CALIFORNIA STATE UNIVERSITY, LONG BEACH

CECS 327

NET-CENTRIC COMPUTING

The Art of MySQL Concurrency

Author:
Tan TRAN

Instructor:
Dr. Burkhard ENGLERT

May 14, 2014

Contents

1 Foreword

The forces underlying a "multiprocessor revolution" in programming are also causing a revolution in the databases field, toward massive multi-client concurrency and the storage of *very large* amounts of data. These forces are both push and pull: database designers are being *pushed* to develop such systems due to the rise of high-throughput web applications, as exemplified by Facebook, which as of 2011 serviced 60 million queries and 4 million row changes per second, and *pulled* to do so in response to recent hardware advances, such as cheap multi-core CPUs and the cheap construction of server farms, that have made it physically possible for databases to service so many transactions at once.

As with multiprocessor programming, databases can manage high throughput in two ways: *distribution* and *concurrency*. Distribution divides data and processing among multiple physical servers, while concurrency manages the simultaneous serving of database operations on single computing units, i.e., one server. This paper introduces the reader to the latter by documenting concurrency principles and practice in MySQL, the most popular open source relational database. The problems addressed will turn out to be just as fascinating as those in multiprocessor programming.

2 Introducing Concurrency Anomalies

In 1969, Edgar Codd formulated the *relational model* of databases, which formalized and argued for a new type of database governed by logical relations, but went without describing its implementation details, except for a mention that "inputs causing [...] inconsistencies [...] can be tracked down if the system maintains a journal of all state-changing transactions" [Codd 387]. When developers in industry implemented Codd's ideas, they found that the implementation of *transactions* would come to be more important than had been anticipated: one traded their usefulness as a description of an atomic unit of work for the new challenge of ensuring database consistency between transactions.

For example, it is difficult to run a banking database without treating money transactions as atomic database transactions. Consider Alice, who has $500, sending $100 to Bob, who has $400. The database starts a transaction, subtracts $100 from Alice's account, and is about to add \$100 to Bob's account when the power to the server fails. Upon restart, Alice has \$500 and Bob has \$500, so that the bank is now in an incorrect state.

Situations like this, and some more complicated, led Jim Gray to define certain correctness conditions for database transactions in 1979, and for Andreas Reuter and Theo Härder to describe them using the acronym *ACID* in 1981. To elaborate, transactions are **a**tomic if transactions either complete successfully or not at all; if they fail partway through, the database should *roll back* the entire transaction. The database ensures **c**onsistency if transactions always bring the database from one correct state to another. Transactions are **i**solated if transaction B does not see the changes made by transaction A until A has finished. Lastly, the database is **d**urable if, once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

The ACID properties are simple to implement for sequential databases. An undergraduate-level implementation, for example, might keep a history of each row's data, assign a timestamp to each transaction, and maintain a global record of the last successful transaction, which transactions would set only upon completing all of their transactions successfully and passing validation. To retrieve data, the database would fetch from history only the rows matching the global timestamp.

But concurrency makes ensuring ACID more difficult, and adds an additional *serializability* requirement [Rob Ch. 10]: that even concurrent transactions maintain ACID; that is, despite transactions not being *physically* isolated, their operations are interspersed as if they *had* been isolated in time. ACIDS (ACID and serializability) is thus useful for evaluating different database concurrency controls.

The difficulty of maintaining ACIDS is illustrated in the following problems.

2.1 Lost Updates

Consider database clients Alice and Bob, who share a bank account tuple and start concurrent transactions to add \$50 and \$25, respectively. Alice reads `balance = 100`, and Bob reads `balance = 100`. Alice sets `balance = 100 + 50 = 150` and Bob sets `balance = 100 + 25 = 125`. Both end their transactions. `balance` should be `100 + 50 + 25 = 175`, but instead is 125 because Alice's update was lost by the subsequent overwrite.

2.2 Dirty Reads

Carol starts a transaction to transfer \$100 from her bank account to Dave's bank account—that is, subtract \$100 from hers and add \$100 to Dave's. Meanwhile, after the subtraction, but before the addition, the bank manager queries the database for the total amount money held by the bank, and sees \$100 less than he should, as Dave has not been given his money yet. The manager has made a dirty read.

