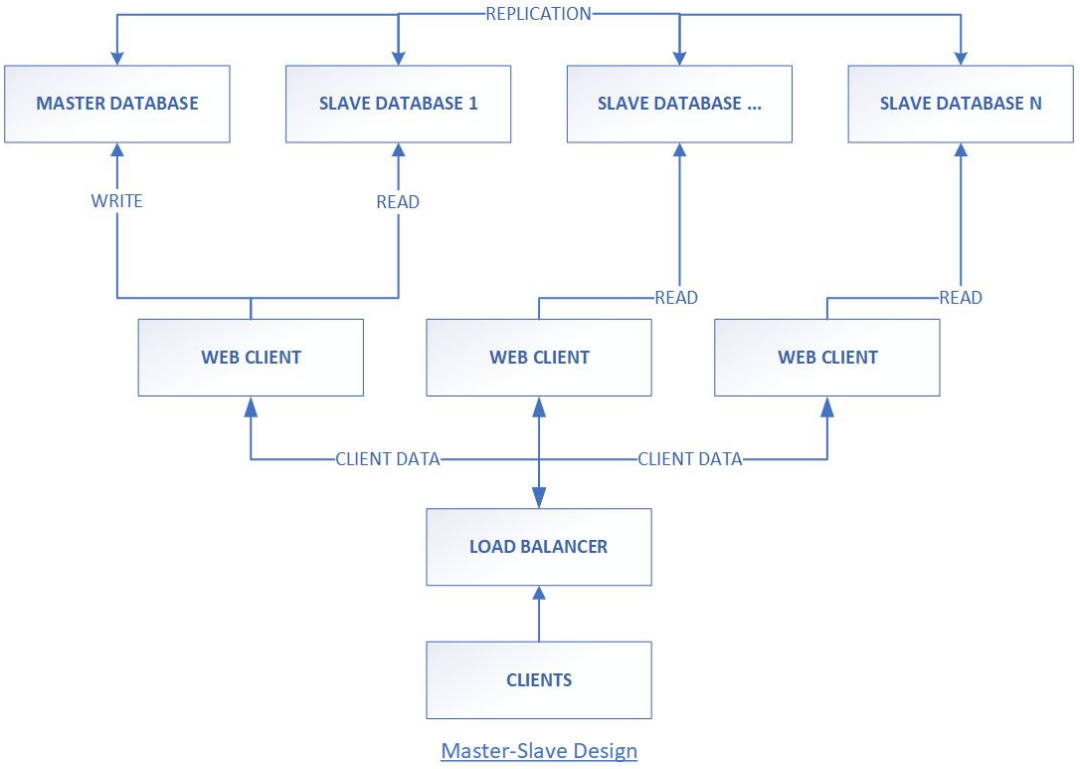

Figure 1. Master Slave Example Design

2 Reflexion Analysis

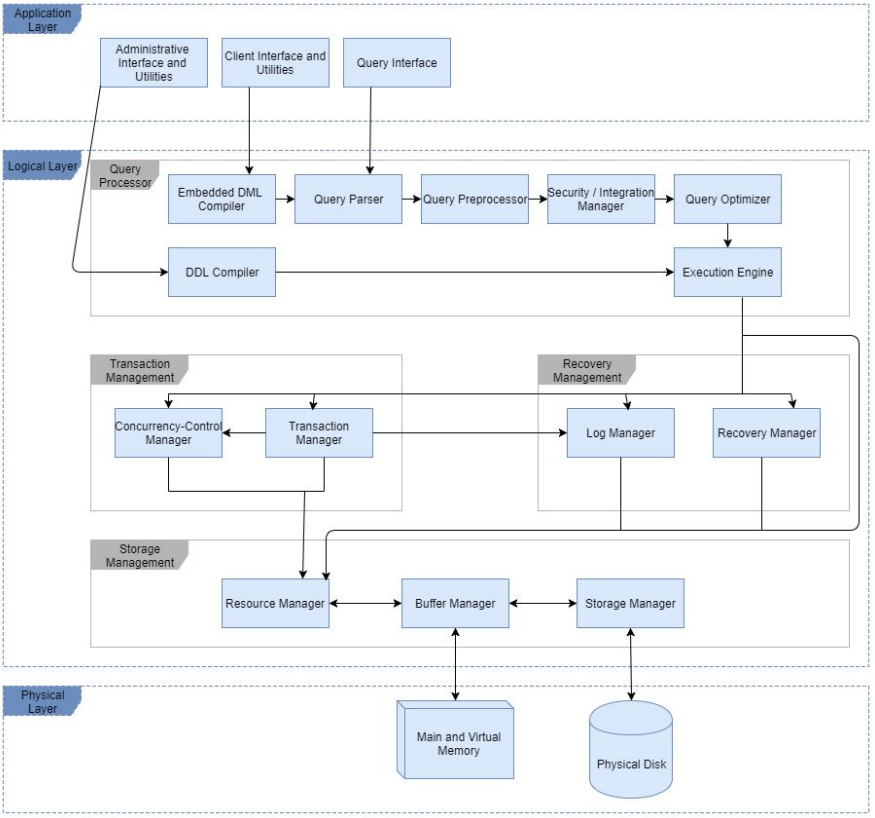

Figure 1. MySQL Conceptual Architecture

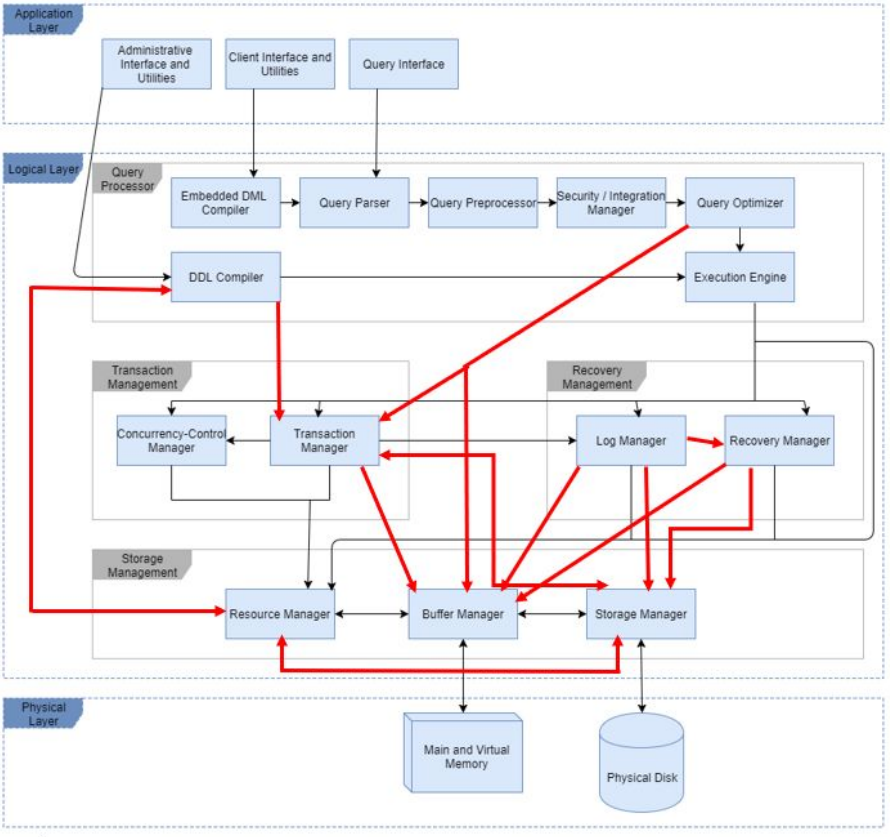

Figure 2. MySQL Concrete Architecture

Looking at Figure 2 and comparing it to Figure 3, you can see the additional dependencies that were extracted using the iterative method with the several TA files that were provided for the MySQL system. In Figure 1, the dependencies that our team extracted by taking a high level analysis of the MySQL API and documents and are shown in black. In Figure 3, the additional dependencies that were extracted through analyzing the source code files for MySQL are shown in red.

