InnoDB Concrete Architecture

Ryan Bannon (mrbannon@uwaterloo.ca)
Alvin Chin (achin@swen.uwaterloo.ca)
Faryaaz Kassam (kassam@swen.uwaterloo.ca)
Andrew Roszko (aroszko@swen.uwaterloo.ca)

CS 798
Professor Ric Holt
2002-02-19

Table of Contents

Abstract

This paper presents a concrete architecture from a proposed conceptual architecture for the InnoDB Database Management System (DBMS). The goal of this was to provide future developers of InnoDB with a specification of the architecture of InnoDB. The paper first goes into a brief introduction of InnoDB and then presents a proposed conceptual architecture based on [15]. Beyond this, InnoDB documentation and literature specific to InnoDB were examined to assist in refining the actual conceptual architecture of InnoDB.

Once the conceptual architecture was refined, a software architecture extraction tool, Swagkit, was used to extract the concrete architecture of InnoDB. The concrete architecture was then compared to the conceptual architecture for InnoDB, and a few noticeable differences were found; in particular, three new subsystems were discovered and a host of unexpected interactions were found to be present in the extraction. Despite these apparent anomalies, it was found that although the actual methods of implementation may not have been predicted accurately, the derived conceptual functionality was indeed found to be correct

1.0 Introduction and Overview

The purpose of this document is to present the findings from the concrete architecture extraction of the InnoDB module of MySQL 4.0. Specifically, this document should inform the reader of what InnoDB is, how the concrete architecture was discovered, and how the concrete architecture was compared to a predefined conceptual architecture of InnoDB.

This document is organized into the following sections. Section 1.0 (this section) is the Introduction and Overview that summarizes the purpose of this document, the organization and the conclusions. Section 2.0 introduces the InnoDB and its role in MySQL. Section 3.0 uses the conceptual architecture of MySQL derived in [15] as a starting point in order to create a conceptual architecture for InnoDB. Section 4.0 goes on to describe the discovered concrete architecture of InnoDB. This part of the document outlines the extraction process, the structure of the concrete architecture, including identification of subsystems within InnoDB, and details regarding many module interactions. Section 5.0 compares the conceptual architecture of InnoDB (Section 3.0) to the concrete architecture (Section 4.0). The discovery of new modules in the concrete analysis of InnoDB is discussed, along with divergences and absences found between the conceptual and concrete architectures. Section 6.0 provides a Glossary of terms used in this document. Finally, Section 7.0 provides a list of references used in research for this project.

Since there is no architecture diagram or documentation provided by the designers, the extraction of InnoDB proved to be a much more tedious process than was originally assumed. It is evident that one can never be entirely sure how to organize modules and subsystems into an architecture. A lot of reworking was done in order to agree on a concrete architecture that was both reasonable and conformed to the program's implementation. While the architecture decided upon may not be exactly as how it was designed, it is a logical architecture based on many hours of work and analysis.

2.0 InnoDB

InnoDB is essentially a stand-alone database management system that is rapidly gaining popularity; it is in fact used to run slashdot.org. In order to elegantly offer vastly superior functionality without compromising performance, MySQL has integrated InnoDB into their system. This addition provides MySQL users with the capability to perform robust transactions and crash recovery, row-level locking, as well as foreign key constraints. The integration was successfully performed by adding the InnoDB table type as an option to the user upon creation. The InnoDB transaction model is among the best on the market for it combines the best properties of a multi-versioning database along with traditional two-phase locking. Locking is performed on a row level and queries are run by default as non-locking consistent reads, in the style of Oracle. The lock table in InnoDB is stored so space-efficiently that lock escalation is not needed: typically several users are allowed to lock every row in the database, or any random subset of the rows, without InnoDB running out of memory.

The conceptual architecture outlined in the first assignment was based on the most recent version of MySQL, which incorporated InnoDB into the backend thus providing the additional functionality outlined above. As a result, the transaction and recovery subsystems are specific to InnoDB; since it is also a stand-alone DBMS, InnoDB also contains some form of storage management and parsing. It therefore seemed like a logical decision to perform the concrete architecture extraction on the InnoDB component.