2.3 Non-repeatable Reads

The bank does two things monthly: 1) credit customers with 5% interest, and 2) apply a monthly service fee of 1%. Consider Edward, who has \$100 in his account. The database starts a transaction, retrieves all customers, and first credits them with interest, so that Edward now has \$105. The transaction is about to apply the service fee, when Edward suddenly makes a deposit of \$95 and so now has \$200. Then, all customers are again retrieved, the monthly service fee is applied, and Edward is left with \$200 - (0.01×200) = \$198. But that is incorrect, as the +5%, -1% process should be applied to either Edward's original balance of \$100, or his new balance of \$100 + \$95 = \$195, but not a mix of both, as transpired when the first read of Edward was *not repeatable* for the second.

2.4 Phantom Reads

The bank applies a monthly service fee only to customers who have not made a deposit in the last 30 days. A transaction to enforce that policy: 1) retrieves those customers, 2) applies the fees, and 3) retrieves those customers again in order to notify them of the charge. Meanwhile, after (2) but before (3), Frank, who has not made a deposit recently, makes a deposit. He just barely escaped being charged in (2), but nevertheless will see the message in (3). His row did not appear in the first range retrieval, but, like a phantom, suddenly appeared in the second.

3 Improving on Amdahl's Law

A first solution to the preceding concurrency problems would involve locking the database before each transaction, but that would defeat the purpose of concurrency. Further refinement would lock only the relevant tables, and further would lock only the relevant *rows*. Lastly, the database would apply different levels of read/write locks; for example, two transactions which read, but do not write, a row should both be able to access it.

Further and further optimization approaches a theoretical speedup limit, as measured simplistically with Amdahl's Law, described in *AoMP*, section 1.5. However, database designers must also pay attention to two parameters not addressed by Amdahl's Law: queuing delay and coherency delay.

Queuing delay is the time that a scheduled task must wait in the task queue before it can execute. In databases, it models the time that transactions must wait on resources such as, obviously, locks, but also buffers, access to the SQL parser, backup and write-ahead logs, and so on.

Coherency delay is the work that the system must perform to synchronize shared data. An example in *AoMP* occurs in its discussion of test-and-test-and-set (TTAS) locks, which minimize coherency delay by having threads spin on cached copies of a lock variable before moving to change it globally. Although databases do not use TTAS locks, they do make use of mutex locking for up to hundreds of database threads, so that coherency delay is significant.

Much literature exists to analyze the scaling of database performance. But out of all techniques, the one most relevant to *AoMP* is the *Universal Scalability Law* because it is Amdahl's Law with an additional *coherency penalty*[1]:

$$C(N) = \frac{N}{1 + \sigma(N-1) + \kappa N(N-1)}$$

where:

- C(N) is the database's capacity to service requests, as measured by, say, requests per second,
- N is the number of processors available to process requests,
- σ is *contention* (the system's performance degradation due to portions of the work being serial instead of parallel), and
- κ is officially *coherency delay*, but unofficially any type of concurrency delay that scales as the *square* of requests. To perform a regression on empirical

[1]If we let $\kappa = 0$, the Law reduces to $C(N) = \frac{N}{1+\sigma(N-1)}$, Amdahl's Law.

performance data[2] , we include queuing delay in coherency delay.

The Universal Scalability Law improves on Amdahl's Law by modeling the fact that, past a certain point, adding more processors actually *decreases* performance, as running threads must take extra time to enforce the coherence of shared variables such as lock variables. This consideration allows it to model database concurrency extremely well; for example, it fits MySQL concurrency data with $R^2 = 0.99$ in [Schwartz].

The moral of Amdahl's Law is that the best way to increase performance is to increase parallelization rather than the number of processors, as the latter process is subject to *diminishing returns*. In the USL and its application to databases, these returns diminish even faster—quadratically—and thus turn negative past a certain point. Parallelization, then, is even more important than Amdahl's Law models. The following material outlines how parallelization is fine-tuned in MySQL.

4 Introducing InnoDB

MySQL is unique among the major databases for featuring multiple storage engines. Prior to the release of MySQL 5.5 in 2010, MySQL's default storage engine was *MyISAM*, which did not support transactions, locked on the table level, and was notorious for not being crash-safe. It competed with the *InnoDB* engine, which Oracle (a then-competitor) had bought and developed as an alternative which supported transactions, locked on the row level, and was crash-safe. Oracle later acquired MySQL's parent company Sun Microsystems, and proceeded to make InnoDB the default MySQL storage engine as a matter of business and of InnoDB's technical superiority. InnoDB is now recommended for most MySQL applications, apart from edge cases like sequentially accessed, non-critical databases. Notably, InnoDB is so important now that Google and Facebook use it and have contributed to its development.

