CONCURRENCY CONTROL SCHEMES IN NEWSQL SYSTEMS

Sanket Lokegaonkar¹, Anish Joshi²

(Computer Engineering, University of Mumbai)

ABSTRACT

OLTP applications, in the current times, require support for ACID transactions and concurrency. But these properties each come with their trade-off. NewSQL systems have emerged as the solution to this conundrum, by overhauling the traditional architecture of RDBMSs. Concurrency control scheme selected has a major impact on the architectural performance. In this paper we firstly discuss the older concurrency control schemes used in the conventional SQL systems like two phase locking protocol and later discuss the modern concurrency control schemes used mainly in the modern NewSQL based systems such as multi version concurrency control, basic timestamp concurrency control, optimistic concurrency control, Timestamp ordering with partition level locking. Later, we have discussed the contemporary NewSQL systems with emphasis on underlying architecture and schemes used for controlling concurrency.

Keywords: NewSQL, Concurrency Control, MVCC, OLTP.

1. INTRODUCTION

Concurrency Control is the process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions. Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems: lost updates, uncommitted data, and inconsistent retrievals. So, concurrency control is an essential element for correctness in any system where two or more database transactions are executed with time overlap and both have access to the same data.[1]

Maintaining concurrency in a database system is of utmost importance otherwise it might raise issues while allowing simultaneous access to shared entities be they objects, data records, or some other representations. A database system with poor concurrency control schemes might cause collisions. A collision is said to occur when two activities, which may or may not be full-fledged transactions, attempt to change entities within a system of record. Collisions will occur if the data is
Concurrency control is one of the main issues in the studies of real-time database systems. Many real-time concurrency control methods are based only on the pessimistic two-phase locking (2PL), where a transaction acquires a lock before any database operation and waits for the lock if it cannot be granted. However, 2PL has some inherent problems such as the possibility of deadlocks and unpredictable blocking time. These problems appear to be serious in real-time systems since real-time transactions need to meet their timing constraints, in addition to consistency requirements. Thus to handle the above problems a new contender has emerged in the database ring which provides means to handle concurrency better than traditional SQL based systems. [3]

NewSQL based systems are designed especially to maintain ACID guarantees of a traditional system and to provide scalable performance achieved by the present NoSQL systems. Such systems are capable of high throughput as the present NoSQL solutions and they also do not need application-level consistency code. Moreover, they preserve the high-level language query capabilities of SQL. Thus such systems have found use in latest real time applications because of the advantages it has over the traditional SQL systems. Examples of NewSQL systems include Clustrix, NimbusDB, and VoltDB. [4] Following mentioned are the characteristics of NewSQL systems that are observed:

i. SQL as the primary mechanism for application interaction.
ii. ACID support for transactions.
iii. A non-locking concurrency control mechanism so real-time reads will not conflict with writes, and thus cause them to stall.
iv. An architecture providing much higher per-node performance than available from traditional RDBMS solutions.
v. A scale-out, shared-nothing architecture, capable of running on a large number of nodes without suffering bottlenecks. [5]

The category of NewSQL data stores, on the other hand, is being used to classify a set of solutions aimed at bringing to the relational model the benefits of horizontal scalability and fault tolerance provided by NoSQL solutions. The first use of the term is attributed to a report of the 451 group in 2011 [6]. The Google Spanner [7] solution is considered to be one of the most prominent representatives of this category, as is also VoltDB [8], which is based on the H-Store [9] research project. Clustrix [10] and NuoDB [11] are two commercial projects that are also classified as NewSQL. All these data stores support the relational model and use SQL as their query language, even though they are based on different assumptions and architectures than traditional RDBMSs.