3.0 Conceptual Architecture

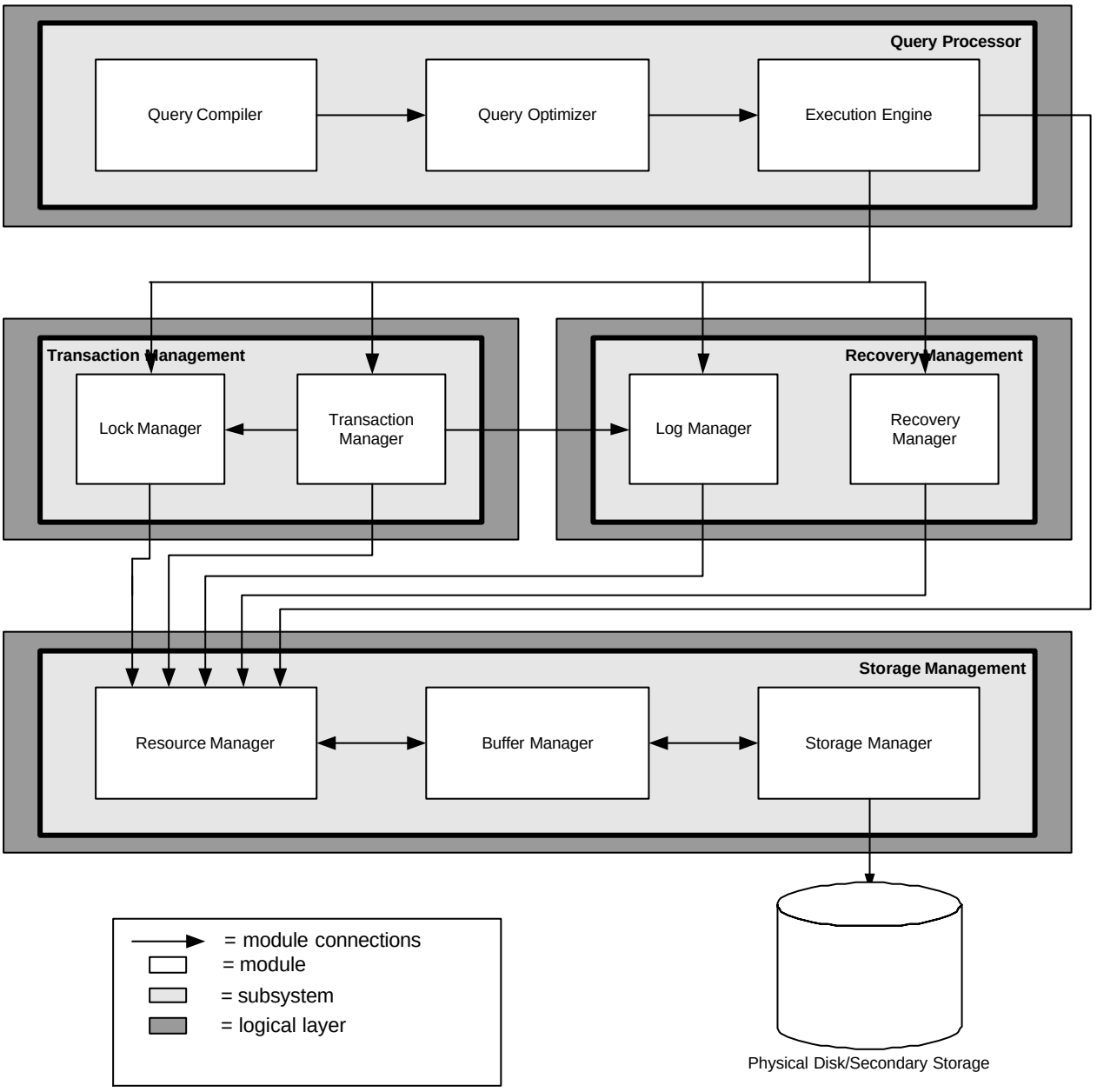

Figure 1: Conceptual Architecture of InnoDB

Using the MySQL conceptual architecture from [15] as a starting point, the InnoDB documentation was analyzed in order to obtain an accurate equivalent representation, seen above in Figure 1. As mentioned previously, since InnoDB is a stand-alone system that provided MySQL with much of its functionality, it was not surprising to find that its conceptual architecture was indeed very similar to the one found in [15]. It is evident that the transaction and recovery subsystems should be identical; however, in addition, it was discovered that the storage management was also very similar. It can be clearly observed in the diagram that, as is the case in

MySQL, the logical layers (marked in dark gray) are connected in a strictly layered architecture [6]. The only major differences found were in the query processor; it was found to only contain a compiler (lexer and parser), an optimizer, as well as the execution engine. It can be noted that the precompiler, preprocessor, DDL compiler, and security manager all found in MySQL were not present unpinned. It can finally be observed that the pipeline architecture [6] of both the query processor and storage management subsystem is indeed maintained.

4.0 Concrete Architecture

The extraction of the concrete architecture shed a tremendous amount of light on the actual implementation of the system. In addition to the discovery of new subsystems, a greater understanding of the specific functionality of the existing modules was obtained. This section provides an overview of the extraction process carried out and then goes on to outline the functionality and interactions between each of modules depicted below in Figure 2. It can be noted that for purposes of simplicity, the utilities subsystem has been omitted from Figure 2.

