MySql Concrete Architecture (Storage Management)

Ante Pimentel (212166187)

Dusan Birtasevic (210451383)

Francis Okoyo (213811336)

Hashim Al-Helli (212172359)

Richard Van (212918652)

York University

Table of Contents

Abstract

This report focuses on the the architecture of the storage management component of the MySQL database management system. Its primary focus is to distinguish the convergences and divergences between the conceptual and concrete architecture of a system; in this case a component.

The Concrete architecture of MySQL was extracted using a pipeline style architecture recovery method. Source code level depencedes were extracted using a code extraction tool called Understand. From those dependencies, subsystem level dependences were obtained using a tool called Jgrok. Then through iterative refinement, the Concrete architecture of MySQL's storage manager was acquired.

1) Introduction and Overview

This report was prepared for EECS 4314: Advanced Software Engineering during the fall 2017 semester at York University. It illustrates the differences between the conceptual and concrete architecture of MySQL 8.0.2's storage management component; convergences (where the conceptual and concrete architecture agree on the dependencies between sub components) and divergences (where the concrete architecture adds new dependencies to other components not seen in the conceptual architecture).

The lesson to be learned from this report is why these dependences were introduced. Be it for additional functionality or for efficiency purposes. Information derived for this report was obtained by searching the MySQL documentation pages within the MySQL reference manual, performing general search queries through various academic journals and databases and utilizing source code and architecture extraction tools like Understand and Lsedit

To better illustrate the conceptual and concrete architecture of MySQL's sub components diagrams are provided. These diagrams were created using Microsoft Visio.

2) High-Level (Conceptual) Architecture

In general, the MySQL architecture can be viewed as a layered architecture where each layer communicates with an adjacent layer(never skipping layers). At the top layer, we have the Application layer which makes its possible for clients to interact with the system. Then we have the Logical layer. This layer is the “heart” of the system and is responsible for processing data. At the bottom we have the Physical layer. This layer contains storage engines. Within these layers lie components that provide different kinds of functionality which allow the system to perform certain tasks efficiently. In a previous project [1], we went into greater detail on the structure of these components and how they work.

