MySQL Concrete Architecture

EECS 4314: Advanced Software Engineering
Tabs vs. Spaces
November 6, 2017

Authors

Kevin Arindaeng

Anton Sitkovets

Glib Sitiugin

Nisha Sharma

Varsha Ragavendran

Ayman Abualsunun

Davood Anbarnam

Table of Contents

List of Figures

List of Tables

Abstract

This report presents an overview of the concrete architecture of the Query Processor subsystem of MySQL 8.0.2. An introduction and overview of its high level concrete architecture is presented as well as its derivation process. A discussion on the use of the dependencies, comparison towards the conceptual architecture, and a reflexion analysis of each major subsystem component is provided. A use case is provided to present the interaction between the user and the Query Cache and Security Integration Manager subsystem. Lastly, a conclusion and lessons learned during the creation of the concrete architecture is shown. Diagrams produced in LSEdit were redrawn using draw.io.

Introduction and Overview

This report presents the concrete architecture of the Query Processor subsystem within the logical layer of MySQL 8.0.2. Specifically, this report will outline the significance of the subsystems present in the Query Processor, as well as the derivation process of the concrete architecture, and the comparison between the concrete architecture findings and the conceptual architecture. The Query Processor has several subsystems which takes in SQL queries as input and determines the execution plan for the queries, provided the queries are syntactically and semantically valid, and finally executes the queries in an optimized manner.

The report begins with an analysis of the architecture of the Query Processor, identifying the subsystems and illustrating the interactions between these subsystems. Each subsystem is examined further in detail, providing information regarding the functionality and the rationale of their interactions with other subsystems. Consideration of the key differences between the conceptual and concrete findings is taken.

Beginning with the Query Parser subsystem, the report identifies the interactions involved between the Parser and other subsystems when converting a SQL query into a Parse Tree. Followed by the Query Preprocessor, which performs semantic validation on the parse tree to ensure integrity checks are in place before optimizing a SQL query. The Query Optimizer generates multiple efficient execution plans. Finally the Execution Engine, following the suggested efficient execution plan, executes the query. An example scenario is presented, illustrating the workflow between the Query Processor's various subsystems. To close out this section of the report, a glossary for various terms/phrases used throughout the document as well as any naming conventions used are

presented. The report concludes with the major findings of the Query Processor subsystem, and lessons learned from putting together this document.

Derivation Process

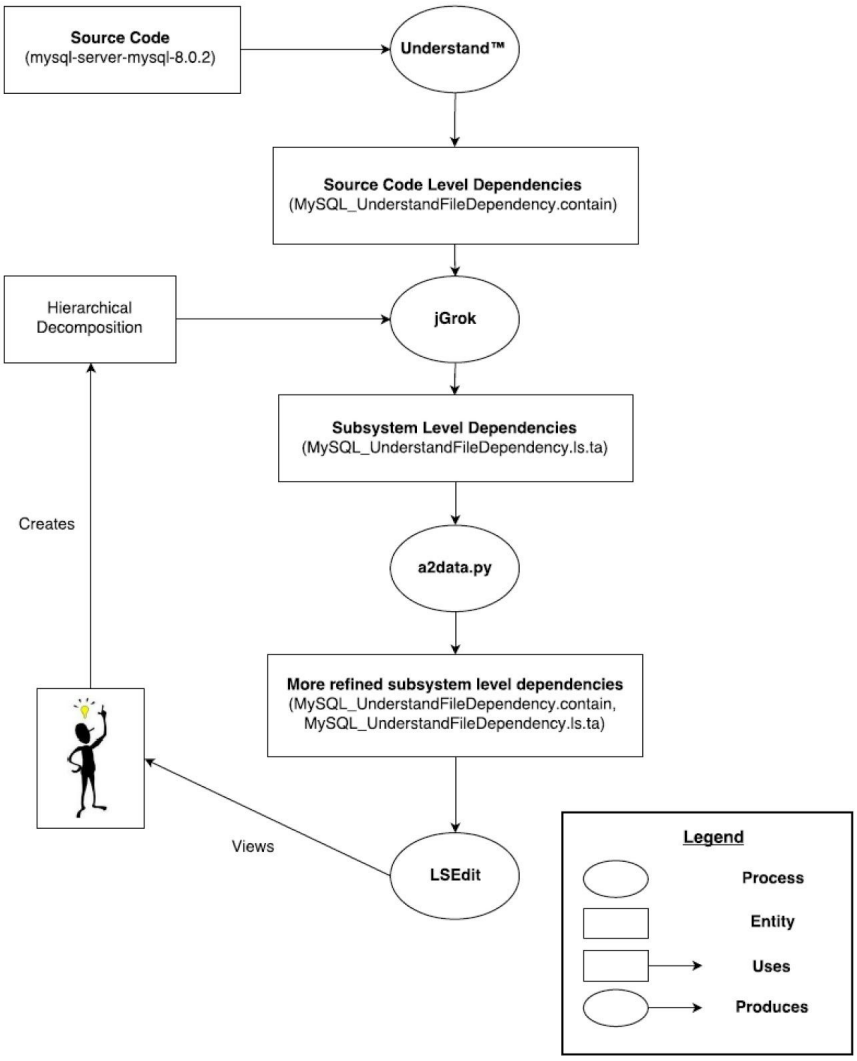


Figure 1. Derivation Process (drawn using draw.io)

A derivation process was required to determine the concrete architecture of the query processor of MySQL. This is mainly how each file from the MySQL Server 8.0.2 source code is grouped into concrete subsystems. This is required since the size of MySQL is very large, with more than 1.5 million lines of code.[1] It would not be practical to have a manual derivation process. A more automated one using specific tools is used as shown in figure 1. It is described in more detail below.