Looking specifically at the Recovery Management subsystem in Figure 3, we previously had imagined that both the Log Manager and Recovery Manager were independent subsystems that received information from each other (specifically the Recovery Manager receiving information from the Log Manager) through the Resource Manager in the Storage Management subsystem. This is an idealized structure that would have made conceptualizing about the interaction between these two subsystems more pipelined and straightforward: the Log Manager would store the logs through passing its information to the Resource Manager who would be solely responsible for communicating with Buffer Manager who would in turn be responsible for communication with the Storage Manager. However, through our analysis we discovered that both the Log Manager and the Recovery Manager have a more direct access to both the Buffer Manager and the Storage Manager, in a way breaking this idealized pipeline architecture. Looking at log0recv.cc in the InnoDB directory of the software system, we see that the hashed log tables are written directly to the buffer using the buffer manager. This is not to say that the conceptual architecture that was derived in the first assignment is wrong necessarily, but that at some point in the development process it was decided to stray from the idealized pipelined strategy for various reasons that will be discussed further throughout this paper. Additionally, the Log Manager has a dependency to the Recovery Manager.

3 Use Case Sequence Diagram

Figure 4 shows some of the basic interactions that occur in the MySQL software system during normal operation between the Log Manager (log0log.cc), the Recovery Manager (log0recv.cc), and the Buffer, which as it turns out both the Log Manager and Recovery Manager have more direct access to. For obvious reasons, not all of the functions that are performed can be shown, but the fundamental operations are shown in the diagram. During execution the Recovery Manager adds records to the hash table of log records using recv_add_to_hash_table(), which it then can copy to the Buffer using recv_data_copy_to_buf(). The Recovery Manager also needs to ability to parse log records in order to make sense of the log in the case of a recovery situation. This is achieved through recv_parse_log_rec(), which parses a single log record from the hash table. At any given point the Recovery Manager may wish to flush the buffer (in the case of a new checkpoint being made, etc), which is achieved through the recv_reset_buffer() function.

The Log Manager performs many auxiliary functions in order to aid in the recovery process. This includes storing checkpoint information (log_checkpoint()), switching the buffer being used at any given time (log_buffer_switch()), and flushing the buffer to disk storage (log_buffer_flush_to_disk()), which is handled through the Buffer Manager.