4.1 Extraction Process

In describing the concrete architecture of a system, one must analyze the directory structure and source code of the system. The **InnoDB** directory (Innobase) of the MySQL 4.0 alpha source code and its subdirectories were traversed and examined. The C files in each subdirectory were then mapped to the modules of subsystems in the InnoDB conceptual architecture in Figure 1. The mapping was accomplished by looking at the name of the C file then trying to infer the particular module that this would belong to. If that was not sufficient, then the functions and comments were inspected further in the C file, to attempt to place it in the proper module. For C files where it was not easily identifiable as to which module it belonged to, they were placed in a sub module within a newly created miscellaneous module outside of the conceptual architecture. The reason for doing this was that by extracting the visualization of the concrete architecture, the relationships between modules and C files would assist in determining the placement of the unknown C files.

After all the C files in the Innobase directory of MySQL were mapped to the appropriate modules of the conceptual architecture, the next step in the process was to extract the concrete architecture by running it through the **Swagkit** pipeline. First, the **cppx** tool extracted the facts from the Innobase C files. Then, the **prepx** tool raised those facts. After that, the **link** tool linked the units given from the **architecture_contain.rsf** file (containment of subsystems and modules from the InnoDB conceptual architecture as well as the mapping of the C files to them) along with clustering and laying out of those facts. Finally, the **lsedit** tool displayed the final concrete architecture.

The visualization of the concrete architecture in lsedit was analyzed and was redrawn in Microsoft Visio to illustrate clearly the call flows between modules; this is shown above in Figure 2. Each call flow was examined in detail and additional paths in the concrete architecture that were not in the conceptual architecture were noted. The C files that were placed in the miscellaneous module were analyzed by investigating their call links to other modules and C files. From here, it was determined as to what particular module that this C file did belong to. The architecture_contain.rsf file was then appropriately modified to reflect one change at a time, and relinked to visualize the change in lsedit. The visualization was further analyzed to determine if additional call links were inserted, and to validate the understanding of the concrete architecture. This process was repeated with each change, until all unresolved C files and sub modules were placed in their appropriate modules from the conceptual architecture.

