California State University, Long Beach

CECS 327

Net-centric Computing

MySQL Concurrency

Author:
Tan Tran

Supervisor:
Dr. Burkhard Englert

April 30, 2014

Contents

1 Introduction

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 user requests on one server. I will learn about the latter by documenting concurrency principles and practice in MySQL, the most popular open source relational database. The problems addressed are just as fascinating as those in multiprocessor programming.

1.1 The Rigors of Database Concurrency

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 forewent 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". 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, banking databases must implement money transfers as atomic 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 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 each transaction brings the database from one correct state to another. Transactions are **i**solated if transaction B does not see that 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 non-concurrent 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 fetches from history only the rows matching the global timestamp.

But concurrency makes ensuring ACID more difficult, and adds an additional *serializability* requirement: that 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 serializability is illustrated in the following problems.

1.1.1 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. This problem is a *write anomaly*; the following are *read anomalies*.

1.1.2 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.

1.1.3 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 credits them with interest, so that Edward now has $105. The transaction is about to apply the service fee, when Edward makes a deposit of $95 and so 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.

1.1.4 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 maintain 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 thus has not been charged in (2), but nevertheless sees the message in (3). His row is a phantom which did not appear in the first range retrieval, but did in the second.

1.2 The Harsh Realities of Ensuring ACID

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. (More to come - this is in the mutual exclusion chapter)

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, table locks, but also buffers, access to the SQL parser, backup logs, etc.

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.

There is a lot of literature out there that analyzes the scaling of database performance. But out of all techniques, the one most applicable to the Net-centric Computing class is the *Universal Scalability Law* because it is Amdahl's Law with an additional *coherency penalty*:

TODO: Some defns from Percona paper. Paraphrase later.

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

TODO: Write equation like Amdahl's in textbook

where N is the number of processes available to process transactions, 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*. In empirical analysis, we include queuing delay in coherency delay, although a more accurate mathematical model would not scale queuing delay as N(N-1), but would rather use Kendall's M/M/c model.

TODO: Example, explanation.

2 Mutual Exclusion

2.0.1 History and important concepts

I outline important concepts before describing which locks are used in each transaction level.

MySQL is unique among the major databases for featuring multiple storage engines. MySQL started development with the MyISAM engine, which did not support transactions and locked on the table level. (Other engines here, inconsequential). At the time of MySQL 5.1, Oracle, a competing company developed the InnoDB engine, which supported transactions and row-level locks, and allowed for explicit locking via SQL queries. Oracle later acquired Sun, which owned MySQL, and proceeded to make InnoDB the default MySQL storage engine. It is now recommended for small and large applications. Notably, Google and Facebook, among others, use it and have contributed to its development.

There are two types of row-level locks in InnoDB, taken from database research. A *shared* (S) lock permits a transaction to read a row (MySQL ref manual). A *exclusive* (X) lock permits a transaction to update or delete a row. Their names are self explanatory; multiple transactions can have shared locks on a row, but not on an exclusive lock.

Additionally, an exclusive lock on a tuple should prevent an exclusive lock on the whole relation. To implement this, all locks must be accompanied by *intention locks* on tables. A transaction holds a *shared intention lock* on a table if the only locks it intends to set are S locks. It holds a *intention exclusive* lock if it intends to set at least one X lock.

Locking is not the only form of concurrency control, however. InnoDB also uses a database concept 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, aka consistent reads, where different transactions operate on different snapshots in parallel, and merge their changes after they are done. There are some complicated protocols involved, which I will discuss later.

MVCC improves speed, but does not prevent all types of anomalies, so MySQL may combine it with record locks, gap locks, and next-key locks. A *record lock* locks a row, aka a record in the index. A *gap lock* locks the gap between index

records. A *next-key lock* combines a record lock on an index and a gap lock on the gap before the index record.