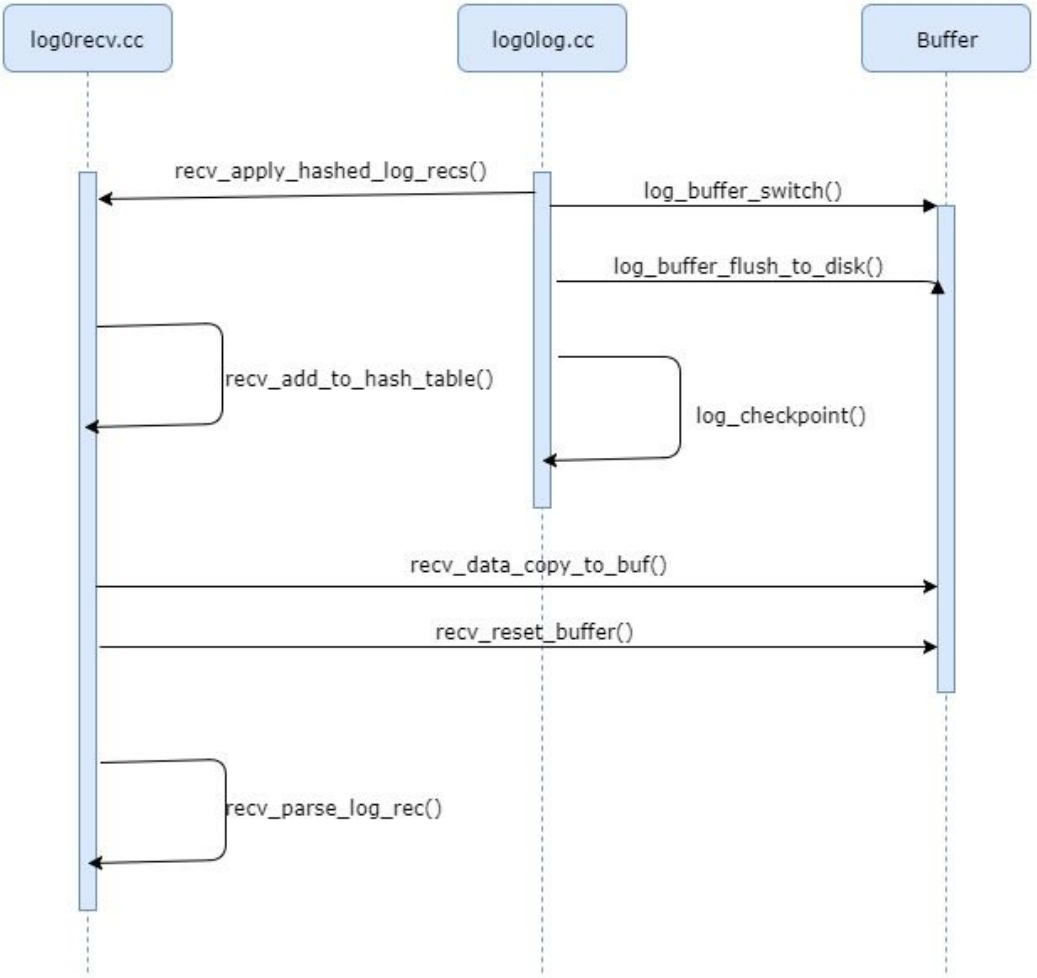

Figure 4. Showing simple interactions between the Log Manager, Recovery Manager, and the Buffer

4 Recovery Scenarios

4.1 Point-in-Time Recovery

Point-in-Time recovery is based on a database backup principle. The database administrator will set up the InnoDB recovery engine upon creating a database with backup options enabled. Binary logging must also be enabled for backups to be successfully brought back into the system. If the MySQL database goes through some form of corruption or failure, the Point-in-Time recovery system will choose the most recent, most complete saved file of the database. This file will then be executed and loaded into the database. Since changes made to the database between backup and crash vital to database operations and data, binary log information, if chosen by the database administrator, may be re-executed on the backup database to up to the most recent state possible, and hopefully up until the crash point.[1]

4.2 Recovery from Data Corruption or Disk Failure

In the event of corruption or disk failure, recovery must be done through backup that is not corrupted, similar to Point-in-Time recovery. Once a most recent, non corrupt database backup is found, it is restored back to the database using the Point-in-Time recovery method. Binary log information from backup to corruption/disk failure is re-executed. Usually, data corruption might not be that severe and would only require dropping and re-creating one or more tables. The MySQL system provides the necessary tools for finding corrupt tables or rows, although it is not guaranteed that every possible piece of corrupt data may be found.[2]

4.3 InnoDB Crash Recovery

A more serious and damaging scenario is database server crash. There are many things that can go wrong in this situation and being able to recover data or transactions on MySQL servers with high load is vital since multiple connections, reads and writes are occurring at any given time. The MySQL crash recovery system is included in the database startup scripts and the only requirement needed for crash recovery to take place is restarting the MySQL server. InnoDB will automatically check the binary logs and perform roll-forwards of the database to a more present state. What usually occurs on high traffic database systems during crash is that an event/statement was currently being processed at the time of crash. This will produce irregularities in the datatables, producing corrupt and incorrect information. InnoDB recovery will automatically roll-back these uncommitted transactions present at the time of crash. The crash recovery process is as follows:[6]

1. **Redo log application**

   This first step is performed during the initialization of the database prior to accepting any connections to the database from clients. Transactions are usually placed in the buffer pool for handling. If the buffer pool is flushed, ie. it was empty already empty at time of crash and nothing needs to be flushed, then the redo log application step is skipped. This step will also be skipped if log files are missing as there is nothing to recover.

   One may wish to remove old log information to speed up the recovery process. This is usually frowned upon even if losing some data is acceptable. Removing redo logs should only be considered upon a clean shutdown of the MySQL server.

2. **Rollback of incomplete transactions**

   Any transaction that was executed and was not able completed at the time of crash will now be re-executed and have a chance to make changes to the database. These roll back transactions cannot be cancelled once they are started. In certain cases where a shorter recovery time is needed, tools are available to the database administrator which enabling them to speed up the roll back process significantly - innodb_force_recovery.