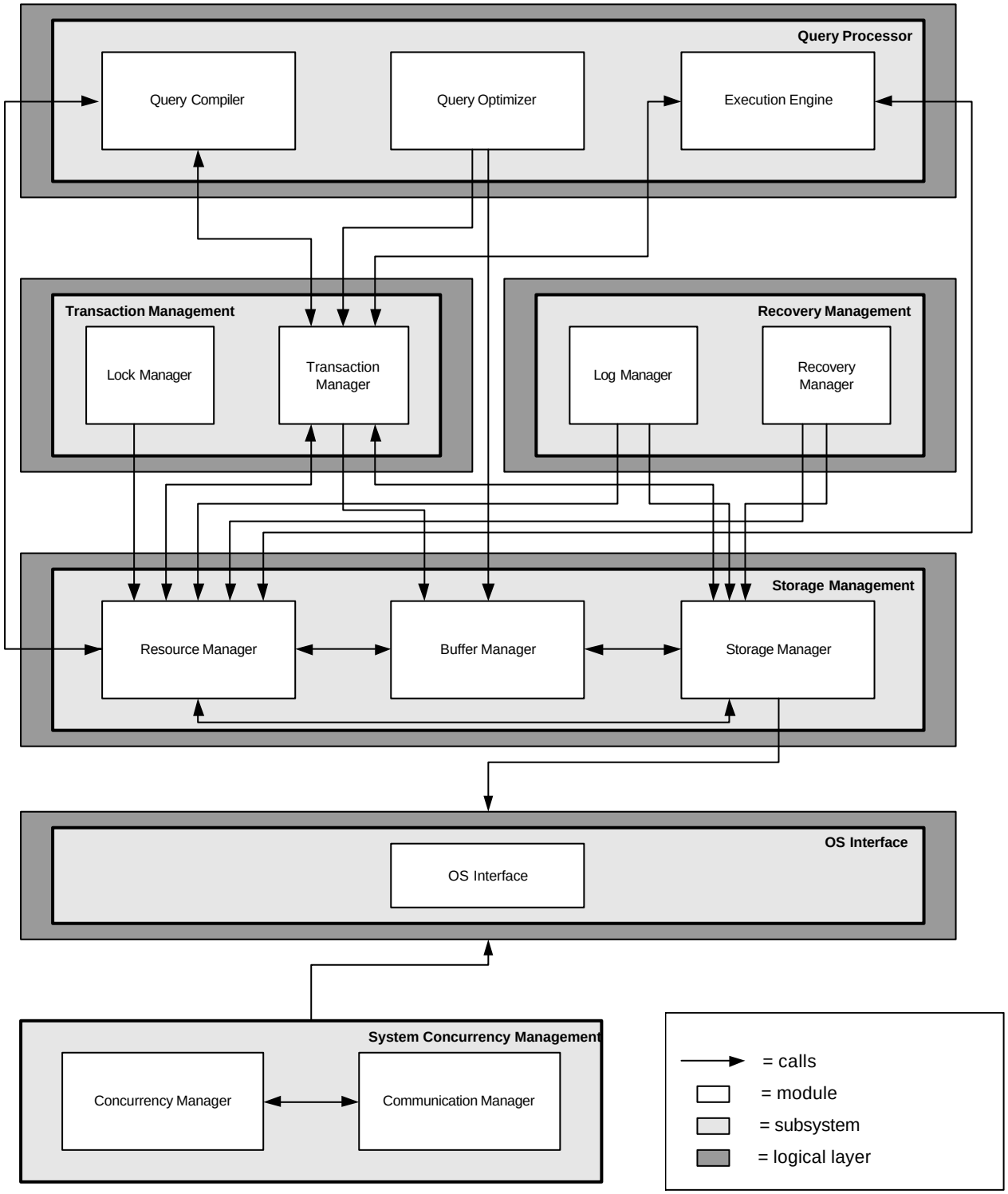

Figure 2: Module Level Concrete Architecture

4.2 Subsystems

4.2.1 Query Processor Subsystem

4.2.1.1 Query Compiler

The Query Compiler performs the parsing (`pars0pars.c` in the *`pars`* directory) and the lexical analysis (`lexyy.c` in the *pars* directory) of the InnoDB query. The parsing symbol table for the Query Compiler is located in `pars0sym.c` in the *pars* directory. The InnoDB query is a SQL query, which consists of SQL statements, that are transactions, which operate on the InnoDB tables. In order to compile the query, the Query Compiler must obtain access to the InnoDB table and record, as well as the transactions in order to build the parse tree. Hence, the Query Compiler obtains the InnoDB table and record (resources) from the Resource Manager. The transaction-based components are obtained from the Transaction Manager.

4.2.1.2 Query Optimizer

The Query Optimizer (`pars0opt.c` in the *pars* directory) optimizes the InnoDB query so that it can run efficiently and quickly. When rollback occurs for database recovery, the Query Optimizer optimizes the transactions (Transaction Manager) from the log. The Query Compiler outputs a parse tree, which is found in memory and in order to optimize it, the Query Optimizer must contact the Buffer Manager in order to successfully retrieve it.

4.2.1.3 Execution Engine

Once the query is optimized for maximum performance, it is ready to be executed. The `eval0eval.c` and `eval0proc.c` in the *eval* directory comprise the Execution Engine. The transaction specific component of the query is performed via calls to the Transaction Manager. It can be noted that the Execution Engine uses the Resource Manager's record comparison utility (rem0cmp.c).

4.2.2 Transaction Management Subsystem

4.2.2.1 Lock Manager

In the InnoDB transaction model, the goal has been to combine the best properties of a multi-versioning database to traditional two-phase locking [12]. InnoDB does locking on a row level, with read and write locks. The Lock Manager located in the file lock0lock.c in the lock directory handles all the locking. Several users on InnoDB rows and tables use a lock table in InnoDB in order to keep track of the various locks. The Lock Manager uses the lock table from the Resource Manager.

4.2.2.2 Transaction Manager

In InnoDB, all user activity happens inside transactions. If the auto-commit mode is used in MySQL, then each SQL statement will form a single transaction [12]. SQL transaction keywords and statements such as `COMMIT` and `ROLLBACK` are parsed and identified by the Query Compiler as part of the InnoDB query. The atomic units in the transaction of the InnoDB query are identified and scheduled by the Transaction Manager. The Execution Engine then executes these. The Transaction Manager reads log records from the Resource Manager in order to perform the undoing of transactions for recovery purposes. It is responsible for starting transactions, committing transactions, assigning rollback segments to each transaction, and functionality to place transactions in various states and queues. The Buffer Manager allocates the queues and transaction objects, whereas the access to log files is obtained from the Storage Manager. The Transaction Manager is located in the *trx* directory.

4.2.3 Recovery Management Subsyste m

4.2.3.1 Log Manager

The Log Manager is responsible for maintaining a log of operations for the InnoDB queries for backup and recovery. The `log0log.c` file in the *log* directory provides the functionality for flushing the database pages to a log file as well as writing the transaction operations to a database log (handled by the Storage Manager) in a safe, concurrent manner. In doing so, it has to access specific library functions that operate on the log located in disk storage. When performing a backup or recovery, InnoDB scans the log files (log records accessed by the Resource Manager) forward from the place of the **checkpoint** applying the logged modifications to the database [12].