2. CONCURRENCY CONTROL SCHEMES IN TRADITIONAL DISTRIBUTED DATABASE SYSTEMS

2.1. TWO PHASE LOCKING

Two-phase locking (2PL) was the first provably correct method of ensuring the correct execution of concurrent transactions in a database system. Under this scheme, transactions have to acquire locks for a particular element in the database before they are allowed to execute a read or write operation on that element. The transaction must acquire a read lock before it is allowed to read the element and similarly it must acquire a write lock in order to modify that element. The DBMS maintains locks for either each tuple or at a higher logical level (e.g., tables, partitions).
The ownership of locks is governed by two rules: (1) different transactions cannot simultaneously own conflicting locks, and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks. A read lock on an element conflicts with a write lock on that same element. Likewise, a write lock on an element conflicts with a write lock on the same element. In the first phase of 2PL, known as the growing phase, the transaction is allowed to acquire as many locks as it needs without releasing locks. When the transaction releases a lock, it enters 2PL. When the transaction terminates (either by committing or aborting), all the remaining locks are automatically released back to the coordinator. 2PL is considered a pessimistic approach in that it assumes that transactions will conflict and thus they need to acquire locks to avoid this problem. If a transaction is unable to acquire a lock for an element, then it is forced to wait until the lock becomes available. If this waiting is uncontrolled (i.e., indefinite), then the DBMS can incur deadlocks. Thus, a major difference among the different variants of 2PL is in how they handle deadlocks and the actions that they take when a deadlock is detected. Now, Different versions of 2PL are specified:

i. 2PL with Deadlock Detection (DL_DETECT):

The DBMS monitors a waits-for graph of transactions and checks for cycles (i.e., deadlocks). When a deadlock is found, the system must choose a transaction to abort and restart in order to break the cycle. In practice, a centralized deadlock detector is used for cycle detection. The detector chooses which transaction to abort based on the amount of resources it has already used (e.g., how long it has been running) to minimize the cost of restarting a transaction.

ii. 2PL with Non-waiting Deadlock Prevention (NO_WAIT):

Unlike deadlock detection where the DBMS waits to find deadlocks after they occur, this approach is more cautious in that a transaction is aborted when the system suspects that a deadlock might occur. When a lock request is denied, the scheduler immediately aborts the requesting transaction (i.e., it is not allowed to wait to acquire the lock).

iii. 2PL with Waiting Deadlock Prevention (WAIT_DIE):

This is a non-preemptive variation of the NO_WAIT scheme technique where a transaction is allowed to wait for the transaction that holds the lock that it needs if that transaction is newer than the one that holds the lock. If the requesting transaction is older, then it is aborted (hence the term “dies”) and is forced to restart. Each transaction needs to acquire a timestamp before its execution and the timestamp ordering guarantees that no deadlocks can occur. [12]

3. CONCURRENCY CONTROL SCHEMES USED IN MODERN NEWSQL AND DISTRIBUTED DATABASE SYSTEMS

3.1. MVCC (Multi Version Concurrency Control)

MVCC is a concurrency control method used to provide concurrent access to data without locking the data. It allows readers to get consistent data without blocking the users to update the data. It provides readers a consistent view of the data by maintaining previous versions of the data. MVCC provides each transaction a snapshot of the data thus each transaction gets a consistent view of the database. All reads in MVCC are consistent to some point in time. In most database systems that have implemented MVCC, the point in time where the snapshot of the data is taken is defined by the isolation level, e.g.
i. Serializable: Point in time for MVCC snapshot is the start of the transaction, providing transaction level read consistency.

ii. Read Committed: Point in time MVCC snapshot is the start of the query, providing statement level read consistency.[13]

Instead of overwriting existing documents, a completely new version of the document is generated. The two benefits are:

i. Objects can be stored coherently and compactly in the main memory.

ii. Objects are preserved-isolated writing and reading transactions allow accessing these objects for parallel operations.

The system collects obsolete versions as garbage, recognizing them as forsaken. Garbage collection is asynchronous and runs parallel to other processes, thus requires no additional time or effort.[14]

3.2. Basic Timestamp Concurrency Control

In Basic Timestamp Concurrency Control every time a transaction reads or modifies a tuple in the database, the DBMS compares the timestamp of the transaction with the timestamp of the last transaction that reads or writes the same tuple. For a read or write operation, the DBMS rejects the request if the transaction’s timestamp is less than the timestamp of the last write to that tuple. Likewise, for a write operation, the DBMS rejects it if the transaction’s timestamp is less than the timestamp of the last read to that tuple.[12]

In timestamp based concurrency control algorithms, each site maintains a logical clock. This clock is incremented when a transaction is submitted at that site and updated whenever the site receives a message with a higher clock value. Each transaction is assigned a unique timestamp and conflicting actions are executed in order of the timestamp of their transactions.