[2]A more accurate, probabilistic model would treat incoming database operations as Poisson-distributed processes, á la the M/M/c queuing model.

5 Important Concepts

InnoDB, like all major databases now, employs two types of row locks. A *shared* (S) lock permits a transaction to read a row, while an *exclusive* (X) lock permits a transaction to update or delete a row. The names explain the purposes: multiple transactions can share a shared lock, but only one transaction can hold an exclusive lock.

Recall that InnoDB employs both row and table locks. Logically, if a thread T1 holds an exclusive lock on a row R in table X, then another thread T2 should not be able to acquire an exclusive lock on the entire table X, lest one of T2's operations include changing row R. To implement this form of mutual exclusion, we require that all row locks be accompanied by an *intention lock* on the containing table. To detail, if a transaction will only read the rows of a table, i.e., it will only set S locks, then it must first acquire an *intention shared* (*IS*) *lock* on that table. Otherwise, it intends to write/modify at least one row, and thus must acquire an *intention exclusive* (*IX*) *lock*.

This does not mean that IX locks serialize access to the entire table; indeed, two transactions can simultaneously hold IX locks on the same table. Rather, intention locks perform two functions: first, they allow MySQL to detect deadlocks when two transactions intend to modify rows of the same table, but in cyclic order (thereby attempting to set S locks against the other's IX lock), and second, they block full table requests such as `LOCK TABLES...WRITE`, in which case the programmer intends to serialize access to the table anyway.

A common practice is to refer to the *lock type compatibility matrix*, which is grounded in common sense [MySQL Manual 14.2.3]:

	X	**IX**	**S**	**IS**
X	Conflicting	Conflicting	Conflicting	Conflicting
IX	Conflicting	Compatible	Conflicting	Compatible
S	Conflicting	Conflicting	Compatible	Compatible
IS	Conflicting	Compatible	Compatible	Compatible

Lastly, row (X) locks exist in three flavors[3]. A *record lock* locks a row's record

[3]The following descriptions refer to a row's record in the index rather than the row itself, but this should not be confusing as the index record is the real *point of access* to a row.

in the table index, preventing that row from further modification or, importantly, deletion. A *gap lock* locks the gap between index records, so that a new row cannot be inserted there. This prevents the obvious kind of phantom row insertion. A *next-key lock* combines both, using a record lock on an index and a gap lock on the gap before the index. This lock totally prevents phantom reads, even in the edge case of a row being deleted (thereby invalidating its record lock), and then another row being inserted in the resulting gap.

Locking is not the only form of concurrency control, however. InnoDB also uses a technique called *multiversion concurrency control* (MVCC). In MVCC, each session connected to the database sees a snapshot of the database at a particular instant in time. In certain isolation modes, this allows for non-locking reads, also known as *consistent reads*, in which different transactions operate on individual *snapshots* of the database, and attempt to merge their changes after they are done. This technique improves performance, but introduces new concurrency problems which will be addressed later.

In the terminology of the MySQL documentation, "plain `SELECT`" statements use MVCC if possible, while locking `SELECT` statements explicate a physical locking using one of two statements: `LOCK IN SHARE MODE`, which sets shared locks on the rows that the transaction will read, or `SELECT...WHERE...FOR UPDATE`, which sets exclusive locks on the rows matching the WHERE predicate.

6 The SQL Isolation Levels

The concurrency anomalies described earlier show that a major problem in database concurrency is juggling *increased parallelization*, *decreased delay*, and the elimination of *concurrency anomalies*. Toward that end, database designers have codified *isolation levels* which trade varying levels of correctness for increased performance. They are outlined in the SQL standard, from "cleanest" (slowest) to "dirtiest" (fastest), as: `SERIALIZABLE`, `REPEATABLE READ`, `READ COMMITTED`, and `READ UNCOMMITTED`.

6.1 Serializable

The transaction history is serializable in the mathematical sense: any two successfully committed transactions will appear to have executed serially, i.e., one after the other, although which one occurred first might not be predictable in advance.