4.2.3.2 Recovery Manager

To recover from a crash of an InnoDB database, the MySQL process has to be restarted. InnoDB will automatically check the logs (via the Storage Manager) and perform a roll-forward of the database to the present. The Recovery Manager consists of one file `log0recv.c` in the *log* directory. InnoDB will automatically roll back uncommitted transactions that were present at the time of the crash. In performing the recovery, the Recovery Manager accesses functions, data and the log records (Resource Manager).

4.2.4 System Concurrency Subsystem

Apart from successfully running concurrent transactions, there is also a great deal of low-level concurrency in the database server. In order to efficiently handle simultaneous client requests and distribute work in the most efficient manner, there are a number of thread pools present in the system. Firstly, a dedicated set of user threads, running at normal priority, waits to receive client requests. Each of these requests is then taken by a single user thread, which starts processing and, when the result is ready, sends it to the client. Secondly, there is another group of user threads, known as the parallel communication threads, which is intended for splitting the queries and processing them in parallel. These threads are suspended on event semaphores while waiting for tasks, which can be run in parallel. Furthermore, a single user thread waits for input from the console (e.g. a command to shut the database). Finally, there are a group of utility threads, which generally run at a lower priority. They are responsible for a variety of background tasks, including the timely flushing of the buffers.

It can be noted that a master thread controls the server. This thread runs at a priority higher than all other threads in the system. It sleeps most of the time and wakes up every 300 milliseconds to check whether there is anything happening in the server that requires intervention. Such situations may occur when flushing of dirty blocks is needed in the buffer pool or an old version of database rows has to be cleaned away.

It is very important to note that this subsystem does not truly interact with the components that contain the actual server functionality. It handles the low level threading and communication, which is used to carry out all the required commands in the other subsystems; it can be viewed as an orthogonal component. It can be observed, however, that it does interact with the OS interface in order to carry out its tasks. The system concurrency subsystem is further broken into two modules and contains the underlying functionality for the synchronization and thread communication in the system. It essentially provides resources crucial to the execution of server tasks.

4.2.4.1 Concurrency Manager

The Concurrency Manager (files located in the *sync* and *thr* directories) contains the implementation of the spin-lock mutex and read-write locks required for thread synchronization. In addition, there is the notion of a thread local storage, which is associated with the thread id and contains private information. There is also an additional implementation of a fast mutex for inter-process synchronization, which utilizes the shared memory implementation in the communication module.

4.2.4.2 Communication Manager

The Communication Manager (files located in the *com* directory) implements the primitives for thread communication. The file com0com.c contains an interface for the communication while the second file, com0shm.c, contains a shared memory implementation for the interface. It can be noted that in order to perform these tasks, the Communication Manager must make calls to the operating system shared memory primitives.

4.2.5 Data/Utilities Subsystem

Every large piece of software contains a number of common utilities and data structures that are continually used throughout the system. This subsystem, which contains two modules, is a representation of these commonly referenced implementations.

4.2.5.1 Data Structures

There are a number of data structures that have been used throughout the application; these include a hashtable with external chains, which in turn uses a simple hashtable utility (found in the *ha* directory), as well as a dynamically allocated array (found in the *dyn* directory). The hashtable is used, for example, in the creation of an efficient index when the database is small enough to be fully loaded into memory whereas the array is used in the concurrency manager for the synchronization of threads.

4.2.5.2 Utilities

There are also a number of common utilities included in the *ut* directory. These include string, memory, as well as hashing and random number utilities in addition to a number of miscellaneous functions such as time stamping, sorting, printing the contents of memory buffers etc.

4.2.6 Storage Management Subsystem

Data is stored on disk, however, in order to be manipulated, it must be first loaded into main memory. This process is handled by the Storage Management Subsystem; the Storage Manager interacts with the file system to retrieve the data, the Buffer Manager efficiently maintains the memory space for data storage, and in addition to initiating the requests, the Resource Manager undertakes the actual manipulation of the data.

4.2.6.1 Storage Manager

The Storage Manager is responsible for providing fast read/write access to tablespaces and logs of the database. A tablespace consists of database pages whose default size is 16kb; these pages are then grouped into files, or segments, of 64 consecutive pages. To acquire more speed in disk transfers, a technique called disk striping is employed. This means that logical block addresses are divided in a round-robin fashion across several disks. The fil0fil.c file in the *fil* directory contains the implementation of the low-level file system, including both the structures and the methods to manipulate them. Furthermore, the fsp0fsp.c file (found in the *fsp* directory) takes care of the file space management, which, for example, handles the allocation of pages to files and keeps track of which files are used, open, and closed. It can be observed that the storage manager receives commands from the buffer manager and interacts with the operating system to return the appropriate database pages.