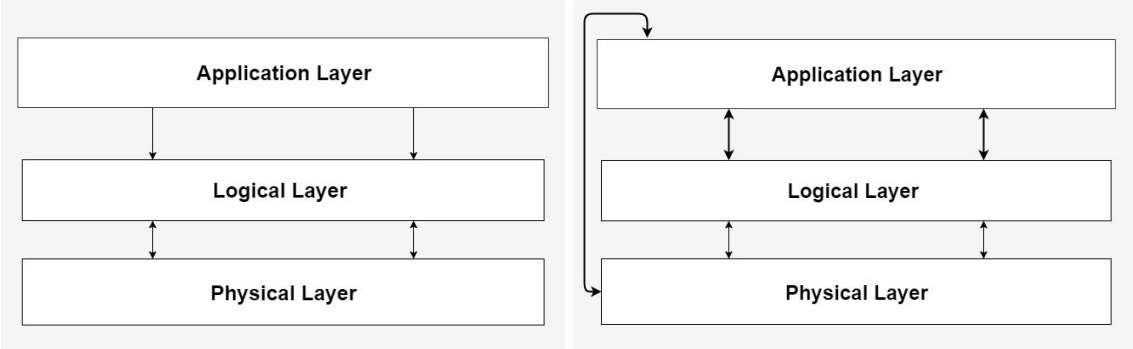
From figure 1, using Understand™ on the MySQL source code creates a .contain file with source code level dependencies. It provides a link to the directory of each source file. From here, a tool called jGrok is used to go through each file and find its dependencies to other files in the source folder. This creates many subsystem level dependencies in the form of a .ls.ta file.

A script called a2data.py was created using Python to help group the dependencies into higher level subsystems based on the Query Processor's conceptual architecture, and filter out any dependencies that were unrelated to the Query Processor.[2] Using a visualization tool called LSEdit, and after multiple iterations of the derivation process, the subsystem hierarchy of the query processor could be determined as seen in figure 4.

Architecture of the Query Processor

Overview

Retrieving the concrete architecture by using above mentioned derivation process and analyzing the MySQL 8.0.2 source code led to similar layered architecture as observed for the conceptual architecture but with some changes in dependencies. Figure 2 below compares conceptual and concrete architecture. It was noticed that Application Layer shares two way dependency with Logical Layer instead of one-dependency. Also a new two way dependency was discovered between Application Layer and Physical Layer. Clearly, MySQL architecture is slightly different from standard Layered Architecture.



(a) Simpler Conceptual Architecture (b) Simpler Concrete Architecture

Figure 2. Overall Architecture of MySQL (drawn using draw.io)

The concrete architecture analysis reveals that there exist some new dependencies between different components and subsystems instead of one

way dependencies. First, the concrete architecture of Query Processor, which is the system under analysis, has dependencies changed from the conceptual architecture. We discovered a lot more dependencies between subsystems of Query Processor. Figure 3 below depicts the conceptual architecture of the Query Processor, while Figure 4 reflects all the differences in retrieved concrete architecture. Bolded arrows indicate new/modified dependencies, and non-bold arrows with solid lines represent dependencies that stay the same while the ones with dashed lines are lost. It should also be noted that the subsystem “Embedded DML Pre Compiler” is now obsolete (represented as box with dashed border and grey title) and has been replaced by DDL Compiler and Lex (represented by boxes with red border). Also, Parse Tree is added as an extra subsystem for understanding the interaction between different components of the code implementation.

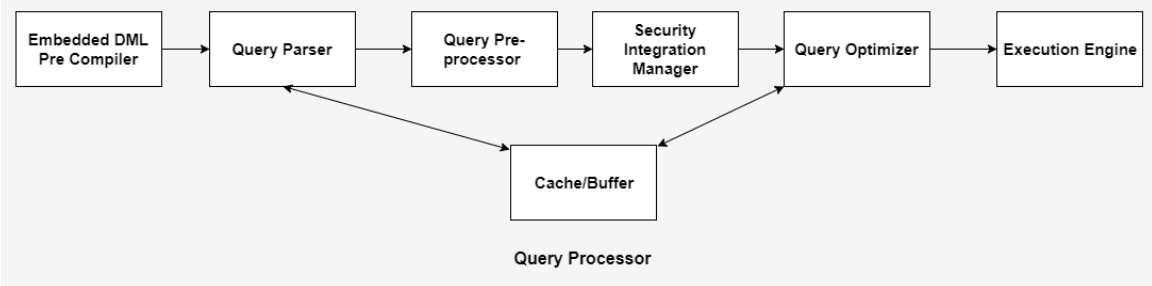


Figure 3. Query Processor Conceptual Architecture (drawn using draw.io)