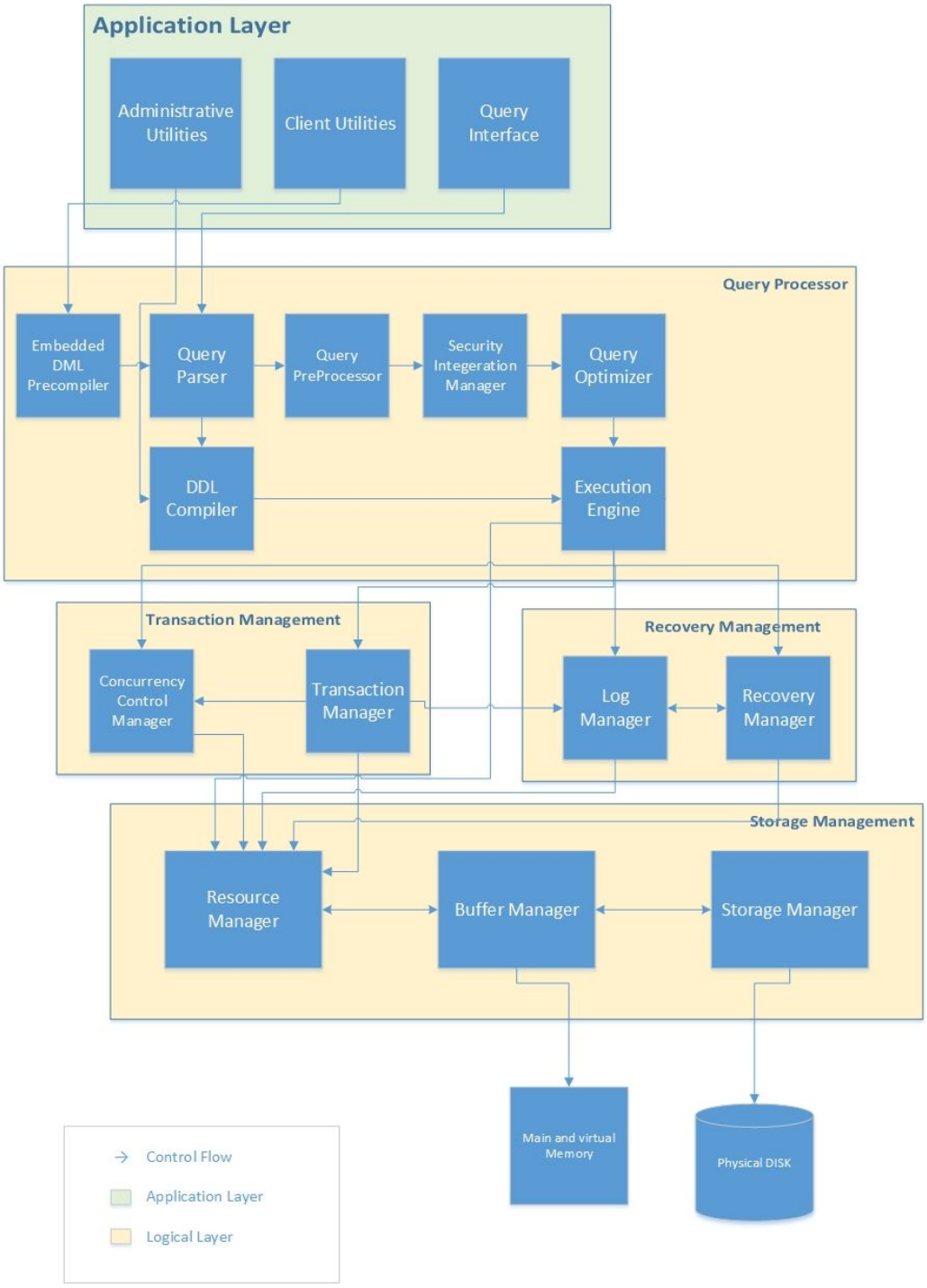

Figure 1-1. MySQL Conceptual Architecture

3) Concrete Architecture

The extracted concrete architecture of the Storage management component revealed more divergent dependences than we expected. This section will go into detail on what theses dependences are and why they were introduced to the system.

3.1) Derivation Process

After further studying the MySQL conceptual architecture and its dependencies, we acquired the MySQL v8.0.2 source code and began our architecture recovery. With the aid of architecture recovery tools like Understand, we were able to lift file level dependencies from the source code and visualize them using LsEdit. The Initial landscape of LsEdit was littered with files, subsystems and dependences. We grouped these files into their respective MySQL architectural components with the aid of documentation found in the source files and a MySQL repository on GitHub [2]. After grouping these files, we were able to structure them in a form that matched that of our conceptual architecture. From here we were able to distinguish divergent dependencies by tracing links from file to file. The LsEdit landscape before and after editing the containment file is depicted below:

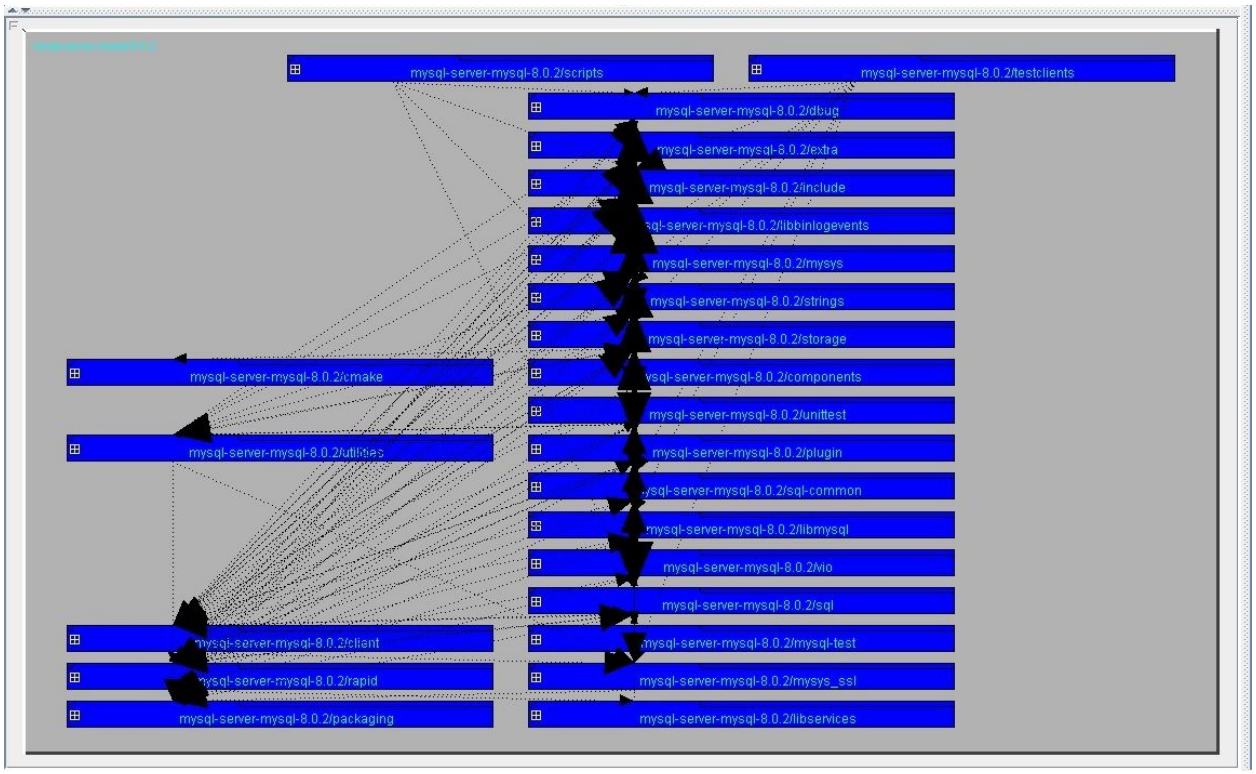

Figure 3.1-1. LsEdit MySQL Original Containment (Using Simplex Algorithm for Viewing)

