A Quick Survey of MultiVersion Concurrency Algorithms

Dibyendu Majumdar
dibyendy@mazumdar.demon.co.uk

Revised November 4, 2007

Abstract

This document looks at some of the problems with traditional methods of concurrency control via locking, and explains how Multi-Version Concurrency algorithms help to resolve some of these problems. It describes the approaches to Multi-Version concurrency and examines a couple of implementations in greater detail.

1 Introduction

Database Management Systems must guarantee consistency of data while allowing multiple transactions to read/write data concurrently. Classical DBMS [1] implementations maintain a single version of the data and use locking to manage concurrency. Examples of SVDB implementations are IBM DB2, Microsoft SQL Server, Apache Derby and Sybase.

To understand how Classical DBMS implementations solve concurrency problems with locking, it is necessary to first understand the type of problems that can arise.

1.1 Problems of Concurrency

Dirty reads. The problem of dirty read occurs when one transaction can read data that has been modified but not yet committed by another transaction.

Lost updates. The problem of lost update occurs when one transaction overwrites the changes made by another transaction, because it does not realize that the data has changed. For example, if a transaction T1 reads record R1, following which a second transaction T2 reads record R1, then the first transaction T1 updates record R1 and commits the change, following

[1] DBMS implementations that are based upon the locking protocols of IBM's System R prototype.

which the second transaction T2 updates R1 and also commits the change. In this situation, the update made by the first transaction T1 to record R1 is "lost", because the second transaction T2 never sees it.

Non-repeatable reads. The problem of non-repeatable read occurs when a transaction finds that a record it read before has been changed by another transaction.

Phantom reads. The problem of phantom read occurs when a transaction finds that the same SQL query returns different set of records at different times within the context of the transaction.

1.2 Lock Modes

To resolve the various concurrency problems, Classical Database Systems use locking to restrict concurrent access to records by various transactions. Two types of locks are used:

Shared Locks. Are used to protect reads of records. Shared locks are compatible with other Shared locks but not with Exclusive locks. Thus multiple transactions may acquire Shared Locks on the same record, but a transaction wanting to acquire Exclusive Lock must wait for the Shared locks to be released.

Exclusive Locks. Are used to protect writes to records. Only one transaction may hold an Exclusive lock on a record at any time, and furthermore, Exclusive locks are not compatible with Shared Locks. Thus a record protected by Exclusive Lock prevents both read and write by other transactions.

The various lock modes used by the DBMS, also called Isolation levels, are given below:

Read Committed. In this mode, the SVDB places Commit duration[2] exclusive locks on any data it writes. Shared locks are acquoired on records being read, but these locks are released as soon as the read is over. This mode prevents dirty reads, but allows problems of lost updates, non-repeatable reads, and phantom reads to occur.

Cursor Stability. In addition to locks used for Read Committed, the DBMS retains shared lock on the "current record" until the cursor moves to another record. This mode prevents dirty reads and lost updates.

Repeatable Read. In addition to exclusive locks used for Read Committed, the DBMS places commit duration shared locks on data items that have been read. This mode prevents the problem of non-repeatable reads.

[2]A Commit duration lock, once acquired, is only released when the transaction ends.

Serializable. In addition to locks used for Repeatable Read, when queries are executed with a search parameter, the DBMS uses key-range locks to lock even non-existent data that would satisfy the search criteria. For example, if a SQL query executes a search on cities beginning with the letter 'A', all cities beginning with 'A' are locked in shared mode even though some of them may not physically exist in the database. This prevents other transactions from creating new data items that would satisfy the search criteria of the query until the transaction running the query either commits or aborts.

Clearly, the different lock modes offer different levels of data consistency, trading off performance and concurrency for greater consistency. Read Committed mode offers greatest concurrency but least consistency. The Serialized mode offers the most consistent view of data, but the lowest concurrency due to the long-term locks held by a transaction operating in this mode.

2 Problems with Traditional Lock based concurrency

Regardless of the lock mode used, the problem with a SVDB systems is that Writers always block Readers. This is because all writes are protected by commit duration exclusive locks, which prevent Readers from accessing the data that has been locked. Readers must wait for Writers to commit or abort their transactions.

In all lock modes other than Read Committed, Readers also block Writers.

3 Introduction to Multi-Version Concurrency

The aim of Multi-Version Concurrency is to avoid the problem of Writers blocking Readers and vice-versa, by making use of multiple versions of data.

The problem of Writers blocking Readers can be avoided if Readers can obtain access to a previous version of the data that is locked by Writers for modification.

The problem of Readers blocking Writers can be avoided by ensuring that Readers do not obtain locks on data.