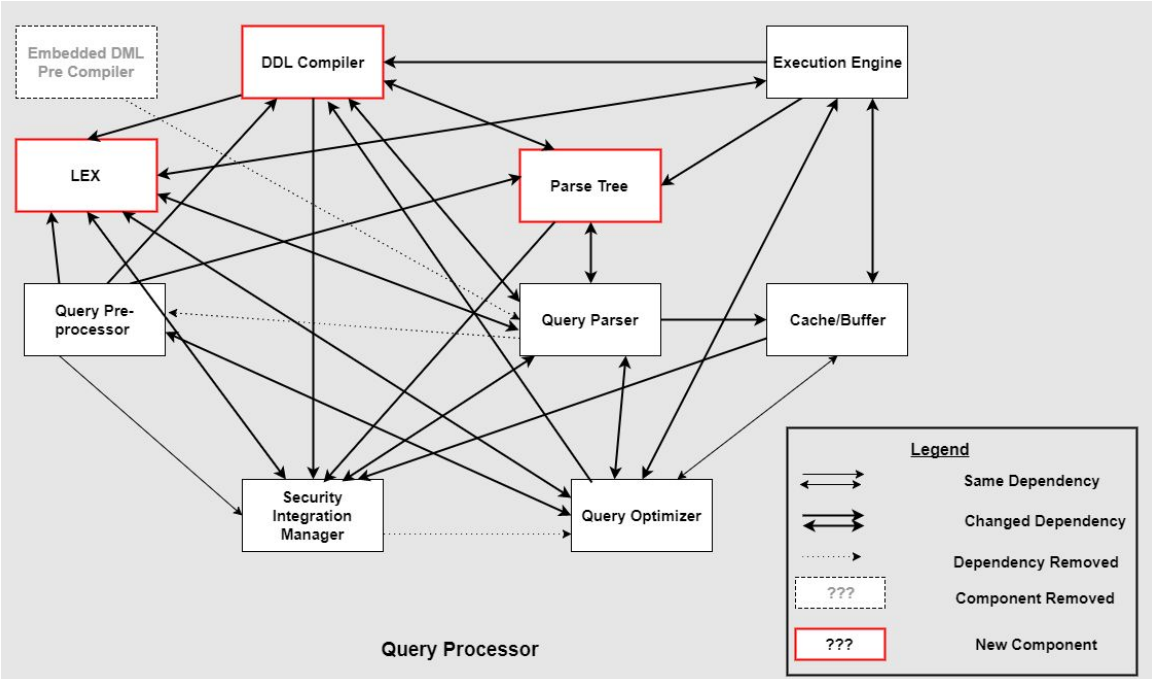


Figure 4. Query Processor Concrete Architecture (drawn using draw.io)

Dependencies that stayed the same are the one way dependency between Query Preprocessor and Security Integration Manager, and a two way dependency between the Cache/Buffer and Query Optimizer. All changes in dependencies are discussed in details in relative sections with a discussion of the possible reasons.

DDL Compiler

The first subsystem in the concrete architecture of the query processor is the DDL Compiler. This compiler converts DDL statements into machine or object code for MySQL to use. DDL statements are SQL commands to modify or alter the database structure. Some example statements are CREATE, ALTER, DROP, RENAME, TRUNCATE. An example of a CREATE statement for a fictional athlete for is shown below.

```
create table PLAYER
(
    player#          char(2) not null,
    call             char(2) not null,
    name             char(30),
    dob              date,
    since                    date,
    primary key      (player#, call),
    foreign key      (call) references TEAM(call)
);
```

The concrete architecture for the DDL Compiler and its dependencies to other is seen in figure 5.

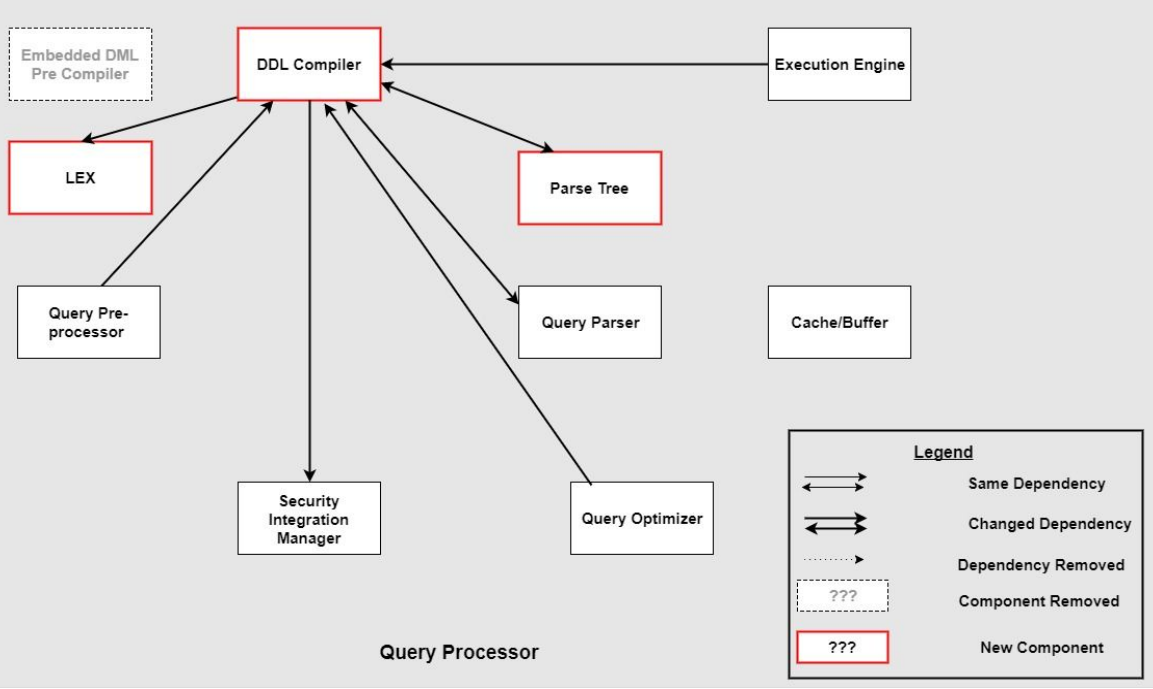


Figure 5. Concrete Architecture of the DDL Compiler (drawn using draw.io)

The DDL Compiler subsystem is not found when comparing figure 5 to the conceptual architecture in figure 3. Upon the creation of the conceptual architecture, it was assumed the Embedded DML Precompiler was part of the DDL Compiler. Upon the derivation process, it was found that it was its own separate subsystem. This is because the DDL Compiler handles query syntax for creating and managing the database, whilst the DML (Data Manipulation Language) compiler handles query syntax for manipulating the data in the database, such as a SELECT query.