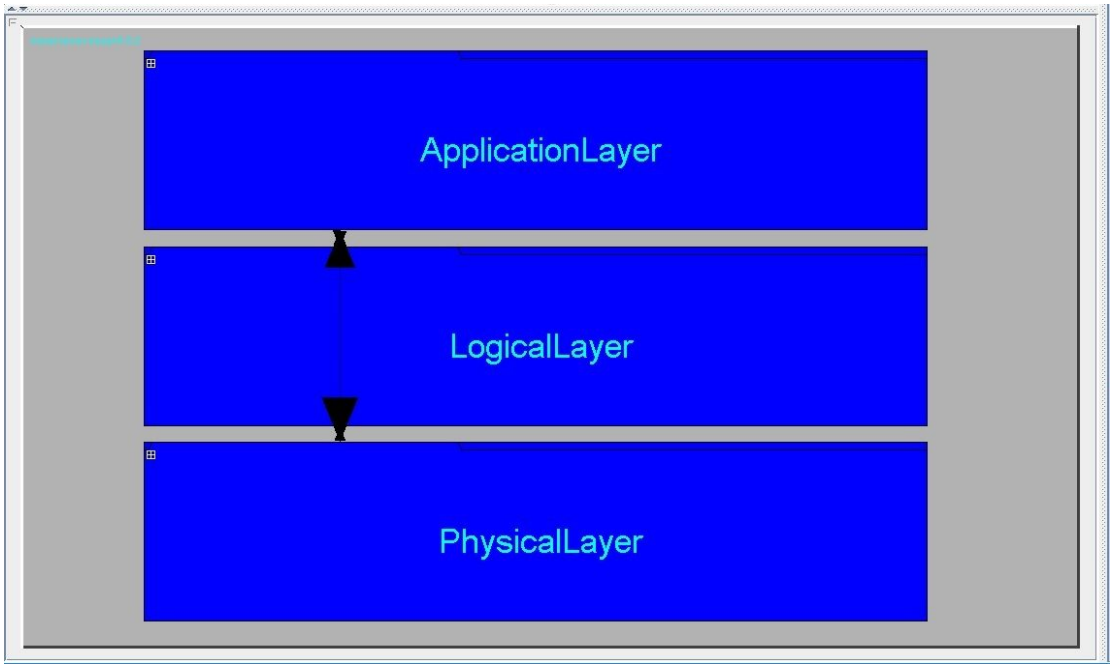

Figure 3.1-2. LsEdit Final MySQL Top-Level Containment

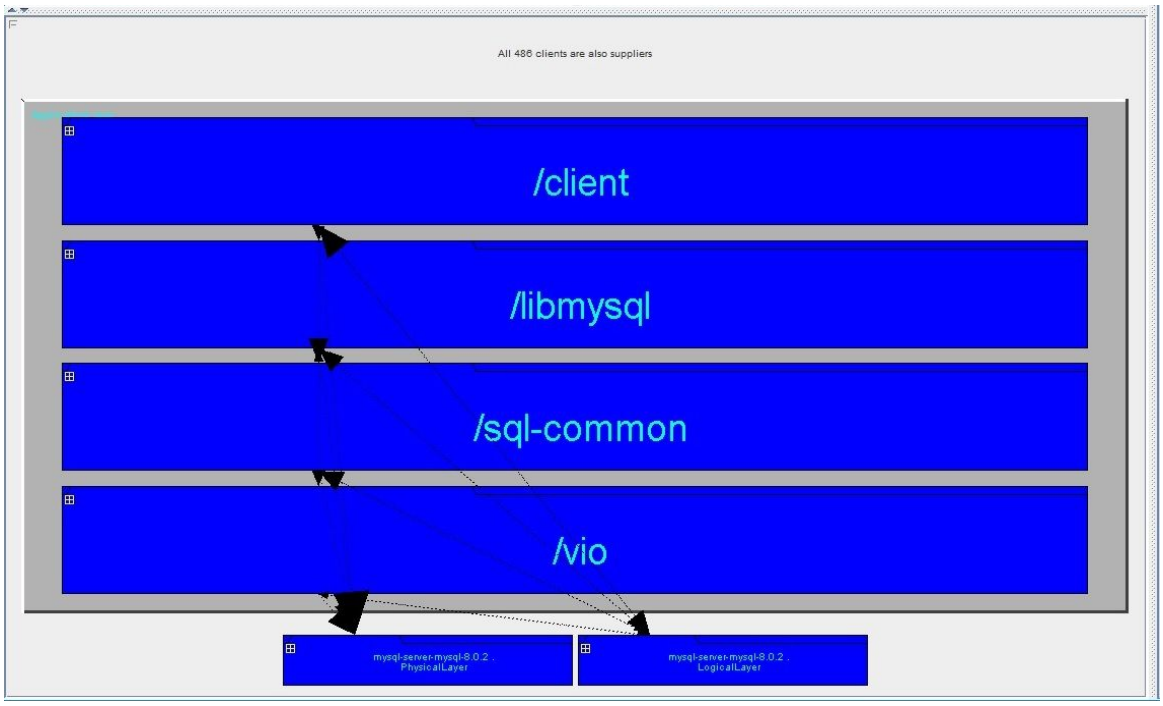

Figure 3.1-3. LsEdit Final Application Layer Containment

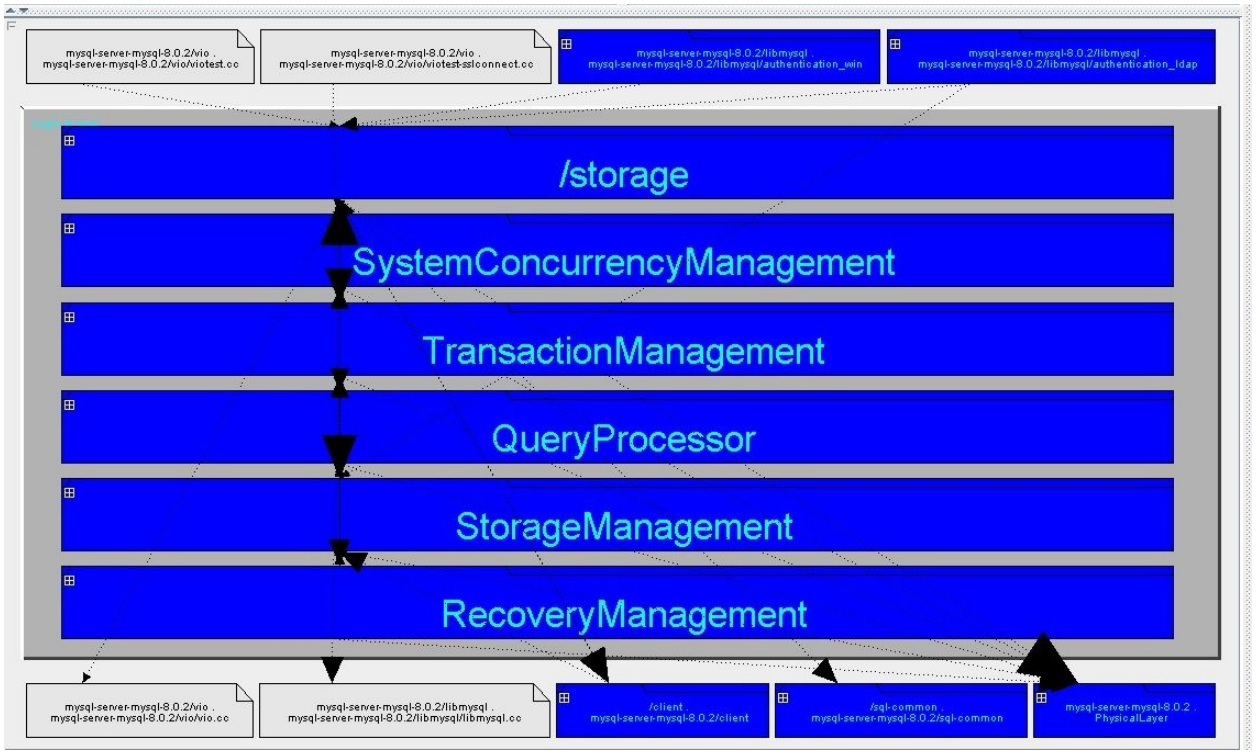

Figure 3.1-4. LsEdit Final Logical Layer Containment

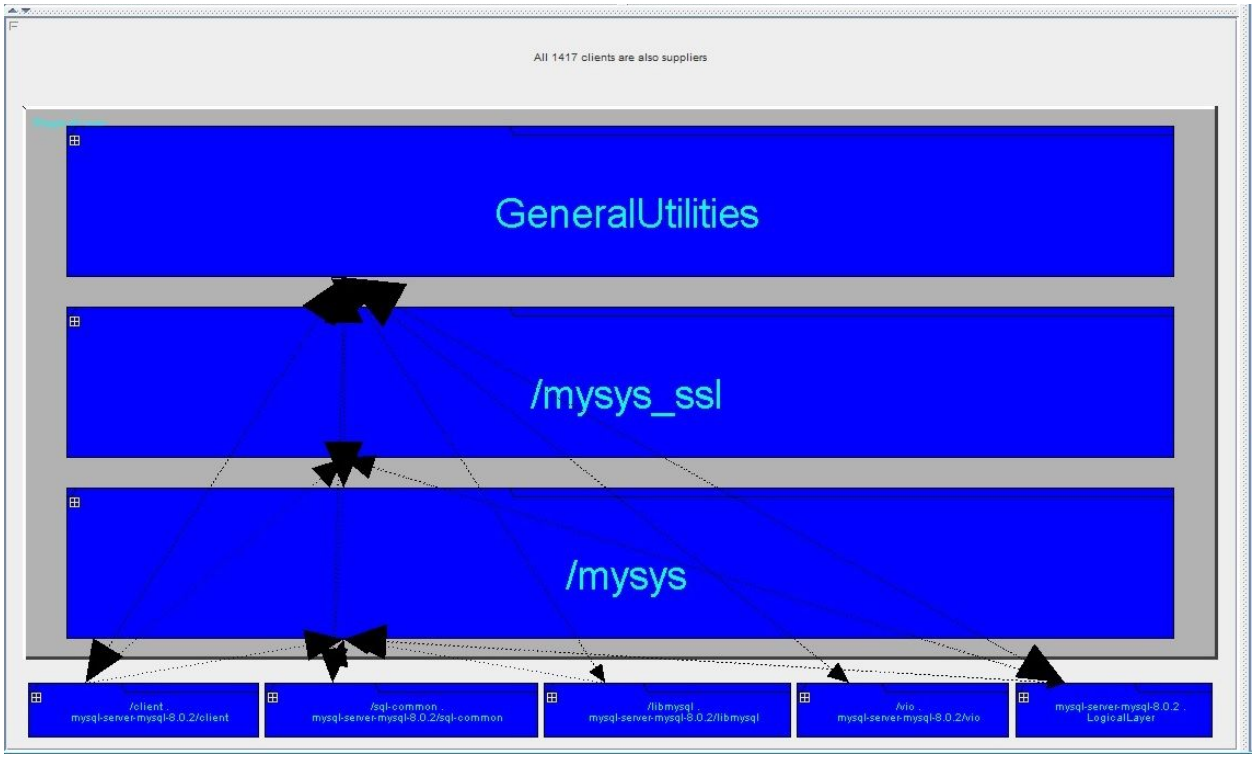

