California State University, Long Beach

CECS 327

Net-centric Computing

The Art of Database Concurrency

Author:
Tan Tran

Supervisor:
Dr. Burkhard Englert

April 22, 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, that have made it physically possible for databases to service so many transactions at once.

The pressure to innovate has thus spawned challengers to the relational database establishment for the business of the highest throughput applications. They are collectively termed *NoSQL* databases, and work by relaxing certain correctness constraints that relational databases are founded on and thus cannot forego. But, in a sense, industry-strength relational databases have anticipated that challenge for many years now: by seeking to ensure database correctness in the case of concurrent access by two or three, ten or a hundred clients, they have put into place, if not implemented, the mechanisms to ensure correctness for thousands and millions of concurrent accesses.

It stands that relational databases have work to do as technology progresses, both by developing novel concurrency solutions and by adapting techniques from multiprocessor programming. I consider the latter approach by relating the techniques taught in the CSULB Net-centric Computing class to those implemented in Oracle MySQL, the most popular relational database and arguably one of the most innovative in response to concurrency concerns. I build upon the things I learned in the Database Fundamentals class and parallel the things I am learning in Net-centric Computing, in particular by matching database concepts to the *Art of Multiprocessor Programming* textbook, chapter by chapter.

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". It was left for industry, rather, to implement Codd's ideas and realize the importance of that statement. In particular, the implementation of *transactions* would come to be more important than had been anticipated; one traded their usefulness as a description of a *unit* of work for the challenges of ensuring database consistency between and within 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 part-way 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)

Further and further optimization approaches a theoretical speedup limit, as measured with Amdahl's Law, described in *AoMP*, section 1.5. However, databases are more susceptible to coherency delay, the work that the system must perform to synchronize shared data, than sequential programs, as Amdahl's Law is made to model. In that case, Neil Gunther's *Universal Scalability Law* presents a more accurate measure of scalability. It is Amdahl's Law with an additional *coherency penalty*:

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

TODO: Some defns from Percona paper. Paraphrase later.

TODO: Write equation like Amdahl's in textbook

TODO: Example.

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 *coherency delay*, which in databases is often mutex locking.

The coherency parameter reconciles *Chapter 1: Introduction*'s coverage of Amdahl's Law with *Chapter 7: Spin Locks and Contention*'s coverage of caching and memory locality.

2 Mutual Exclusion

The central problem of database concurrency, then, is reconciling the scalability goals of *increased parallelization* and *decreased coherency delay* with the correctness goals of eliminating *concurrency anomalies*. Toward that end, database designers have codified *isolation levels* which trade varying levels of mutual exclusion for increased performance. They are outlined in the SQL standard, from ”dirtiest” to ”cleanest”, as:

Read Uncommitted Statements can read data that has been modified by ongoing transactions. That is, allow dirty reads, non-repeatable reads, and phantom reads.

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.

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.

Serializable Repeatable read, with additional *range locking* to make phantom reads impossible. This makes all transactions linearizable as defined in *AoMP*.

TODO: cover lost updates for all 4

Each level is implemented with different types of mutual exclusion locks

3 Conclusion

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

Aliquam lectus. Vivamus leo. Quisque ornare tellus ullamcorper nulla. Mauris porttitor pharetra tortor. Sed fringilla justo sed mauris. Mauris tellus. Sed non leo. Nullam elementum, magna in cursus sodales, augue est scelerisque sapien, venenatis congue nulla arcu et pede. Ut suscipit enim vel sapien. Donec congue. Maecenas urna mi, suscipit in, placerat ut, vestibulum ut, massa. Fusce ultrices nulla et nisl.

References

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