The DDL compiler has dependencies on the Lex, Parse Tree, Query Parser, and Security Integration Manager subsystems. It has a dependency to Lex because it needs to read tokens from the parse tree. It has a dependency to the parse tree to obtain the proper syntax for SQL statements. It has a dependency to the Query Parser to pass compiled SQL statements to the parser, and it has a dependency to the Security Integration Manager to ensure that the user has correct permissions to run DDL commands. One dependency that was not in the conceptual architecture was the DDL Compiler towards the Security Integration Manager. A reflexion analysis is provided below.

DDL Compiler → Security Integration Manager

Which?	Alter_info alter_info() (in .../sql/sql_cmd_ddl_table.cc) *depends on* create_table_precheck() (in .../sql/auth_auth.common.h
Who?	GlebShchepa
When?	Committed on Jul 14, 2016, 1:48PM EDT
Why?	True bottom-up server parser: refactoring of the CREATE TABLE statement WL#8434: True bottom-up server parser: refactoring of partitioning-related stuff WL#8435: True bottom-up server parser: cleanup and refactoring column definition stuff WL#8433: Separate DD commands from regular SQL queries in the parser grammar WL#7840: Allow parsing a single expression

Table 1. Reflexion Analysis for the DDL Compiler (history from Github)

The reasoning behind this change seems to mainly originate from refactoring purposes. Upon the introduction of MySQL 8.0.2, it relies now on the security

integration manager to do a pre-check whenever the user tries to create a table, to make sure the user has proper permissions.

Query Parser

After digging more into the parser subsystem, it was clear that the Parser uses the interpreter design pattern. Interpreter design pattern is a behavioral design pattern, which provides a way to evaluate language grammar or expressions. This pattern involves implementing an expression interface that interprets SQL queries. Moreover, the LEX subsystem that is being mainly used by the parser implements the visitor design pattern, and it has a separate file called select_lex_visitor-t.cc. The concrete architecture of the query processor follows a pipe and filter style as it is identified in the conceptual architecture. However, there are new subsystems that were discovered with new dependencies as shown in the figure 6.
The Parser uses a Parse Tree which wasn't clearly identified in the conceptual architecture and it was assumed to be within the parser itself. The parser dependency on the parse tree is mainly caused by the Parse_tree_helper and the Parse_tree_hints classes, which helps the query to be tokenized. Moreover, the Lex subsystem was also assumed to be within the parser itself, not a stand alone subsystem. The parser depends on the Lex duo to the sql_lex_visitor class which helped in identifying the visitor pattern within the parser subsystem.

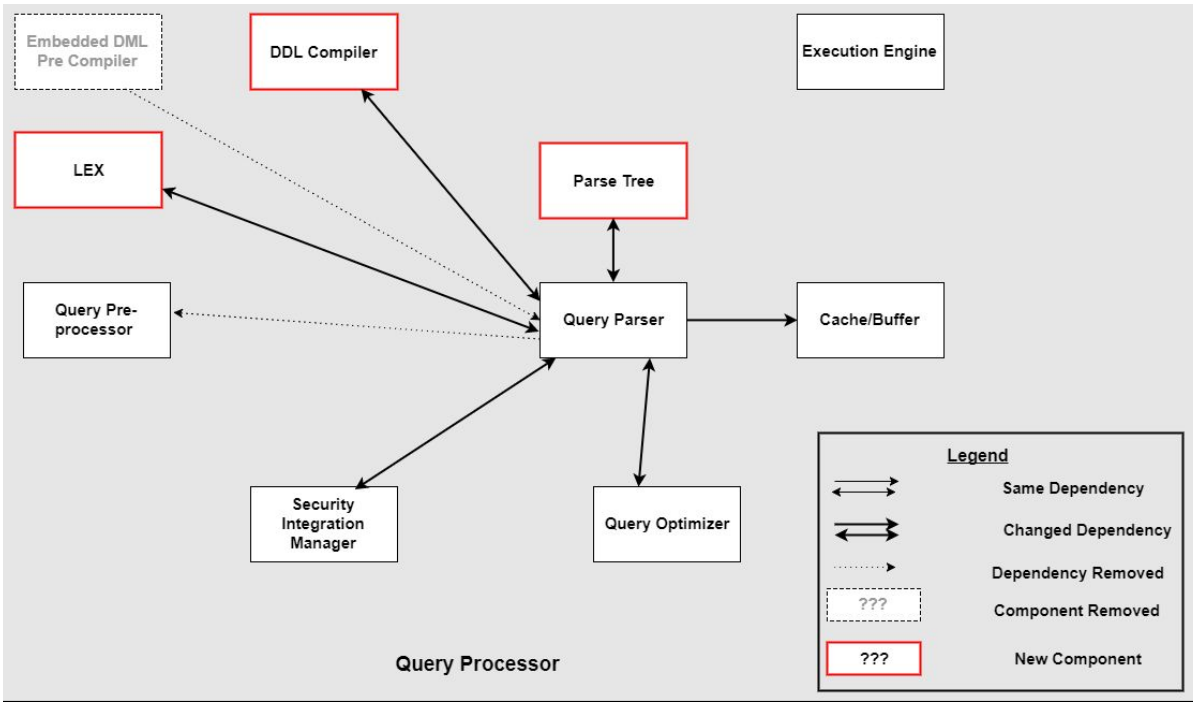


Figure 6. Concrete Architecture of the Parser (drawn using draw.io)

One dependency that was not in the conceptual architecture was the Query Parser towards the DDL Compiler. A reflexion analysis is provided below.