Figure 3.1-5. LsEdit Final Physical Layer Containment

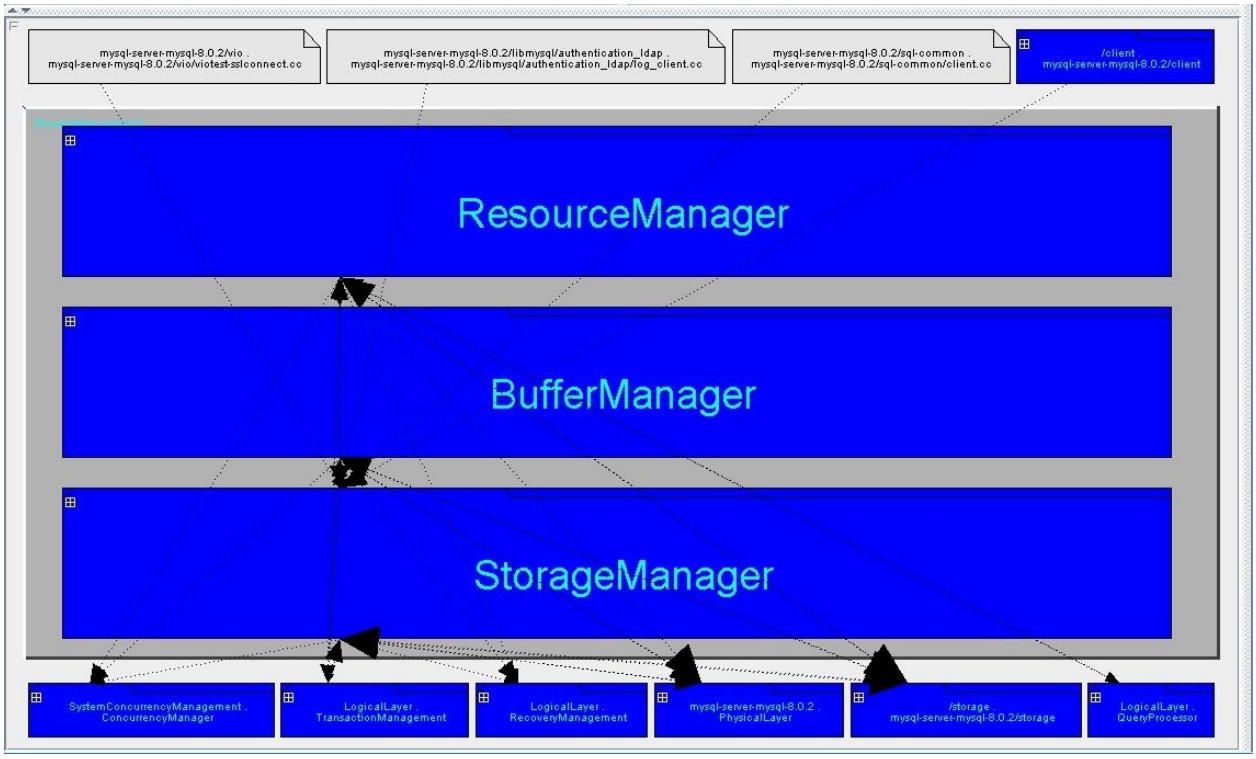

Figure 3.1-6. LsEdit Final Storage Management Containment

When all divergent dependences were isolated, we went into each file, looking at the source code and documentation. This was done to understand why these dependences were introduced. A few of the dependences we discovered were introduced to fix bugs in the system. The remainder either lacked sufficient documentation to distinguish why they were implemented or were in the system from the beginning. The way we determined this was by going into the documentation and source code of previous MySQL versions, searching for information.