3. **Change buffer merge**

   Any information or changes from the change buffer will now be put into the leaf pages of the secondary indexes as the main index pages are read to the buffer pool.

4. **Purge**

   Any records that have been re-executed will now be deleted to free up space and hide them from active transactions and transactions that need to be executed.

It is important to note that the steps after the redo log do not depend on the redo log and are executed in a parallel fashion to optimize resource usage and do not have to be executed one step at a time as a pipeline. ie roll backs can be executed, buffer merge change and purge can occur all while more roll backs, buffer merge changes and purges still need to take place. The most vital steps of the recovery process are contained in step 2 - rollback of incomplete transactions.[6]

5 InnoDB Recovery Management

5.1 Overview

The conceptual architecture states that within the recovery manager at the logical layer, there reside two modules: the log manager, and the recovery manager. Upon diving within the concrete architecture, it was found to be the case, and furthermore each module was compacted into its separate, singular source file. These were the known as “log0log.cc”, and “log0recv.cc”. Overall, there was little deviation from within the recovery manager architecture, but as seen from Figure 1 and Figure 2, deviations did occur. Most notably, the recovery managers are specific to storage engines. Such a concept was not unexpected as from the conceptual architecture, the recovery manager connects to the storage management’s resource manager. With different storage managers (MyISAM, InnoDB, etc.), the recovery manager would likely change. However, because the two modules are linked, there were more interactions from the recovery management module to the storage management module than the conceptual architecture showed.

5.2 log0log.cc

This file is specific to the InnoDB storage engine, and is the main file used for logging transactions done to the database. Every logged transaction is saved with a log sequence number (LSN), which provides every completed transaction with a unique identity. The LSN is also used to track the sequence in time at which each transaction was done relative to others. Thus, given a certain LSN, other logs can be identified as transactions completed either before or after. As logs are written after transactions to the database, they must be written while the server is online. Thus, the act of reading can cause concurrency issues, and so every log written requires mutex access to the database.

This file interacts with the storage management module as it records all transactions done to the database reflected by the server, so a form of direct access is preferable. While the conceptual architecture shows entrance through the resource manager, direct access to the buffer manager and storage manager improves upon speed when calling functions. Furthermore, it provides better simplicity when tracing the path that functions within log0log take, removing unnecessary detours to the resource manager module.

5.3 log0recv.cc

This file is also specific to the InnoDB storage engine, with the source code in the same folder as log0log.cc. Compiled, it acts as the main recovery tool for recovering the InnoDB database with the help of logs. To undergo recovery, the database has to set flags that indicate the previous shutdown was done so unsafely. Whenever the database restarts, those flags are read and the Crash Recovery Environment is loaded. Initialization allocates memory from the heap for storing temporary tablespaces, hash tables, and pages. Mutexes are also requested, as rebuilding requires writing to the database while the server is online.

Using the logs built by log0log.cc, the logs are scanned from the checkpoint at which the database was deemed safe, until to the point at which the database failed. The logs are then rebuilt to all be up to date, at which they are then parsed by log0recv and hashed for storage in a hash table. When all the logs are parsed and hashed, the hash tables are dumped into pages, and slowly the B-Trees used by InnoDB are rebuilt. This procedure is done for all affected B-Trees, and their respective roots nodes are re-linked into the affected tablespaces.

Upon completion, the variables within log0recv are reset, memory allocated previously in the initialization is freed, and the mutex is released.

For reasons similar to log0log, log0recv also deviates from the conceptual architecture, calling the storage manager and buffer manager directly. Speed and simplicity are improvements seen, but code reuse is an important reason. As log0recv works on rebuilding the database, direct calls to functions within the storage manager can be used to write in recovered tablespaces. Using the existing functions removes redundant code, but the trade-off is tighter coupling and thus further considerations if changes within the storage manager are made, as functions which are dependant can easily break.