4.2.6.2 Buffer Manager

The Buffer Manager is responsible for efficiently storing the data in memory for manipulation; it accepts formatted table requests from the Resource Manager and decides precisely how much memory to allocate. The implementation of this mechanism can be found in the InnoDB *buf* directory. The buf0buf.c file is the buffer pool where the file pages are loaded into memory blocks known as buffer frames. In addition, buf0flu.c contains the functionality to flush the buffer and buf0lru.c contains the replacement algorithm, which decides which blocks should be shifted back to disk. Finally there is a file called buf0rea.c, which calls the storage manager to initiate a file read. It can be noted that the Buffer Manager also contains the mem0mem.c and mem0pool.c files, which provide functionality for the lower level Data/Utilities Subsystem. This handles the mapping of the buffer pool to an actual memory pool as well as the implementation of the memory heap.

4.2.6.3 Resource Manager

The responsibility of the Resource Manager is to accept requests from the higher levels, whether it is a lock, transaction, or log request, and translate them into an appropriate "table" format that can be understood by the Buffer Manager. Once the data has been loaded into memory, the Resource Manager contains the functionality that actually manipulates the data. The functionality in the *row* directory is included in the Resource Manager, which as the name implies, manipulates rows of the database; operations include the purging of obsolete records, the updating of indices upon row update, the construction of rows based on searches in the index, undo row operations, the selection and update or rows as well as alphabetical comparison of rows. Furthermore, all of the files in the *dict* directory have been clustered into the Resource Manager. These files implement all the data dictionary (metadata) functionality that is associated with the actual data. It should be noted that there are a number of interactions where the Resource Manager skips the Buffer Manager and accesses the storage manager directly. At first, this notion seemed counter-intuitive for it would seem that all the commands required data to be loaded into the buffer pool. However, upon further inspection, it was found that not only could a number of commands be performed by accessing the storage manager directly, but the effect of the pipeline was indeed maintained for a direct call to the storage manager would in turn cause a call to the Buffer Manager ordering it to load the relevant data.

4.2.7 Operating System Interface

InnoDB has provided, in the *OS* directory, an interface to the operating system that allows access to the file I/O access, shared memory, synchronization, threading, and process control primitives. The concurrency and communication modules as well as the storage manager for file access access this module predominantly. It can be noted, as seen in Figure 2 above, that this subsystem can be viewed as the fourth and bottommost logical layer in the layered architecture.

5.0 Architectural Comparison

The ensuing section will highlight the main differences between the conceptual and concrete architectures. There were indeed a number of surprising interactions, including a vast number of extracted connections that were not accounted for in the conceptual representation as well as a number of expected connections that were not found in the concrete architecture. The comparison of the two architectures can be seen below in Figure 3; the red lines represent the *divergences* (links found in the concrete and not in the conceptual), the green lines represent *absences* (connections in the conceptual not found in the concrete), and the black lines represent *convergences* (links found in both representations). It can be noted that these three terms were introduced by Gail Murphy et al [8].

5.1 Discovered Modules

As seen below in Figure 3, the completed extraction indicated the existence of three subsystems that had not been accounted for in the conceptual architecture: the System Concurrency Subsystem, the Data/Utilities Subsystem, as well as the Operating System Interface Subsystem. During the formation of the conceptual architecture, we focused mainly on the actual functionality of the DBMS and it was found that we were indeed quite accurate in this respect. However, we did not pay close attention to the required implementation for such behavior; this oversight is evidenced by the overlooked subsystems. The three of them merely provide services to the modules, which actually carry out the documented system behavior. It should be noted, as demonstrated in the Figure 3 above, that the Operating System Interface is seen as a fourth logical layer in the system; it is used by the storage manager to access the secondary storage. This classification can be contrasted to the Data/Utilities Subsystem, which is used throughout the server, as well as the System Concurrency Control Manager, which, as explained above, can be viewed as an orthogonal component.

5.2 Divergences

The extraction produced a number of interesting interactions, depicted in red in Figure 3, that were not accounted for in the conceptual architecture. In most cases, these newfound links provided valuable feedback regarding the server functionality. The reasoning for the presence of all the links found in the extracted architecture has been outlined in Section 4; certain interesting differences should, however, still be highlighted. It should be noted that in virtually all cases, the divergences were not as a result of a misunderstanding of the system functionality; the behavior was merely carried out in a slightly different manner than expected.

5.2.1 Layered Architecture Violation

The conceptual architecture outlined in Section 4 is a strict layered architecture; however, it can be observed from the concrete architecture in Figure 3 below that there were a number of interactions that violated this model. In particular:

- Query Compiler -> Resource Manager
- Query Optimizer -> Buffer Manager
- Execution Engine -> Resource Manager

In all three of these cases, the interaction skips a logical layer thus violating the clean, layered mental model. However, as explained previously, it is indeed logical for the activities carried out by the Query Processor to require Storage Management Subsystem as an intermediary. Despite these exceptions, the conceptual model serves its purpose in portraying the high level functionality of the system in a simplistic fashion.