3.2) Storage Management Subsystem and Interactions

Within the logical layer of the MySQL architecture lies the storage management component. This is responsible for reading and writing data to secondary storage. Inside this component are the resource, buffer and storage managers.

3.2.1) Resource Manager

The responsibility of the Resource Manager is to accept requests from the transaction and log manager and translate them into table format, which can be understood by the Buffer Manager. The resource manager is implemented inside the

dict [6] directory. The files inside this directory form the Resource Manager. These files implement all data dictionary (metadata) functionality.

It is observed from the Figure 4-1 that the resource manager interacts directly with the storage manager without going through the buffer manager. This divergence was caused by refactoring and bug fixing.

3.2.2) Buffer Manager

The Buffer Manager is responsible for efficiently storing data in memory for manipulation. It accepts table requests from the Resource Manager and decides how much memory to allocate in order to complete the request. The buffer manager is implemented in the buf [3] folder. The file buf0buf.cc file represents the buffer pool where the file pages are loaded into memory blocks known as buffer frames. In addition, buf0flu.cc contains the functionality to flush the buffer and buf0lru.cc contains the replacement algorithm, which decides which blocks should be shifted back to disk. Finally there is a file called buf0rea.cc, which calls the storage manager to initiate a file read.

3.2.3) Storage Manager

The responsibility of the Storage Manager is to provide fast read/write access to tablespaces and logs of the database. A tablespace consists of database pages whose default size is set to 16kb these pages are then grouped into segments of 64 consecutive pages [4]. To acquire more speed in disk transfers, a technique called disk striping is employed. The implementation of the Storage Manager can be found in the fil [4] and fsp [5] directories in the MySQL v8.0.2 directory. The fil0fil.cc file in the fil directory contains the implementation of the low-level file system. This includes structures and methods needed to manipulate them. On the other hand the fsp0fsp.cc file which is found inside the fsp directory takes care of the file space management. i.e. handles the allocation of pages to files and keeps track of which files are used, open, and closed.

4) Architectural Divergences

This section highlights the differences between the conceptual (Figure 1-1) and concrete architecture (Figure 4-1) of MySQL Storage management. The overall architecture remains the same, with the exception of a few architectural violations. The majority of these violations are of the layered and pipeline style architecture. Instead of data flowing from component to component, it bypasses the “middle man” components and goes straight to its destination. Example: Figure 4-1; Resource manager to Storage Manager and the reverse.