Query Parser → DDL Compiler

Which?	Query_result() (in .../sql//sql_parse.cc) depends on execute_show() (in .../sql/query_result.h)
Who?	Jon Olav Hauglid committed with Tor Didriksen
When?	Commited on Feb 11, 2015, 4:15AM EST
Why?	Bug#20302351: MOVE SELECT_RESULT_INTERCEPTOR SUBCLASSES FROM SQL_CLASS.H Follow-up patch: Rename select_result and decendants to Query_result Note: This is a re-push of a patch reverted by mistake. The original patch went into 5.7. This patch is now 5.8 only. (cherry picked from commit 6a1752c32919ba1e1779ace1e7561c5558d883f0) Conflicts: sql/sql_derived.cc

Table 2. Reflexion Analysis for the Query Parser (history from Github)

The reasoning behind this change seems to mainly originate from refactoring purposes in terms of renaming. Instead of having the relevant subclasses in sql_class.h, the method was moved to sql_parse.cc, which is what the DDL Compiler depends on. The information on the bug could not be found, but it is assumed that having this subclass in sql_parse.cc caused problems, which is why it was moved.

Query Preprocessor

Query Preprocessor performs additional semantic validation on the parse tree, which the parser is not able to do, such as verifying table integrity, column existence, etc.[3] It has other validations which are:

- Verify that the relation and attribute names specified in the query exist in the database schema.
- Verify that attribute names have a corresponding relation name specified in the query
- Check types of the attributes when comparing with constants or other attributes

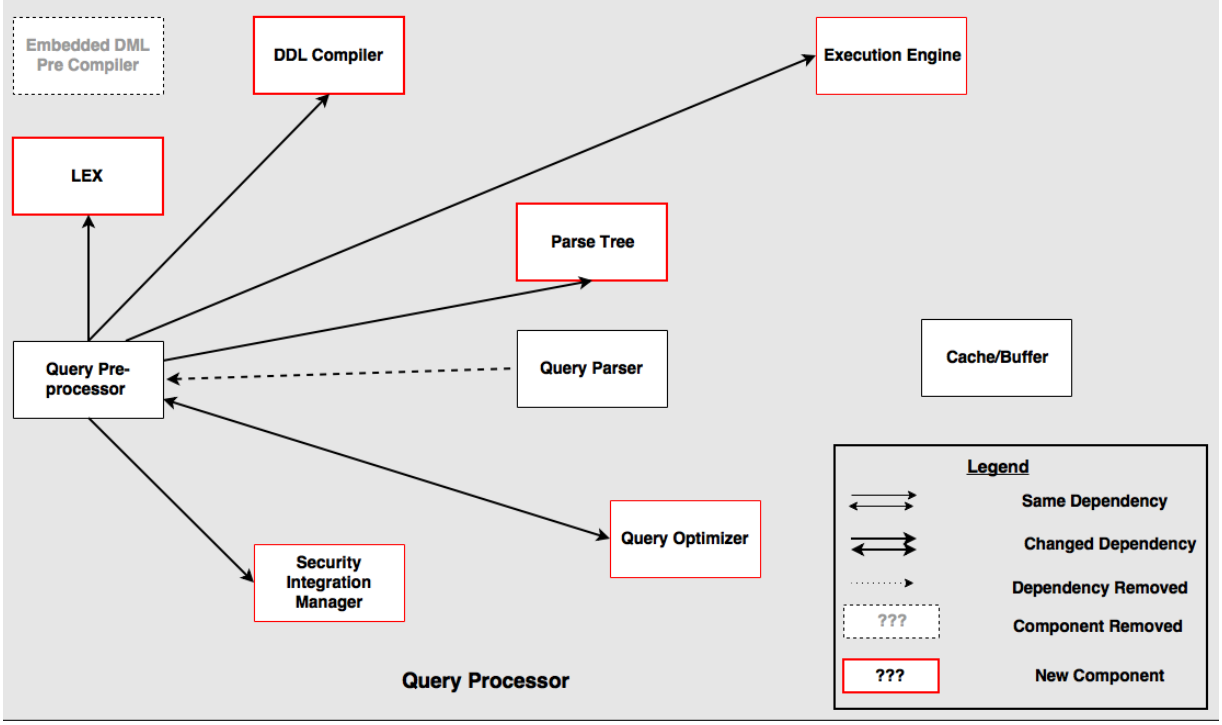


Figure 7. Concrete Architecture of the Query Preprocessor (drawn using draw.io)

In the conceptual architecture, the query preprocessor had only one dependency, the Parser subsystem. Digging deeper into the source code, from Figure 7 above, the query preprocessor subsystem has many dependencies with other subsystems. Firstly, the query preprocessor has a dependency on the Parse Tree subsystem. It uses the tokens of the query found in the parse tree subsystem to perform the semantic validation. To read these tokens, the query preprocessor makes use of the Lex subsystem. Next, the query preprocessor has a dependency on the DDL Compiler to verify whether the query is syntactically correct. As stated earlier, the parse tree is only valid if it passes the syntax and semantic checks. Therefore, the query preprocessor relies on the DDL Compiler for syntax checks. Thirdly, the query preprocessor has a dependency on the Security subsystem, to verify if the client has the privileges on the tables and attributes mentioned in the query. This comes under the semantic validation, to ensure that the query is meaningful. Finally, the query has a dependency on the Execution subsystem, especially when resolving table and column names specified in the query. The preprocessor ensures that the tables and column actually exist within the database schema.

A parse tree is valid if and only if it is semantically and syntactically valid. If a parse tree is valid, the system moves onto the query optimizer, otherwise an

error is generated. One dependency that was not in the conceptual architecture was the Query Preprocessor towards the Query Optimizer. A reflexion analysis is provided below.

Query Preprocessor → Query Optimizer

Which?	SELECT_LEX::prepare() (in .../sql/sql_resolver.cc) depends on Prepare_error_tracker (in .../sql/sql_optimizer.h)
Who?	Guilhem Bichot
When?	Commited on Nov 24, 2012, 9:45AM EST
Why?	fix for Bug#13996639 CRASH IN ITEM_REF::FIX_FIELDS ON EXEC OF STORED FUNCTION WITH ONLY_FULL_GROUP_BY. See per-file comments. … sql/sql_resolver.cc: If prepare () fails, tag the Lex as "broken"

Table 3. Reflexion Analysis for the Query Preprocessor (history from Github)

The reasoning behind this change seems to mainly originate from resolving a bug. From the commit comments, the class “Prepare_error_tracker” was added as a workaround to prevent a prepare() method from failing. Judging from the comments on the bug, it seems that MySQL crashed if this scenario happened. This class was added as a subclass to the Query Optimizer subsystem, which is why the dependency exists.