2.0.2 High-level Concerns

(Transition from last section - this is how we make use of the above concepts) The preceding examples illustrate that a big 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" to "dirtiest", as: `SERIALIZABLE`, `REPEATABLE READ`, `READ COMMITTED`, and `READ UNCOMMITTED`.

2.0.3 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. This is implemented with...

2.0.4 Read Committed

Statements cannot read data that has been modified by ongoing transactions.

But, during a transaction, data can be *changed* by other transactions. That is, allow non-repeatable reads and phantom reads, but disallow dirty reads. This is implemented with...

2.0.5 Repeatable Read

Statements cannot read modified data, nor write data being read by other transactions. However, there are no *range* locks, so that phantom reads are allowed, even if non-repeatable reads and dirty reads are not. This is implemented with...

2.0.6 Read Uncommitted

This level provides the most concurrency and the least protection. Statements can read data that has been modified by ongoing transactions. SELECT statements

are performed in a nonlocking fashion (MySQL doc), although a possible earlier version of a row might be used.

This makes dirty reads, non-repeatable reads, and phantom reads possible. This is implemented with...

2.0.7 Notes on MVCC Serializability

The SQL standard was written with locking databases in mind, and defined the four isolation levels *in terms* which anomalies they prevented. That led to the assumption that `SERIALIZABLE` would then guarantee correctness, since the only way to prevent the named anomalies was to lock down entire tables, forcing actual temporal serializability.

However, when most major databases began using MVCC, it was found that an MVCC implementation of `SERIALIZABLE` could prevent the named 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, Berenson et al proposed an alternative isolation level for MVCC databases called `SNAPSHOT ISOLATION`. This level 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, transactions see non-overlapping snapshots of the database in time.

TODO: detailed explanation why.

As others describe, the distinction between `SERIALIZABLE` and `SNAPSHOT` is akin to that between pessimistic and optimistic locking: `SERIALIZABLE` locks beforehand to ensure correctness, while `SNAPSHOT` immediately makes changes to a snapshot first, but is able to fail once it attempts to commit to the database.

It is theoretically possible for MVCC databases to give the appearance of true serializability using a combination of MVCC and *predicate locking*, which prevents phantom reads by preventing one transaction from inserting or modifying a row that would have matched the `WHERE` condition of another transaction's `SELECT` operation. 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 formal definition (see Herlihy & Wing).

The most popular databases differ on this issue. *Oracle Database*, the most popular database, 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). Afterward, it practically implemented true serializability using a new technique called *serializable snapshot isolation*, which uses MVCC for speed but adds additional checking for anomalies.

Lastly, as it pertains to this paper, MySQL is the third most popular database. Its InnoDB engine has always aimed to give the appearance of true serializability, using a combination of MVCC, index locking, and predicate locking. Prior to MySQL 5.5 (December 2010), this implementation was buggy, as illustrated as follows. Consider client A running a transaction in which it will `SELECT` all rows `WHERE x = 1`. Then, client B runs a transaction to insert a row with `x = 2, y = 10`. This statement passes predicate locking, as it does not affect A's `WHERE` clause. But in this buggy implementation, next-key locking only locks the next key of the *most recently inserted row*. So, client B then inserts another row with `x = 1, y = 20`, which should not pass, but does because (to figure out).

3 Practicum

To do.

4 Conclusion

To do.

References

[Maurice Herlihy and Nir Shavit] The Art of Multiprocessor Programming. Amsterdam; London: Elsevier/Morgan Kaufmann, 2012. Print.

[Edgar Codd] A Relational Model of Data for Large Shared Data Banks.

[Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil] A critique of ANSI SQL isolation levels: ACM, SIGMOD '95 Proceedings of the 1995 ACM SIGMOD. 1995.

[Herlihy, Maurice and Jeanette Wing] Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Prog. Lang. and Sys. Vol. 12, No. 3, July 1990, Pages 463-492. URL http://www.cs.brown.edu/ mph/HerlihyW90/p463-herlihy.pdf

[Various MySQL textbooks and documentation; to cite]