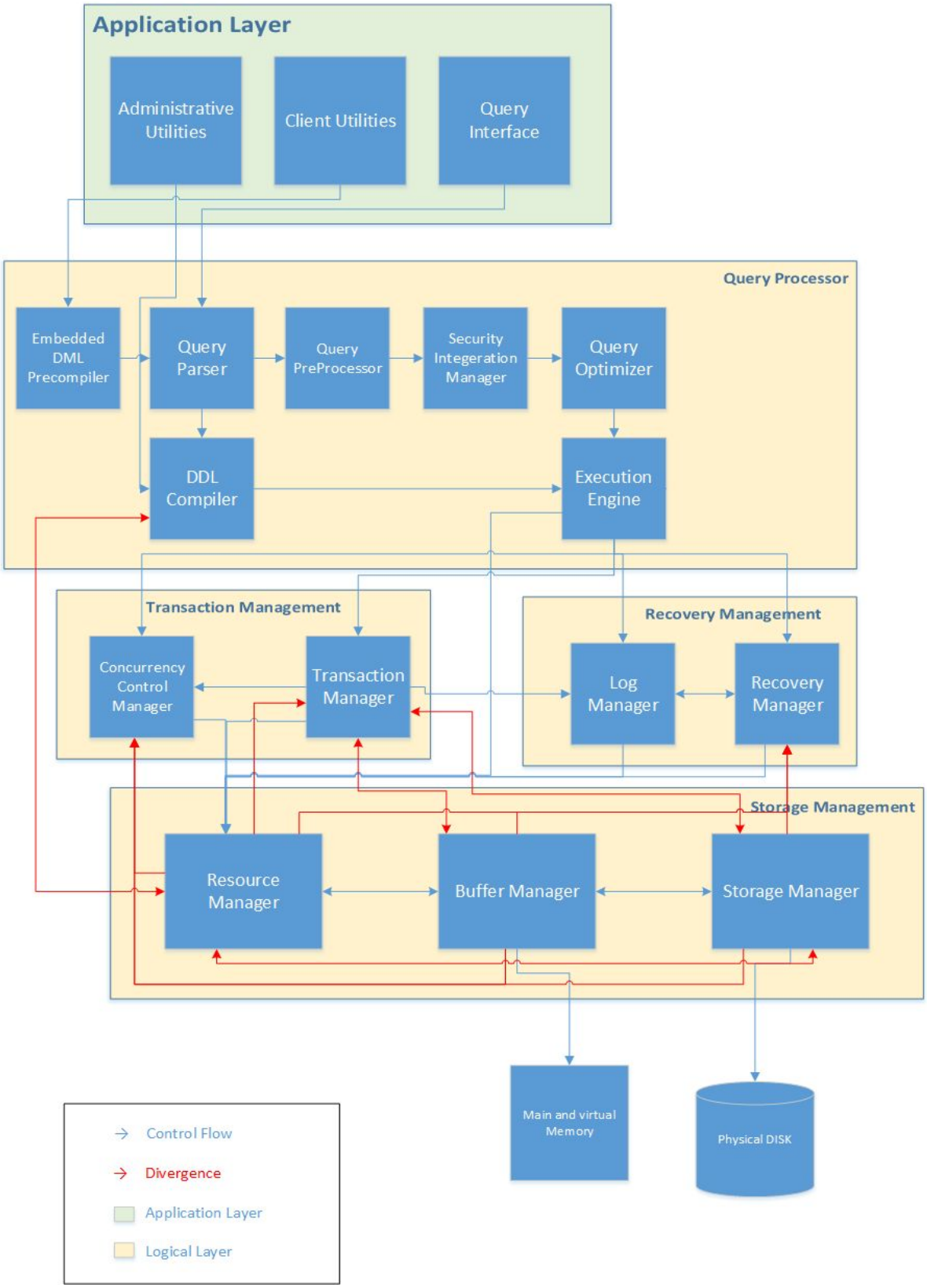

Figure 4-1. Storage Management Concrete Architecture

4.1) Divergences

Figure 4-1 shows the concrete architecture of the storage management subsystem in the context of the overall conceptual architecture (Figure 1-1). There were no extra subsystems or missing dependencies found during the investigation. Below is a list of the 15 divergences found using sticky notes, along with the files that cause them and a reason why the divergence might be there.

1.

Divergence:	**Resource Manager to Storage Manager**
Cause:	**dict0dict.cc calling fsp0fsp.cc via fil_make_filepath**
Reason:	**Was added Jan 17, 2014 by developer Kevin Lewis.**
Note:	**The update was part of some refactoring being done in the dict0dict.cc file.**

2.

Divergence:	**Storage Manager to Resource Manager**
Cause:	**fil0fil.cc calling dict0mem.cc via dict_mem_create_temporary_tablename**
Reason:	**Was added Sept 5, 2012 by developer Jimmy Yang. The update was part of a larger bug fix.**
Note:	**Involves online DDL preventing further DDL on a table.**

3.

Divergence:	**Resource Manager to Concurrency Control Manager**
Cause:	**dict0dict.cc calling lock0lock.cc via lock_table_has_locks**
Reason:	**Resolving conflicts caused by few files.**
Note:	**N/A**

4.

Divergence:	**Storage Manager to Transaction Manager**
Cause:	**fsp0fsp.cc calling trx0sys.cc via trx_sys_undo_spaces**
Reason:	**Tables being upgraded from 5.7 may have a list of old-style undo tablespaces.**
Note:	**This helps check if they are there.**

5.

Divergence:	**Buffer Manager to Transaction Manager**
Cause:	**buf0buf.cc calling trx0sys.cc via references**
Reason:	**buf0buf.cc has some references to trx0sys.cc in order to get some data about a transaction.**
Note:	**Buffer pages are monitored based on their subsystem.**

6.

Divergence:	**Resource Manager to Recovery Management**
Cause:	**dict0load.cc calling trx0trx.cc via references**
Reason:	**dict0load.cc leaves references to a specific transaction using type definitions from trx0trx.cc.**
Note:	**N/A**

7.

Divergence:	**Storage Manager to Recovery Management**
Cause:	**fil0fil.cc calling log0recv.cc via recv_calc_lsn_on_data_add**
Reason:	**Calculating the log sequence number when more data is being written to the log.**
Note:	**N/A**

8.

Divergence:	**Buffer Manager to Recovery Management**
Cause:	**buf0buf.cc calling log0recv.cc**
Reason:	**Is there since the start. Buffer pages are monitored based on their subsystem.**
Note:	**N/A**

9.

Divergence:	**Resource Manager to Query Processor**
Cause:	**dict0crea.cc calling pars0pars.cc via pars_info_create**
Reason:	**dict0crea.cc makes use of the parser info struct in pars0pars.cc**
Note:	**N/A**

10.

Divergence:	**Query Processor to Resource Manager**
Cause:	**pars0sym.cc calls dict0dict.cc via variables**
Reason:	**pars0sym.cc makes use of the dictionary system created in dict0dict.cc**
Note:	**N/A**

11.

Divergence:	**Transaction Manager to Buffer Manager**
Cause:	**trx0sys.cc calls buf0dblwr.cc via trx_sysf_create**
Reason:	**trx0sys.cc uses the double write buffer at database creation.**
Note:	**N/A**

12.

Divergence:	**Transaction Manager to Storage Manager**
Cause:	**trx0sys.cc calls fsp0fsp.cc via fseg_create**
Reason:	**the transaction system uses fseg_create to create headers for buffer segments.**
Note:	**N/A**

13.