Security Integration Manager

The Security Integration Manager is responsible for checking whether the client has access to the connection they are trying to make, and keeps track of table and record privileges. After running the reflexion analysis method, we discovered a number of differences between the dependencies for the Security Integration Manager and other subsystems.

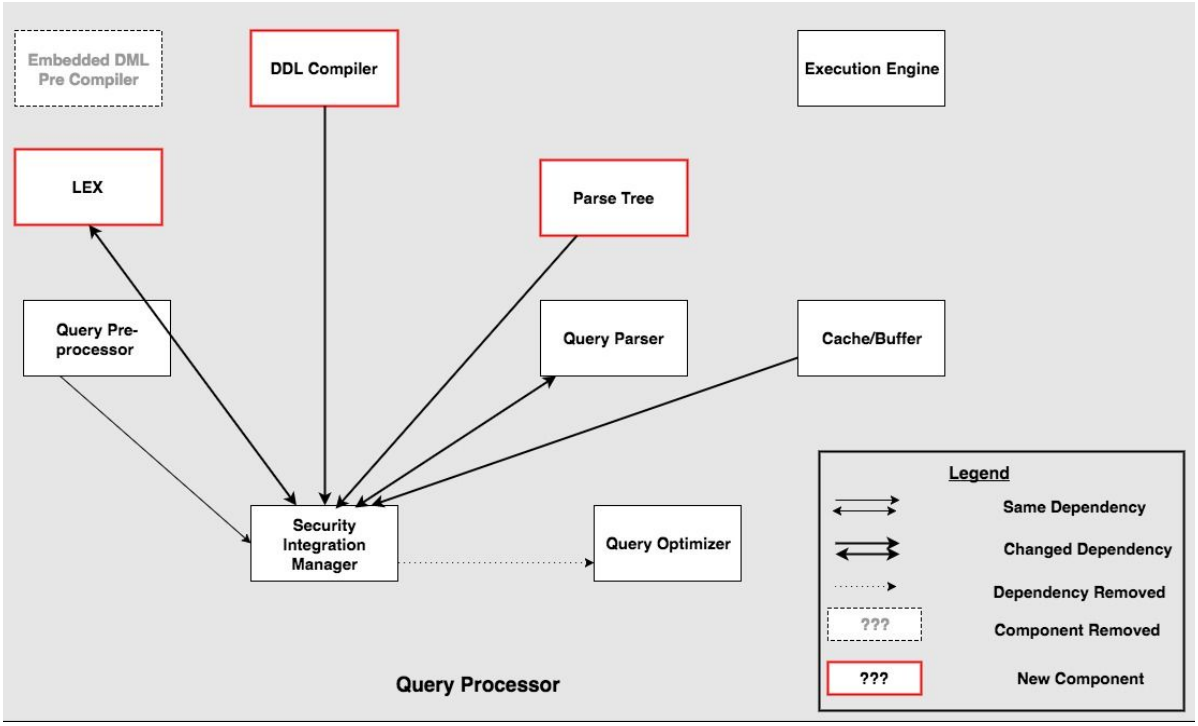


Figure 8. Concrete Architecture of the Security Integration Manager (drawn using draw.io)

Firstly, we found a correct connection in that there is only a one way dependency from the preprocessor to the Security Integration Manager, where the preprocessor used the security manager for checking privileges. Secondly, from our conceptual architecture we said that the security manager uses the optimizer, but from the concrete architecture we found that there is no dependency between the security manager and the optimizer or the execution engine.

We also found unexpected dependencies where the Cache, Parse Tree, Lex, DDL compiler and parser subsystems all use the Security Manager. Digging into the code and reading the source code and comments, we can figure out why the unexpected dependencies occurred. The cache depends on the Security manager as the Query Cache needs to make sure that the user retrieving a query from the cache has access to the table. The rest of the subsystems depending on the security manager check the security context manager to see if the user running the query is a privileged user. Also, the security manager uses the Lex and Parser subsystems. The security manager uses the parser subsystem to retrieve information on who the user is for the thread.
The way the cache subsystem uses the security subsystem is illustrated in the sequence diagram below. When running a query by the user, MySQL first checks if the query is already in the Query Cache and then checks if the user has access to view the resulting table.