6 InnoDB Replication Implementation

6.1 Overview

The heart of the replication system is based off one master server and two slaves that communicate between each other. The slaves will request copies of the binary logs from the master server and execute the transactions copied in from the master. Each slave is independent - they will read data from the master and execute transactions independently. An important feature of this design is that each slave will work at its own pace and update the database accordingly, moving on to the next set of data once the last has been completed. The

master therefore has only one job - handling requests from slaves by sending the next block of binary log data.[8]

6.2 Details

The Master-Slave replication module is implemented using three slaves: Binlog dump thread, Slave I/O thread and Slave SQL thread. The Binlog dump thread is the main thread that does the work for the master server. It will send binary log contents to the slaves for copying. This thread will acquire a lock on the master binary log for reading, as soon as a section is ready to be shipped and copied to a slave, the lock is lifted for other slave threads waiting in line. The Slave I/O thread is the slave end of the Master-Slave communication system which sends requests binary logs from the master server. This thread will accept the binary log packages and copy them as local files to the slave's relay log. Copying binary log information to slave relays takes the load off the master since the data does not need to be processed on the master and the read lock can be lifted more frequently. Finally, the Slave SQL thread is the inter-slave communication thread. This thread will read from the slave's relay log and execute the commands. [8]

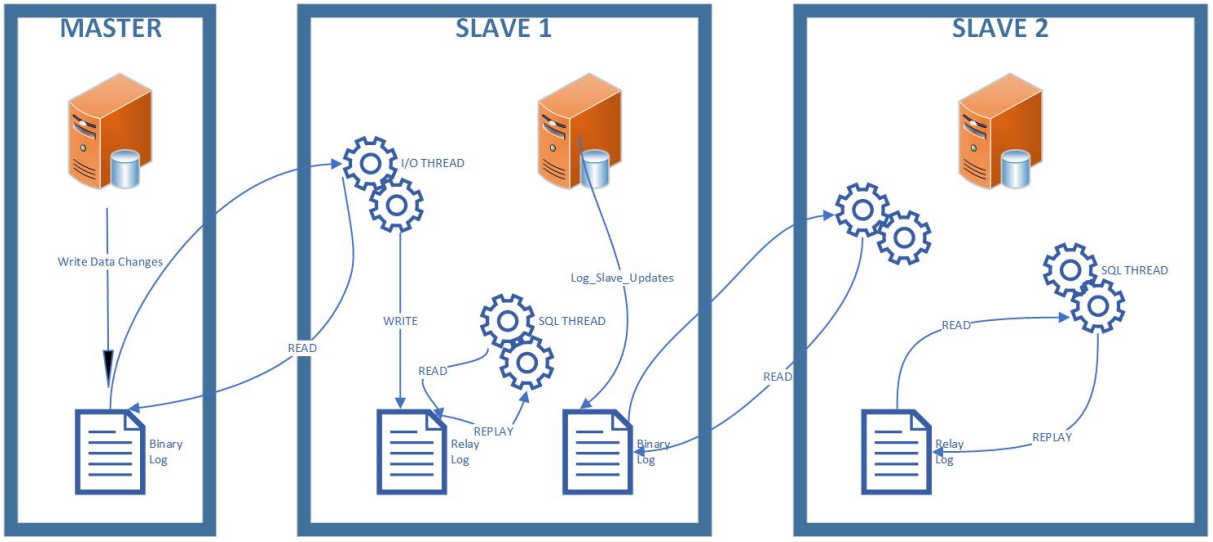

Figure 5. Master Slave Replication Process

The slave uses two threads to separate reading and executing and executing them as independent tasks. This is an attempt to remove stress from both slaves - reading is not hindered by statement execution which would slow down the whole system. [8]

7 The Binary Log (sql/binlog.cc)

The binary log is an essential part of the recovery management subsystem and performs many critically important processes in order to ensure that during the recovery process, the integrity of the data is preserved and the database can continue as if there was never a corruption or crash in the first place. The binary log contains information about

particular instances of MySQL server states. [5] These states are preserved with the use of events. Events are any statements that have been committed as transactions that change the state of the database system. This can be in the form of updates or changes to the schema of the database. It is important to note that statements such as SELECT and others than simply grab information from the database (essentially reads on the database) do not get logged as events in the binary log since they are not making changes to the state of the database. [5] Thus when the recovery process takes place, the re execution of these querying statements will not make any changes to the database and will not aid in the recovery of the database to its pre-corrupted or pre-crashed state. Once a backup is restored using whatever backup strategy is in place, the events in the binary log are re executed (since the results of their execution were not saved in the most recent backup), restoring the database system to its true original state (pre-crash / pre-corruption).