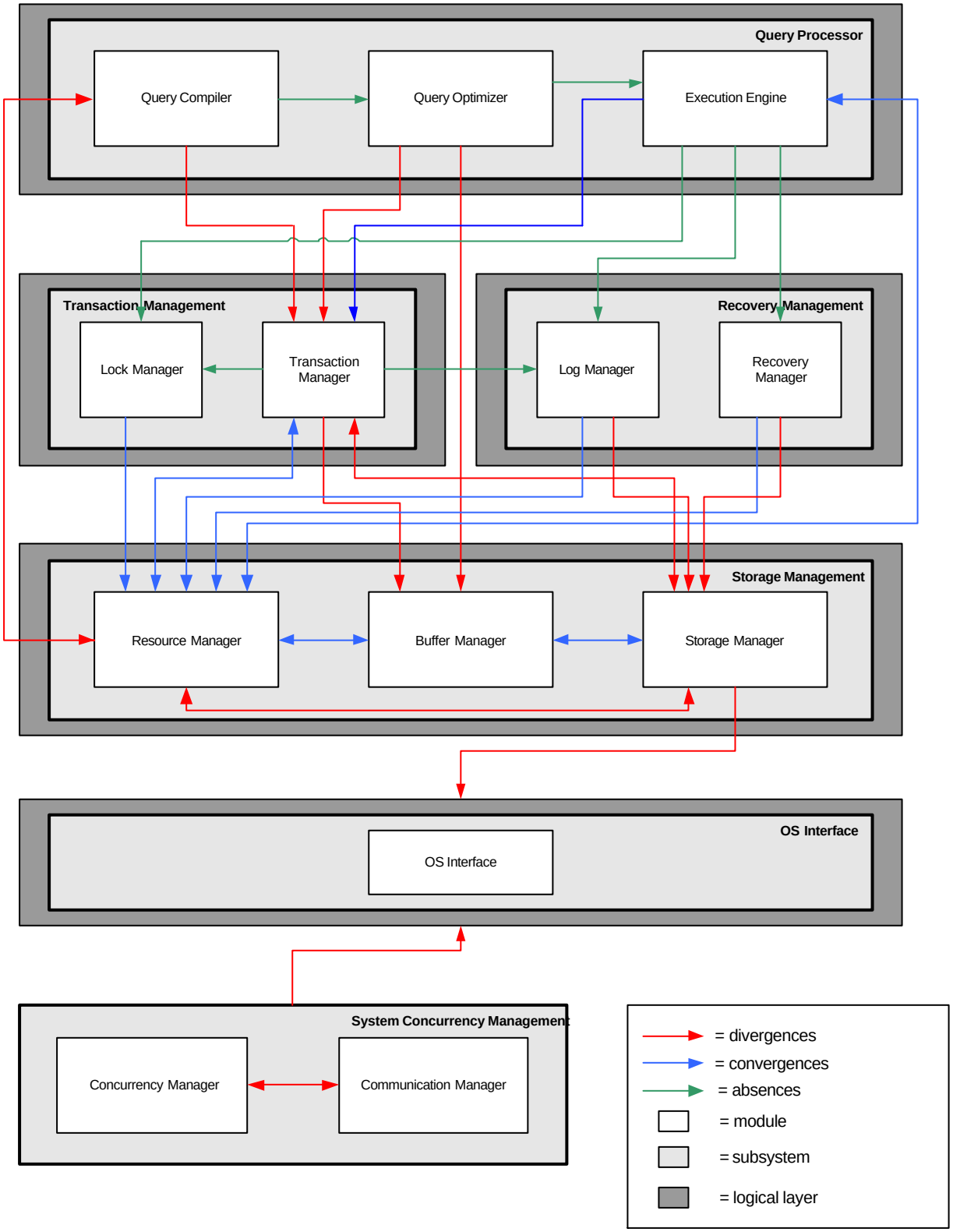

Figure 3: Comparison Between Conceptual and Concrete Architectures

5.2.2 Storage Management Pipeline

One of the distinctive, elegant features of the conceptual architecture is the pipeline in the Storage Management Subsystem. It was assumed that all requests coming into the Storage Management Subsystem would be directed at the Resource Manager, at which point they would get handed off to the Buffer Manager, which would in turn use the Storage Manager to retrieve the appropriate data from secondary storage. The concrete extraction demonstrated that this was indeed not the case; not only did the intra-module interactions violate the pipeline, but a number of the higher level modules accessed the buffer and storage modules directly. The particular divergences of interest are:

- Transaction Manager -> Buffer Manager
- Transaction Manager -> Storage Manager
- Log Manager -> Storage Manager
- Recovery Manager -> Storage Manager
- Resource Manager -> Storage Manager

The specific explanations for each of these surprising connections can be found in Section 5 of the report. The main conclusion drawn is that all requests from the upper layers do not need to be directed to the Resource Manager. There are a number of operations, such as log manipulation, which could be performed by accessing the underlying Storage Manager directly. It is very important to note in this case that the fore-mentioned functionality implied by the pipeline is, for the most part, maintained. In the vast majority of cases, a direct call to the Storage Manager would cause it to load data into memory by calling the Buffer Manager. As a result, it is evident that the system interactions were indeed understood at the conceptual level though the actual control flow was slightly different than anticipated.