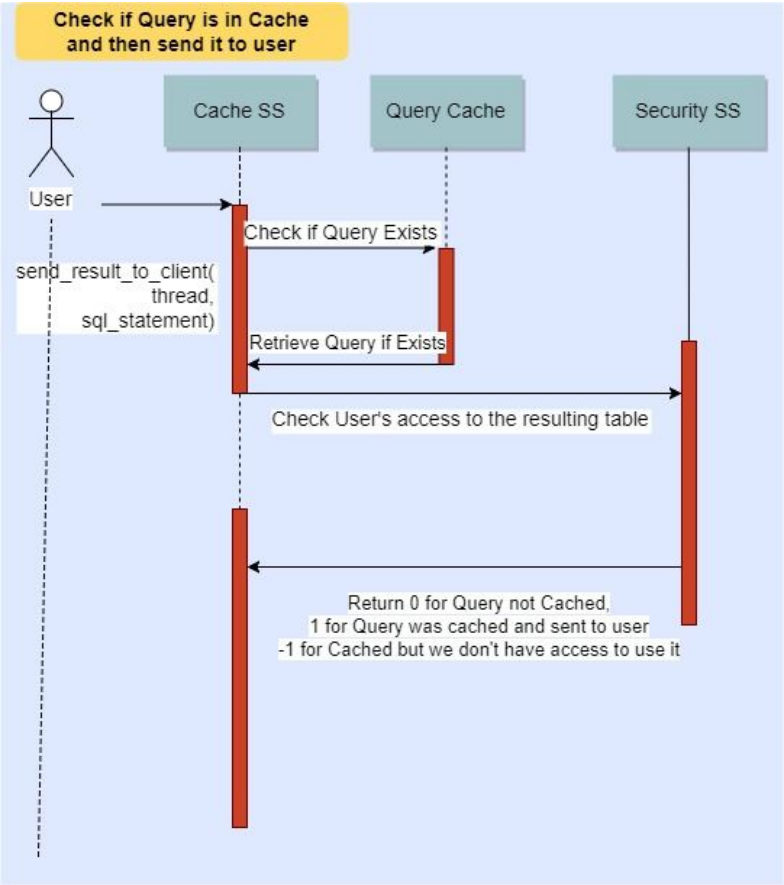


Figure 9. Use Case for the Security Integration Subsystem (drawn using draw.io)

Query Optimizer

Query Optimizer is used to select the most efficient way to compute a query. There are several dependency differences between the conceptual architecture of the Optimizer and its concrete architecture.

First, let's discuss the similarities between the conceptual and concrete architectures of the Optimizer. Both architectures specify a dependency between the Execution Engine and the Optimizer. The dependency is just a consequence of the logical order of query computation: first compile, then parse, then optimize and finally execute. Also, both architectures specify a two-way dependency between the Cache Buffer and the Optimizer. This dependency allows for faster query computation: tables computed for recent queries are cached and then simply retrieved by the Optimizer. This allows the system to bypass filtering by systems between the Parser and the Optimizer. To improve computation efficiency, communication channels were introduced between the Optimizer and those systems.

While speaking of architecture differences between the architectures, it is important to note the absence of communication of Security Integration Manager and the Optimizer. Instead, Optimizer communicates directly with the Pre-processor and is not burdened with security issues. Another interesting difference between the concrete and the conceptual architectures is the seemingly unreasonable dependency between the Optimizer and the DDL compiler. After looking into the code, it was established that the dependency is needed for debugging purposes: since the optimizer changes the way queries are computed, it must be verified that the optimized query and the original query return same data.

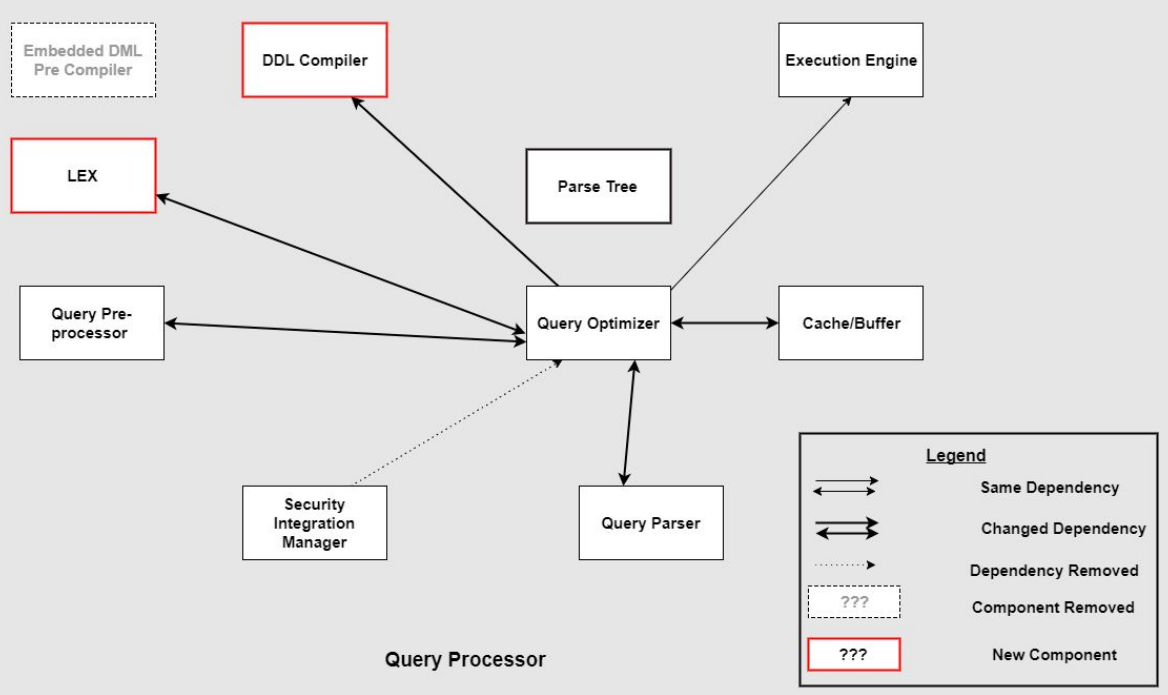


Figure 10. Concrete Architecture of the Query Optimizer (drawn using draw.io)