In order to ensure concurrency of writes to the binary log, mutex locks are used to ensure that if one server is writing to a particular file in the binary log, no other server can write to that file. This is required because at any point in time of the MySQL server being run, there may be several slave servers making writes to the binary log at any given time.

The binary log is not just one file, but consists of several files that make up the whole of the binary log. The servers that make up the database write to these files in binary format. The binary log serves as a mechanism to be used in the case of a recovery being needed. The contents of the binary log can be looked up manually by anyone with access to it using the mysqlbinlog command. This command lists the contents of the binary log in text readable format. [5]

As the binary log is being appended to with new events that have made changes to the database in question, backups are made periodically as a sort of checkpoint in time to where the binary log should start executing events in the case of a recovery being needed. There are several different backup strategies that can be used in MySQL. In the case of InnoDB, the default storage engine for MySQL, MySQL Enterprise Backup is used. [3] MySQL Enterprise Backup is a licensed product that performs hot backups. A hot backup consists of copying the data files of the database while the database is running (making reads and writes). [4] This backup strategy uses two steps: an initial copy of the data is made which produces a raw backup. This backup does not include the changes that may have been made while the backup was running, however. A final step is made to make these changes which produces a prepared backup. This prepared backup is finalized and is ready to be used when a recovery of the database is needed. [4] Other storage engines may use other backup strategies such as warm backups, where updates to the tables in the database are restricted while the backup is being performed. This essentially makes the tables read only while the backup is being performed. This of course is not ideal in the case of busy applications that perform many writes to the database constantly. [3]

8 MyISAM Recovery

Unlike the InnoDB storage engine, MyISAM does no support transactions and as such is unable to perform rollback or commit operations. This means that if there is a system

failure or crash that MyISAM will have only retained the data that was put into the system up to the point of failure of the system. As such, recovery for MyISAM requires the user or administrator to intervene in the recovery process through the use of the myisamchk utility. [11]

In terms of how the myisamchk utility operates, it is first important to identify how MyISAM tables are constituted. When a table is created, it is represented in the form of three files: a structure/definition file (.FRM), a data file (.MYD), and an index file (.MYI). [13] If the table is corrupt, these files are used to attempt to fix the table when myisamchk is invoked with the repair option. For best practice, it would behoove the administrator to have these tables already backed up prior to the use of myisamchk as data loss may occur if it is used under scenarios such as the server performing simultaneous tasks on the tables at the time of them being checked and repaired. [11] To prevent this, it is important to either have the MySQL server offline or to enable external lacking through the system variable "skip_external_locking" found in mysqld. [9] When invoked, myisamchk will check the targeted tables that are inputted in the command and marks if the table is corrupted. If the option of --recover or --force are added as part of the command, the utility will also attempt to repair the the tables being scanned with their respective files associated with them. [11]

There are some example scenarios in how the myisamchk can operate based on the source of which of the files associated with a table being scanned are corrupt. The first is if just the index file is corrupted, then it is the user's prerogative to first backup the data file as it will be used by myisamchk to repair the index file. The second scenario is if both the index and data files are corrupt, that the backup of the data file is first used to repair the data file and then using that to repair the index file as in the first scenario. The third scenario is if the structure file is corrupt as well, then you would need backups for both the structure and data files in order to repairs those before the data file fixes the index file. If there are no backups for the data or structure files, then myisamchk will not be able to fix this and the user will have to reconstruct the tables from the beginning. [10]

If the user wishes to automate the repair functionality of the MyISAM storage engine, they can do so by setting the "myisam-recover-options" startup option in mysqld to FORCE. To continue following the practice of backing up the data file, the BACKUP option can also be added. [12]

9 Derivation Process

In order to obtain the finalized concrete architecture of the MySQL software system, we used the iterative process of modifying the TA file and using the createContain.sh shell script to obtain the TA file used for the visualization tool LSEdit (Landscape Editor). Due to the ambiguous naming scheme used in many of the files throughout the MySQL software system, the conceptual architecture obtained from assignment 1 was first used as a sort of template to guide the placement of the various subsystems in our concrete architecture. This explains why the subsystems in Figure 1 and Figure 2 remained unchanged. Through the iterative process various high and low level subsystems discovered, which were then placed

in the conceptual subsystem that was deemed most appropriate for it according the the API description of that subsystem or file.