5.3 Absences

The green links in Figure 3 represent the links in the conceptual architecture that were not reflected in the extraction. These interconnections were quite a mystery at first; however, upon careful analysis, it was found that each and every one of these missing links is implemented indirectly through an intermediate module. For example the Query Compiler->Query Optimizer->Execution Engine pipeline is indeed still implemented; however it is through the Transaction Manager and Buffer Manager. In addition, detailed analysis also proved that the Transaction Manager-> Log Manager link still exists; however, it is merely done through the storage management subsystem. In order to read and write to the log, it must be clearly be implemented via the underlying memory and file resources; however, conceptually it is understandable to view it as a link directly between the modules. In the end, all the functionality implied in the conceptual is indeed present; as in the base with the divergences, the means of implementation was in a slightly different manner than predicted.

6.0 Glossary

Absence: Term used for interactions present in the conceptual architecture not found in the extraction

API (Application Programming Interface): A set of functions in a particular programming language issued by a client that interfaces to a software system.

Atomically : The "all or nothing" execution of transactions.

Convergence: Term used for coinciding interactions found in both the concrete and conceptual architectures

Disk Controller: A circuit or chip that translates commands into a form that can control a hard disk drive.

Divergence: Term used for interactions present in the concrete architecture not present in the conceptual architecture

DML (Data Manipulation Language): this is a generic language in which the actual SQL statements are embedded in the client or translated from the client code.

DDL (Data Definition Language): This is the language that the RDBMS understands. In MySQL, and all SQL databases, the DDL is SQL itself.

Deadlock A failure or inability to proceed due to two transactions having some data that the other needs.

Index: A means to speed up access to the contents in database tables. The MySQL query optimizer takes advantages of indexes in order to speed up the processing of queries.

Main Memory: The storage device used by a computer to hold the currently executing program and its working data for fast access.

Meta-data: Data whose purpose is to represent the structure and meaning of the actual data.

Query Optimizer: Component in the Query Processor whose primary purpose is to optimize the queries so that they can be processed faster.

Query Precompiler: Processes the client code from the application layer to extract the SQL statements or translate the code into SQL statements.

Query Preprocessor: Performs the checking of syntax and semantics of the query before the query is processed, optimized and executed.

RDBMS (Relational Database Management System): A system that provides the management, storage and handling of requests to a relational database.

Transaction: One or more commands grouped together into a single unit of work.

Virtual Memory: Memory, often as simulated on a hard disk, that emulates RAM, allowing an application to operate as though the computer has more memory than it actually does.

7.0 References

1. Atkinson, Leon. Core MySQL: The Serious Developer's Guide. New Jersey: Prentice Hall Publishing, 2002.
2. Date, C.J. An Introduction to Database Systems. Menlo Park: Addison-Wesley Publishing Company, Inc, 1986.
3. Dubois, Paul. MySQL. New York: New Riders Publishing, 2000.
4. Frost, R.A. Database Management Systems. New York: Granada Technical Books, 1984.
5. Garcia-Molina, Hector. Database System Implementation. New Jersey: Prentice Hall, 2000.
6. Garlan, David. Shaw, Mary. "An Introduction to Software Architecture". Pittsburgh, PA USA: School of Computer Science, Carneigie Mellon University, 1994.
7. Kruchten, Phillippe. "Architectural Blueprints – The 4+1 View Model of Software Architecture". Rational Software Corp, 1995.
8. Murphy, Gail et al. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models". Vancouver, British Columbia: Computer Science Dept, University of British Columbia, 1995.
9. Silberschatz, Abraham et al. Database System Concepts. New York: McGraw-Hill, c1997.
10. U.S. Department of Commerce. "An Architecture for Database Management Standards". Washington: U.S. Government Printing Office, 1982.
11. http://www.mysql.com/. MySQL AB, 2002.
12. http://www.innodb.com/ibman.html. 2002
13. Yarger, Randy Jay MySQL & mSQL. Sebastopol: O'Reilly & Associates, 1999.
14. www.swag.uwaterloo.ca/~swagkit. February 5th, 2002
15. Bannon, Chin, Kassam, and Roszko. "MySQL Conceptual Architecture". Waterloo, Ontario: Software Architecture Group, University of Waterloo, 2002