Timestamps can be used in two ways:

i. To determine the currency of a request with respect to the data object it is operating on.

ii. To order events with respect to one another.

In timestamp based concurrency control algorithms, the serialization order of transactions is selected a priori, and transactions are forced to follow this order.[15]

3.3. Optimistic Concurrency Control

The DBMS tracks the read/write sets of each transaction, and stores all of their write operations in their private workspace. When a transaction commits, the system determines whether that transaction’s read set overlaps with the write set of any concurrent transactions. If this validation succeeds, then the DBMS applies the changes from the transaction’s workspace into the database; otherwise, the transaction is aborted and restarted. The advantage of this approach for main memory DBMSs is that transactions write their updates to shared memory only at commit time, and thus the contention period is short.[12]

Optimistic concurrency control protocols have the nice properties of being non-blocking and deadlock-free. These properties make them especially attractive for real-time database systems. Because conflict resolution between the transactions is delayed until a transaction is near to its completion, there will be more information available in making the conflict resolution. Although optimistic approaches have been shown to be better than locking protocols for RTDBS(Real-Time Database System), they have the problem of unnecessary restarts and heavy restart overhead. This is due to the late conflict detection that increases the restart overhead since some near to–complete transactions have to be restarted.[16]
Optimistic concurrency control or optimistic locking assumes that conflicts are possible, but rare. Therefore, instead of locking the record, the data store checks at the end of the operation to determine whether concurrent users have attempted to modify the same record. If a conflict is identified, different conflict-resolution strategies can be used, such as failing the operation immediately or retrying one of the operations.[17]

The major overhead incurred by optimistic concurrency control algorithms is due to transaction rollback. Since such algorithms detect conflict only after a transaction is run to completion, a significant amount of wasted computation may be incurred. In the multi-version algorithms presented in this paper read-only transactions are never rolled back due to conflicts. Any transaction may be rolled back due to premature compaction, but the probability of this happening can be brought arbitrarily close to zero by increasing the time between successive compactions. [18]

3.4. T/O with Partition-Level Locking (H-Store)

The database is divided into disjoint subsets of memory called partitions. Each partition is protected by a lock and is assigned a single-threaded execution engine that has exclusive access to that partition. Each transaction must acquire the locks for all of the partitions that it needs to access before it is allowed to start running. This requires that the DBMS has knowledge of what partitions that each individual transaction will access before it begins. When a transaction request arrives, the DBMS assigns it a timestamp and then adds it to all of the lock acquisition queues for its target partitions. The execution engine for a partition removes a transaction from the queue and grants it access to that partition if the transaction has the oldest timestamp in the queue.[12]

A single H-Store instance is a cluster of two or more computational nodes deployed within the same administrative domain. A node is a single physical computer system that hosts one or more sites. A site is the basic operational entity in the system; it is a single-threaded daemon that an external OLTP application connects to in order to execute a transaction. We assume that the typical H-Store node has multiple processors and that each site is assigned to execute on exactly one processor core on a node. Each site is independent from all other sites, and thus does not share any data structures or memory with collocated sites running on the same machine. Every relation in the database is divided into one or more partitions. A partition is replicated and hosted on a multiple sites, forming a replica set.

When a running transaction executes a SQL command, it makes an internal request through an H-Store API. At this point, all of the parameter values for that query are known by the system, and thus the plan is annotated with the locations of the target sites for each query sub-operation. The updated plan is then passed to a transaction manager that is responsible for coordinating access with the other sites. The manager transmits plan fragments to the appropriate site over sockets and the intermediate results (if any) are passed back to the initiating site. H-Store uses a generic distributed transaction framework to ensure serializability.