In this mode in MySQL, all plain `SELECT` statements are converted to locking `SELECT`s, which set shared next-key locks on all rows encountered. This means that another transaction T2 can read the same rows that transaction T1 is reading if, and only if, T1 reads, but does not write to, those rows.

The enforcement of next-key locking throughout the transaction prevents all anomalies: lost updates, non-repeatable reads, phantom reads, and dirty reads[4].

6.2 Repeatable Read

In contrast to `SERIALIAZABLE`, plain selects are not converted to locking selects. Rather, the serializability of a transaction is only enforced when the first plain select is executed, after which all subsequent reads will see the first select's snapshot.

Alternatively, one can force the snapshot by performing a locking read, the form of which branches to two behaviors. If the locking read was of the form `SELECT...WHERE`—i.e., a select on a unique predicate—only record locks are used. If the locking read was of the form `SELECT...WHERE...[field] BETWEEN [lower] AND [upper]`—i.e., a range retrieval—gap and next-key locks are used, thereby preventing phantom reads[5].

As with `SERIALIZABLE`, all anomalies are prevented. The difference is that `SERIALIZABLE` transactions are linearized when they start and the lock is acquired, while `REPEATABLE READ` transactions are linearized at some indeterminate point after the transaction starts, when the first read is executed. But, since entire tables do not need to be locked down, this mode presents a considerable speed improvement over `SERIALIZABLE`, and so is made the default in MySQL.

[4]This and subsequent implementation details taken from [MySQL Manual 13.3.6].

[5]Recall that a phantom read is, by definition, a new row in a range retrieval.

6.3 Read Committed

Further concurrency is allowed in that, within a transaction, two `SELECT` statements can retrieve a different collection of rows if and only if the changes to those rows were *committed* to the database in the meantime. In detail, each plain select statement, even within a single transaction, "sets and reads its own fresh snapshot" [MySQL Manual 14.2.4] via MVCC.

Locking `SELECT`s in this mode use record locks, but not gap/next-key locks, so that non-repeatable reads and phantom reads are possible. But the sudden disappearance/appearance of some rows is acceptable in this mode, as they must have been part of a recently *committed* transaction. However, dirty reads do not occur in this mode because they only arise when a different transaction is in the middle of execution, before it has committed.

6.4 Read Uncommitted

Whereas each plain select in the `READ COMMITTED` mode sets its own snapshot, plain selects in this mode may use earlier snapshots in order to increase performance. Record locks are still used for locking `SELECT`s, as a matter of enforcing basic table integrity. In all other usages, this mode is the same as `READ COMMITTED`.

The usage of earlier snapshots and the lack of next-key locking makes dirty reads, non-repeatable reads, and phantom reads possible[6].

6.5 The Caveats of MVCC Serializability

The SQL standard defined the four isolation levels *in terms* which anomalies they prevented rather than specifying their implementation details [Berenson et al], although it was assumed that they would be implemented with locks. The `SERIALIZABLE` mode, then, would guarantee correctness, since the only way to prevent the anomalies was to lock down entire tables, forcing actual temporal serializability.

However, when MVCC was devised and most major databases began adopting it, it was found that an MVCC implementation of `SERIALIZABLE` could prevent the

[6]Lost updates are prohibited in all modes, also as a matter of basic table integrity.

SQL anomalies but still be incorrect in some cases involving aggregate functions—i.e., one transaction `SELECT`ing the `SUM` of a column while another transaction inserts another row with a value in that column [Berenson et al]. In response, the authors proposed an alternative isolation level for MVCC databases called `SNAPSHOT ISOLATION`. As the MVCC equivalent of `SERIALIZABLE`, it guarantees that "all reads made in a transaction will see a consistent snapshot of the database, and that the transaction will commit only if no updates that it has made conflict with any concurrent updates made since that snapshot". That is, it holds that transactions must see non-overlapping snapshots of the database in time.

The distinction between `SERIALIZABLE` and `SNAPSHOT`, and thus between the locking and MVCC philosophies, is akin to that between pessimistic and optimistic locking: `SERIALIZABLE` locks beforehand to ensure correctness, while `SNAPSHOT` immediately makes changes to the transaction's uniquely held snapshot, but is able to fail once it attempts to commit to the database. If a transaction fails, MySQL forces a `rollback`, so that the user application can retry the transaction or do something else.