Execution Engine

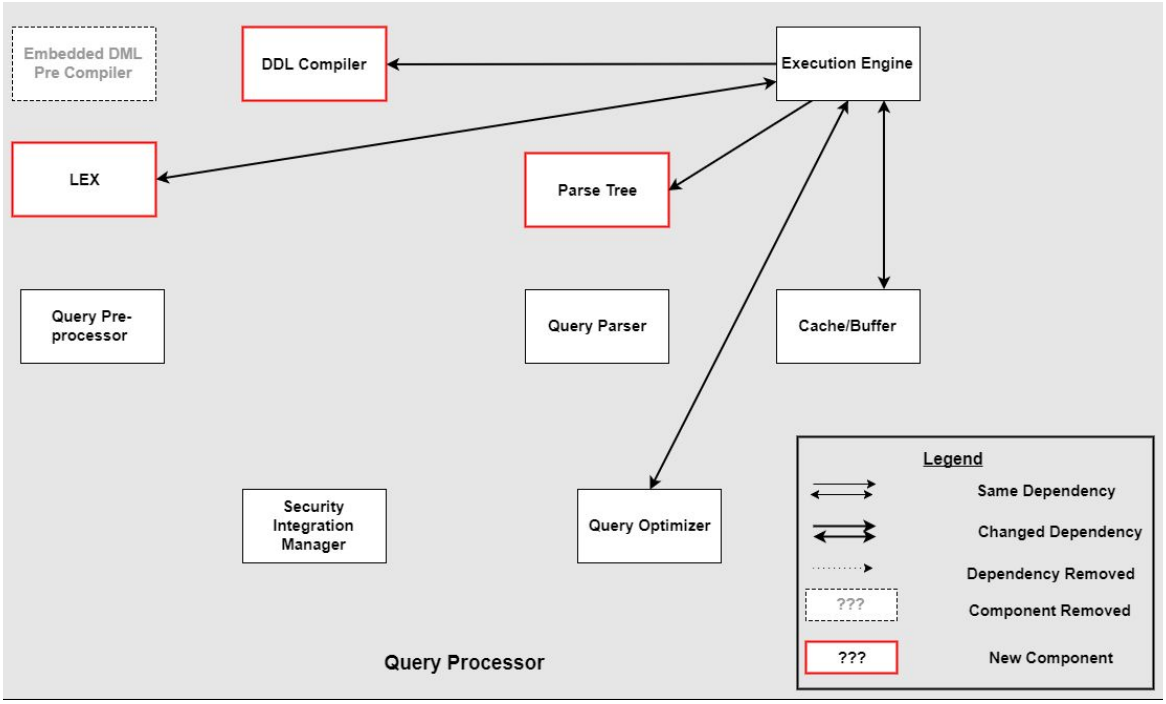


Figure 11. Concrete Architecture of the Execution Engine (drawn using draw.io)

In the conceptual architecture, the execution engine was discovered to be the last step of the query processor. However, after the discovery of the new subsystems in the concrete architecture, it was clear that the pipe-and-filter style wasn't linear. In fact, a number of other subsystems depends on the Execution Engine besides the Optimizer. The Execution Engine depends on the Cache, which saves time if the query was already executed before. Moreover, the Execution Engine subsystem uses the Parse Tree and after tracking the dependency due to the class "Parse_tree_nodes" being used. Usually the Lex subsystem is used with the Parse Tree which explains the dependency between the Execution Engine and Lex.

Data Dictionary

Cache: Stores recently used information so that it can be quickly accessed at a later time.
Compiler: Computer software that transforms computer code written in one programming language (the source language) into another computer language (the target language).

DDL: Data Description Language. DDL is used to define database schemas.
Parser: A program, usually part of a compiler, that receives input in the form of sequential source program instructions, interactive online commands, markup tags, or some other defined interface and breaks them up into parts that can then be managed by other programming. A parser may also check to see that all input has been provided that is necessary.
Query: An inquiry into the database using the SELECT statement. A query is used to extract data from the database in a readable format according to the user's request.
Optimizer: A query optimizer is a critical database management system (DBMS) component that analyzes Structured Query Language (SQL) queries and determines efficient execution mechanisms. A query optimizer generates one or more query plans for each query, each of which may be a mechanism used to run a query. The most efficient query plan is selected and used to run the query.

Naming Conventions

API: Application Programming Interface
CRUD: Create, read, Update, Delete
DCL: Data Control Language
DDL: Data Definition Language
DML: Data manipulation language
FTS: Full text Search
RDBMS: Relational Database Management System
SQL: The Structured Query Language

Conclusions

This investigation has shown that although both the conceptual and concrete architecture of MySQL and its Query Processing Subsystem follow their expected architectural styles, Layered Architecture and Hybrid Compiler Architecture respectively, divergences between the two exist. This is the result of:

1. Additional dependencies between subsystems in the concrete architecture that were not considered in the conceptual architecture
2. Additional subsystems in the concrete architecture that were not considered in the conceptual architecture

While this is to be expected, the number of additional dependencies in the Query Processing Subsystem were far greater than originally anticipated, leading to the notion that this Subsystem has a greater degree of coupling amongst its various components. One of the reasons of this is because after compiling and building

MySQL, new files are introduced into the system, causing more dependencies to each subsystem.

Lessons Learned

Architecture Recovery is a time-consuming process, one that has a significant workload. It was found that using an iterative approach, in which subsystems at the highest level of abstraction were firstly created followed by the breakdown of these subsystems into their constituent components (sub-subsystems), made the task much more manageable. Combining this approach with the use of Automated tools like scripts, grep and Regex, whose use is outlined in the Derivation Process, further simplified the the Architecture Recovery process. Complications, however, arose when subsystems were introduced late in the project lifecycle, since it resulted in additional dependencies that had to be accounted for on short notice. These issues can be easily resolved with improved project management, such as setting a hard deadline for the completion of system and subsystem breakdowns.

References

[1] S. Smith, "MySQL code size over releases", *Ramblings*, 2017. [Online]. Available: https://www.flamingspork.com/blog/2013/03/05/mysql-code-size/. [Accessed: 06- Nov- 2017].

[2] "azkevin/EECS4314", *GitHub*, 2017. [Online]. Available: https://github.com/azkevin/EECS4314/blob/master/A2/a2data.py. [Accessed: 06- Nov- 2017].

[3] Schwartz, Baron, et al. High Performance MySQL: Optimization, Backups, Replication, and more. Beijing, OReilly, 2012