Further information on specific subsystems were derived by delving within the source files and reading the comments for functions and their implementations. Supporting documentation from official MySQL Source Code Documentation which outlined all functions and static variables was also used to summarize source files.

10 Lessons Learned and Conclusion

Deriving the concrete architecture of any software system purely from the source code provided is a difficult task. When dealing with an incredibly large scale software system such as MySQL, the task can seem almost impossible. Confining the focus of the software system to a particular subsystem (in this case the Recovery Management subsystem of MySQL), the task becomes simpler, but the dependencies between the particular subsystem and others makes understanding the entire software system critical to understanding the subsystem. With the aid of visualization tools such as LSEdit, the task is further simplified, but the sheer amount of subsystems, files, and dependencies makes classification of these subsystems and files daunting to say the least.

References

[1]"MySQL :: MySQL 5.5 Reference Manual :: 7.5 Point-in-Time (Incremental) Recovery Using the Binary Log", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.5/en/point-in-time-recovery.html. [Accessed: 27- Oct- 2017].

[2]"MySQL :: MySQL 5.5 Reference Manual :: 14.21.2 InnoDB Recovery", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.5/en/innodb-recovery.html. [Accessed: 27- Oct- 2017].

[3]"MySQL :: MySQL 5.7 Reference Manual :: 7.2 Database Backup Methods", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/backup-methods.html. [Accessed: 27- Oct- 2017].

[4]"MySQL :: MySQL 5.7 Reference Manual :: MySQL Glossary", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_hot_backup. [Accessed: 27- Oct- 2017].

[5]"MySQL :: MySQL 5.7 Reference Manual :: 5.4.4 The Binary Log", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/binary-log.html. [Accessed: 27- Oct- 2017].

[6]"MySQL :: MySQL 5.7 Reference Manual :: 14.18.2 InnoDB Recovery", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/innodb-recovery.html#innodb-recovery-point-in-time. [Accessed: 27- Oct- 2017].

[7]"MySQL Master-Slave Replication on the Same Machine", *Toptal Engineering Blog*, 2017. [Online]. Available: https://www.toptal.com/mysql/mysql-master-slave-replication-tutorial. [Accessed: 04- Nov- 2017].

[8]"MySQL :: MySQL 5.5 Reference Manual :: 17.2 Replication Implementation", *Dev.mysql.com*, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/5.5/en/replication-implementation.html. [Accessed: 27- Oct- 2017].

[9]"MySQL :: MySQL 8.0 Reference Manual :: 7.6.1 Using myisamchk for Crash Recovery", Dev.mysql.com, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/myisam-crash-recovery.html. [Accessed: 01- Nov- 2017].

[10]"MySQL :: MySQL 8.0 Reference Manual :: 7.6.3 How to Repair MyISAM Tables", Dev.mysql.com, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/myisam-repair.html. [Accessed: 01- Nov- 2017].

[11]"MySQL :: MySQL 8.0 Reference Manual :: 4.6.4 myisamchk — MyISAM Table-Maintenance Utility", Dev.mysql.com, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/myisamchk.html. [Accessed: 01- Nov- 2017].

[12]"MySQL :: MySQL 8.0 Reference Manual :: 5.1.4 Server Command Options", Dev.mysql.com, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/server-options.html#option_mysqld_myisam-recover-options. [Accessed: 01- Nov- 2017].

[13]"MySQL :: MySQL 8.0 Reference Manual :: 16.2 The MyISAM Storage Engine", Dev.mysql.com, 2017. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html. [Accessed: 01- Nov- 2017].