It is theoretically possible for MVCC databases to enforce serializable correctness using a combination of snapshot isolation and *predicate locking*, which prevents transaction A from inserting or modifying rows that transaction B has or will retrieve in the `WHERE` condition of one of its select clauses. But predicate locking is computationally expensive because it involves keeping track of all `SELECT...WHERE` statements being performed globally. So, instead, in many MVCC databases, the `SERIALIZABLE` isolation level is not true serializability but snapshot isolation. This is a point of confusion among database professionals, as these implementations are correct under the SQL definition of serializability, but not under the time-tested mathematical definition.

It is left for the database professional to understand how each of the major databases implements the `SERIALIZABLE` mode. *Oracle Database*, the most popular, implements snapshot isolation by default. One enables true serializability by manually locking tables. *Microsoft SQL Server*, the second most popular, implements true serializability. A separate `SNAPSHOT ISOLATION` level is available for performance. *PostgreSQL*, the fourth most popular database, implemented snapshot isolation prior to version 9.1 (September 2011), but afterward guaranteed serializable-like correctness using a new technique from contemporary research called *serializable snapshot isolation*, which uses snapshot isolation for speed but adds additional checking for anomalies. This checking is based on the observation that all snapshot isolation anomalies correspond to a *cycle* in the timeline of compounded snapshots.

Lastly, InnoDB, as the default MySQL storage engine, guarantees serializable-like correctness using a combination of snapshot isolation, next-key locking, and predicate locking[7].

7 Pertinent SQL Statements

One sets the transaction level using the SQL statement:

```
SET [GLOBAL/SESSION]
TRANSACTION ISOLATION LEVEL
[SERIALIZABLE / READ COMMITTED / READ UNCOMMITTED / SERIALIZABLE],
```

where terms in brackets are mutually exclusive options.

A *session* in MySQL begins when the connection between client and server is established, continues for as long as the user executes queries and transactions by sending them to the server, and ends when the client disconnects. Isolating a transaction level to a session can be very useful, in situations such as when the global database needs few correctness guarantees and only requires serializability for, say, sessions that process credit card purchases.

One starts a transaction using the `START TRANSACTION` statement, and ends it using `COMMIT` or `ROLLBACK`. Alternatively, one can `SET autocommit=0`, so that the changes made by any number of previously executed statements are not saved to the physical database, i.e., to disk, until the `COMMIT` statement is executed. By default, `autocommit` is set to `true`, so that all individual statements, outside of a transaction, are saved to disk immediately after execution. That is, each statement is its own transaction.

8 Closing Remarks

I, the author of this paper, learned a lot about database concurrency in the course of writing this paper. I believe that I have gone as far as possible in the goal of introducing

[7]It is important to note that, prior to MySQL 5.5 (December 2010), this implementation was buggy and was not trusted to provide serializable correctness: stackoverflow.com/questions/6269471/does-mysql-innodb-implement-true-serializable-isolation.

a hypothetical CECS 323 and 327 student to database concurrency concepts without delving too deeply into software-specific implementation details. That is, the citation of MySQL implementation details hopefully has been just enough to show the reader the variety of approaches one can take to implement database concurrency, without detracting from the main idea that this is a field rife with opportunity for innovation.

References

Herlihy, Maurice, and Nir Shavit. The Art of Multiprocessor Programming. Amsterdam: Elsevier/Morgan Kaufmann, 2008.

Rob, Peter, Carlos Coronel, and Steven Morris. Database Systems: Design, Implementation, and Management. Boston: Cengage Learning, 2012.

Schwartz, Baron and Ewen Fortune. ”Forecasting MySQL Scalability with the Universal Scalability Law.” Percona.com. Percona, 2014.

Codd, E. F. “A Relational Model of Data for Large Shared Data Banks.” Communications of the ACM 13.6 (1970): 377-87.

Härder, Theo, and Andreas Reuter. ”Principles of Transaction-oriented Database Recovery.” ACM Computing Surveys 15.4 (1983): 287-317.

“MySQL 5.6 Reference Manual” MySQL
Developer Zone. Oracle Corporation, 2014.

Berenson, Hal, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. ”A Critique of ANSI SQL Isolation Levels.” ACM SIGMOD Record 24.2 (1995): 1-10.