Multiversion concurrency control

Multiversion concurrency control (**MCC** or **MVCC**), is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.[1]

Without concurrency control, if someone is reading from a database at the same time as someone else is writing to it, it is possible that the reader will see a half-written or inconsistent piece of data. For instance, when making a wire transfer between two bank accounts if a reader reads the balance at the bank when the money has been withdrawn from the original account and before it has deposited in the destination account, it would seem that money has disappeared from the bank. Isolation is the property that provides guarantees in the concurrent accesses to data. Isolation is implemented by means of a concurrency control protocol. The simplest way is to make all readers wait until the writer is done, which is known as a read-write lock. Locks are known to create contention especially between long read transactions and update transactions. MVCC aims at solving the problem by keeping multiple copies of each data item. In this way, each user connected to the database sees a *snapshot* of the database at a particular instant in time. Any changes made by a writer will not be seen by other users of the database until the changes have been completed (or, in database terms: until the transaction has been committed.)

When an MVCC database needs to update a piece of data, it will not overwrite the original data item with new data, but instead creates a newer version of the data item. Thus there are multiple versions stored. The version that each transaction sees depends on the isolation level implemented. The most common isolation level implemented with MVCC is snapshot isolation. With snapshot isolation, a transaction observes a state of the data as when the transaction started. MVCC introduces the challenge of how to remove versions that become obsolete and will never be read. In some cases, a process to periodically sweep through and delete the obsolete versions is implemented. This is often a stop-the-world process that traverses a whole table and rewrites it with the last version of each data item. PostgreSQL adopts this approach with its VACUUM process. Other databases split the storage blocks into two parts: the data part and an undo log. The data part always keeps the last committed version. The undo log enables recreation of older versions of data. The main inherent limitation of this latter approach is that when there are update-intensive workloads, the undo log part runs out of space and then transactions are aborted as they cannot be given their snapshot. For a document-oriented database it also allows the system to optimize documents by writing entire documents onto contiguous sections of disk—when updated, the entire document can be re-written rather than bits and pieces cut out or maintained in a linked, non-contiguous database structure.

MVCC provides point-in-time consistent views. Read transactions under MVCC typically use a timestamp or transaction ID to determine what state of the DB to read, and read these versions of the data. Read and write transactions are thus isolated from each other without any need for locking. However, despite locks being unnecessary, they are used by some MVCC databases such as Oracle. Writes create a newer version, while concurrent reads access an older version.

Contents

Implementation

MVCC uses timestamps (**TS**), and *incrementing transaction IDs*, to achieve *transactional consistency*. MVCC ensures a transaction (**T**) never has to wait to *Read* a database object (**P**) by maintaining several versions of the object. Each version of object **P** has both a *Read Timestamp* (**RTS**) and a *Write Timestamp* (**WTS**) which lets a particular transaction $\mathbf{T_i}$ read the most recent version of the object which precedes the transaction's *Read Timestamp* $\mathbf{RTS(T_i)}$.

If transaction $\mathbf{T_i}$ wants to *Write* to object **P**, and there is also another transaction $\mathbf{T_k}$ happening to the same object, the Read Timestamp $\mathbf{RTS(T_i)}$ must precede the Read Timestamp $\mathbf{RTS(T_k)}$, i.e., $\mathbf{RTS(T_i)} < \mathbf{RTS(T_k)}$, for the object *Write Operation* (**WTS**) to succeed. A *Write* cannot complete if there are other outstanding transactions with an earlier Read Timestamp (**RTS**) to the same object. Like standing in line at the store, you cannot complete your checkout transaction until those in front of you have completed theirs.

To restate; every object (**P**) has a *Timestamp* (**TS**), however if transaction $\mathbf{T_i}$ wants to *Write* to an object, and the transaction has a *Timestamp* (**TS**) that is earlier than the object's current Read Timestamp, $\mathbf{TS(T_i)} < \mathbf{RTS(P)}$, then the transaction is aborted and restarted. (This is because a later transaction already depends on the old value.) Otherwise, $\mathbf{T_i}$ creates a new version of object **P** and sets the read/write timestamp **TS** of the new version to the timestamp of the transaction $\mathbf{TS}=\mathbf{TS(T_i)}$.[2]

The drawback to this system is the cost of storing multiple versions of objects in the database. On the other hand, reads are never blocked, which can be important for workloads mostly involving reading values from the database. MVCC is particularly adept at implementing true snapshot isolation, something which other methods of concurrency control frequently do either incompletely or with high performance costs.

Examples

Concurrent read-write

At Time = 1, the state of a database could be:

Time	Object 1	Object 2
0	"Foo" by T0	"Bar" by T0
1	"Hello" by T1	

T0 wrote Object 1="Foo" and Object 2="Bar". After that T1 wrote Object 1="Hello" leaving Object 2 at its original value. The new value of Object 1 will supersede the value at 0 for all transactions that start after T1 commits at which point version 0 of Object 1 can be garbage collected.

If a long running transaction T2 starts a read operation of Object 2 and Object 1 after T1 committed and there is a concurrent update transaction T3 which deletes Object 2 and adds Object 3="Foo-Bar", the database state will look like at time 2:

Time	Object 1	Object 2	Object 3
0	"Foo" by T0	"Bar" by T0	
1	"Hello" by T1		
2		(deleted) by T3	"Foo-Bar" by T3

There is a new version as of time 2 of Object 2 which is marked as deleted and a new Object 3. Since T2 and T3 run concurrently T2 sees the version of the database before 2 i.e. before T3 committed writes, as such T2 reads Object 2="Bar" and Object 1="Hello". This is how multiversion concurrency control allows snapshot isolation reads without any locks.

History

Multiversion concurrency control is described in some detail in the 1981 paper "Concurrency Control in Distributed Database Systems"[3] by Phil Bernstein and Nathan Goodman, then employed by the Computer Corporation of America. Bernstein and Goodman's paper cites a 1978 dissertation[4] by David P. Reed which quite clearly describes MVCC and claims it as an original work.

The first shipping, commercial database software product featuring MVCC was Digital's VAX Rdb/ELN. The second was InterBase, both of which are still active, commercial products.

See also

- List of databases using MVCC
- Read-copy-update
- Timestamp-based concurrency control
- Vector clock
- Version control system

References

1. "Clojure - Refs and Transactions" (https://clojure.org/reference/refs). *clojure.org*. Retrieved 2019-04-12.
2. Ramakrishnan, R., & Gehrke, J. (2000). Database management systems. Osborne/McGraw-Hill.
3. Bernstein, Philip A.; Goodman, Nathan (1981). "Concurrency Control in Distributed Database Systems" (http://portal.acm.org/citation.cfm?id=356842.356846). *ACM Computing Surveys*.
4. Reed, David P. (September 21, 1978). "Naming and Synchronization in a Decentralized Computer System" (http://www.lcs.mit.edu/publications/specpub.php?id=773). *MIT dissertation*.

Further reading

- Gerhard Weikum, Gottfried Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann, 2002, ISBN 1-55860-508-8