Divergence:	**Resource Manager to Transaction Manager**
Cause:	**dict0load.cc calls trx0trx.cc via trx_allocate_for_background**
Reason:	**dict0load.cc uses this to create a transaction object when replacing records in SYS_TABLESPACE and SYS_DATAFILES**
Note:	**N/A**

14.

Divergence:	**Buffer Manager to Concurrency Control**
Cause:	**fsp0fsp.cc calls sync0rw.cc via fseg_alloc_free_page_low**
Reason:	**fsp0fsp.cc uses sync0rw.cc to create buffer blocks, which contain a lock for synchronization.**
Note:	**N/A**

15.

Divergence:	**Storage Manager to Concurrency Control**
Cause:	**dict0stats.cc calls sync0rw.cc via dict_stats_exec_sql**
Reason:	**dict0stats.cc uses sync0rw.cc to execute sql statements.**
Note:	**N/A**

5) Lessons Learned

Recovering the concrete architecture of a system can be very complex; even with the aid of architecture recovery software. The major problem we had was navigating the systems dependencies. When you have a large scale system like MySQL, you will find that there are a lot of code and file level dependencies between directories. Through iterative refinement, detailed documentation and a substantial amount of determination, we were able to extract the concrete architecture from the MySQL source code.

6) Use Case

We will do simple flow of of an Insert statement and which directories are involved. User enters insert statement then the message is transferred over TCP/IP which is the vio folder. The Server then parses the statement done through the sql directory. The server calls the low level functions associated with the storage engine Handler that is choosing which storage engine is set then the storage engine calls mysys folder which deals with the low level details of storing the file on disk.

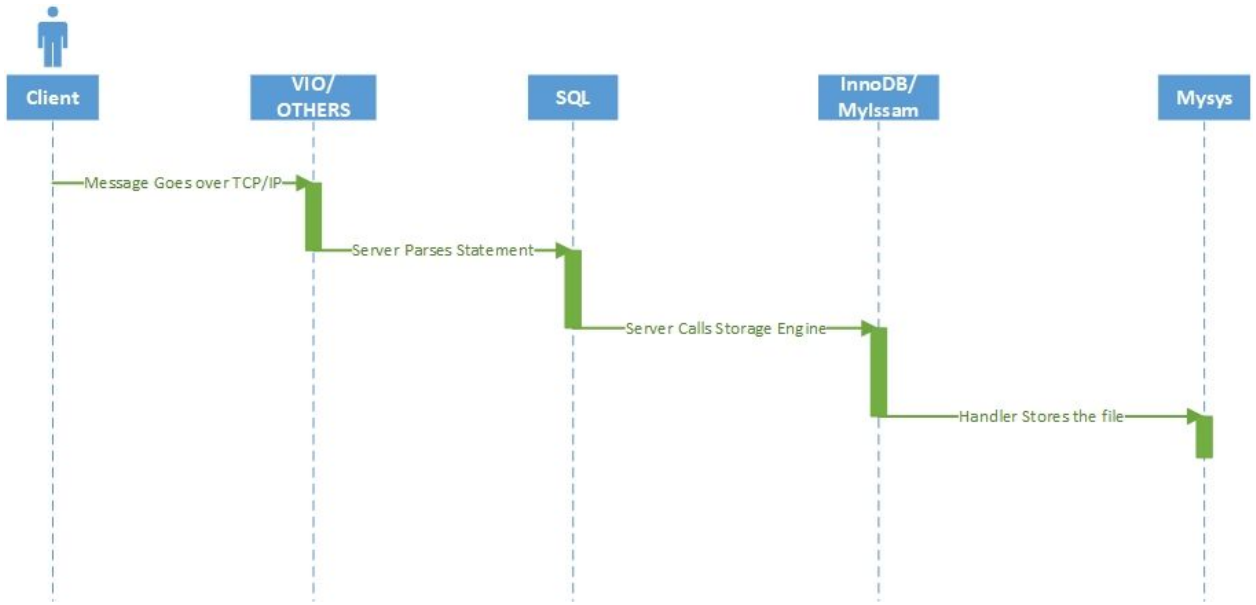

Figure 6-1. Insert statement sequence diagram

7) References

[1] MySql conceptual architecture.
https://github.com/hashim93a/eecs4314MYSQL/raw/master/Assignments/Assignment1/Conceptual_Architecture_Report.pdf

[2] MySQL source code repository.
https://github.com/mysql/mysql-server/tree/8.0

[3] MySQL buf/ directory
https://github.com/mysql/mysql-server/tree/8.0/storage/innobase/buf

[4] MySQL fil/ directory
https://github.com/mysql/mysql-server/tree/8.0/storage/innobase/fil

[5] MySQL fsp/ directory
https://github.com/mysql/mysql-server/tree/8.0/storage/innobase/fsp

[6] MySQL dict/ directory
https://github.com/mysql/mysql-server/tree/8.0/storage/innobase/dict

[7] MySQL Internals Manual
https://dev.mysql.com/doc/internals/en/guided-tour-majordir-client.html