The data storage backend for H-Store is managed by a single-threaded execution engine that resides underneath the Transaction manager. Each individual site executes an autonomous instance of the storage engine with a fixed amount of memory allocated from its host machine. Multi-site nodes do not share any data structures with collocated sites, and thus there is no need to use concurrent data structures.

The performance of distributed main memory databases is dependent on the data placement strategies and supporting data structures of the physical database design. But because OLTP systems repeatedly execute the same set of transactions, the design can be optimized to execute just these transactions while completely ignoring ad hoc queries. H-Store supports the execution of the latter, but it provides no guarantees that the queries are executed in a timely manner. [19]
4. SOME OF THE CONTEMPORARY NEWSQL DATABASE SYSTEMS EVOLVED WITH THE MECHANISMS DISCUSSED EARLIER

4.1. ClustrixDB

ClustrixDB uses a combination of Multi-Version Concurrency Control (MVCC) and 2 Phase Locking (2PL) to support mixed read-write workloads. In ClustrixDB, readers enjoy lock-free snapshot isolation while writers use 2PL to manage conflict. The combination of concurrency controls means that readers never interfere with writers (or vice-versa), and writers use explicit locking to order updates. ClustrixDB implements a distributed MVCC scheme to ensure that readers are lockless, so that readers and writers never interfere with each other. As writers modify rows within the system, ClustrixDB maintains a version history of each row. Each statement within a transaction uses lock-free access to the data to retrieve the relevant version of the row.[20]

4.2. VoltDB

VoltDB executes all operations in a deterministic order. Previous work in this area has typically used time as the ordering mechanism (timestamp order); however, VoltDB does not use a clock-based scheme. A single-node transaction is examined in the user-space VoltDB client library, where parameters are substituted to form a runnable transaction. The client library is aware of VoltDB sharding, so the transaction can be sent to a node controller at the correct node. This node controller serializes all single-node transactions and they are executed from beginning to end in this order without any blocking. Any application that consists entirely of single-node transactions will obtain maximum parallelism and, in theory, run all CPUs at 100% utilization, as long as the sharding achieves good load balance. Other kinds of transactions are sent to a special global controller, which is responsible for deciding a serial order. The various SQL commands in such a transaction must be sent to the correct sites, where they are inserted into the single-node transaction stream at any convenient place. Hence, they are arbitrarily interleaved into the single-node transaction mix. Various optimizations are implemented and others are planned for the classes of more general transactions. [21]

4.3. Google Spanner

Spanner is Google’s scalable, multi-version, globally distributed, and synchronously-replicated database. At the highest level of abstraction, it is a database that shards data across many sets of Paxos state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows. TrueTime API is one of the key enabler which allows Spanner to handle concurrency control in an OLTP system scaled over Paxos state machines in datacenters spread all over the world. The API directly exposes clock uncertainty, and the guarantees on Spanner’s timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty. Google’s cluster-management software provides an implementation of the TrueTime API. TrueTime is implemented by a set of time master machines per datacenter and a timeslave daemon per machine. Every daemon polls a variety of masters to reduce vulnerability to errors from any one master. Daemons apply a variant of Marzullo’s algorithm to detect and reject liars, and synchronize the local machine clocks to the nonliars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted.[7]
5. CONCLUSION

For today’s modern database systems, having large datasets and huge number of users concurrency is one of the major issues to deal with so that the transactions can perform concurrently, without interfering with one another (isolation from other transactions), and also to maintain consistency and integrity of the database system. Thus, the solution to the above mentioned issue is to use modern NewSQL systems (which support the ACID properties of traditional SQL systems and offer higher throughput as compared to their SQL rivals) along with modern concurrency control schemes mentioned in this paper. Due to increasing growth of data in OLTP systems, traditional database system cannot be efficaciously scaled over it. Concurrency control is the deciding factor which keeps in check the efficient implementation of ACID guarantees made by the NewSQL systems. Using NewSQL systems with the modern concurrency control schemes can vastly improve performance in large-scale OLTP applications which demand consistency as paramount.

REFERENCES