Multi-Version Concurrency allows Readers to operate without acquiring any locks, by taking advantage of the fact that if a Writer has updated a particular record, its prior version can be used by the Reader without waiting for the Writer to Commit or Abort. In a Multi-version Concurrency solution, Readers do not block Writers, and vice versa.

While Multi-version concurrency improves database concurrency, its impact on data consistency is more complex[HB95][AF04].

4 Requirements of Multi-Version Concurrency systems

As its name implies, multi-version concurrency relies upon multiple versions of data to achieve higher levels of concurrency. Typically, a DBMS offering multi-version concurrency (MVDB), needs to provide the following features:

1. The DBMS must be able to retrieve older versions of a row.
2. The DBMS must have a mechanism to determine which version of a row is valid in the context of a transaction. Usually, the DBMS will only consider a version that was committed prior to the start of the transaction that is running the query. In order to determine this, the DBMS must know which transaction created a particular version of a row, and whether this transaction committed prior to the starting of the current transaction.

5 Challenges in implementing a multi-version DBMS

1. If multiple versions are stored in the database, an efficient garbage collection mechanism is required to get rid of old versions when they are no longer needed.
2. The DBMS must provide efficient access methods that avoid looking at redundant versions.
3. The DBMS must avoid expensive lookups when determining the relative commit time of a transaction.

6 Approaches to Multi-Version Concurrency

There are essentially two approaches to multi-version concurrency. The first approach is to store multiple versions of records in the database, and garbage collect records when they are no longer required. This is the approach adopted by PostgreSQL and Firebird/Interbase.

The second approach is to keep only the latest version of data in the database, as in SVDB implementations, but *reconstruct* older versions of data dynamically as required by exploiting information within the Write Ahead Log. This is the approach taken by Oracle and MySQL/InnoDb.

The rest of this paper looks at the PostgreSQL and Oracle implementations of multi-version concurrency in greater detail.

7 Multi-Version Concurrency in PostgreSQL

PostgreSQL is the Open Source incarnation of Postgres. Postgres was developed in University of California, Berkeley, by a team led by Prof. Michael Stonebraker (of INGRES fame). The original Postgres implementation offered a multi-version database with garbage collection. However, it used traditional two-phase locking model that led to the "readers blocking writers" phenomenon.

The original purpose of multiple-versions in the database was to allow time-travel, and also to avoid the need for a Write-Ahead Log. However, in PostgreSQL support for time-travel has been dropped, and the multi-version technology in original Postgres is exploited for implementing a Multi-Version concurrency algorithm. PostgreSQL team also added Row level locking and a Write-Ahead Log to the system.

In PostgreSQL, when a row is updated, a new version (called a tuple) of the row is created and inserted into the table. The previous version is provided a pointer to the new version. The previous version is marked "expired", but remains in the database until it is garbage collected.

In order to support multi-versioning, each tuple has additional data recorded with it:

xmin - The ID of the transaction that inserted/updated the row and created this tuple.

xmax - The transaction that deleted the row, or created a new version of this tuple. Initially this field is null.

To track the status of transactions, a special table called `PG_LOG` is maintained. Since Transaction Ids are implemented using a monotonically increasing counter, the `PG_LOG` table can represent transaction status as a bitmap. This table contains two bits of status information for each transaction; the possible states are in-progress, committed, or aborted.

PostgreSQL does not undo changes to database rows when a transaction aborts - it simply marks the transaction as aborted in `PG_LOG`. A PostgreSQL table therefore may contain data from aborted transactions.

A Vacuum cleaner process is provided to garbage collect expired/aborted versions of a row. The Vacuum Cleaner also deletes index entries associated with tuples that are garbage collected.

Note that in PostgreSQL, indexes do not have versioning information, therefore, all available versions (tuples) of a row are present in the indexes. Only by looking at the tuple is it possible to determine if it is visible to a transaction.

In PostgreSQL, a transaction does not lock data when reading. Each transaction sees a snapshot of the database as it existed at the start of the transaction.

To determine which version (tuple) of a row is visible to the transaction, each transaction is provided with following information:

1. A list of all active/uncommitted transactions at the start of current transaction.

2. The ID of current transaction.

A tuple's visibility is determined as follows (as described by Bruce Momijian in [BM00]):

Visible tuples must have a creation transaction id that:

- is a committed transaction
- is less than the transaction's ID and
- was not in-process at transaction start, ie, ID not in the list of active transactions

Visible tuples must also have an expire transaction id that:

- is blank or aborted or
- is greater than the transaction's ID or
- was in-process at transaction start, ie, ID is in the list of active transactions

In the words of Tom Lane:

A tuple is visible if its xmin is valid and xmax is not. “Valid” means “either committed or the current transaction”.

To avoid consulting the `PG_LOG` table repeatedly, PostgreSQL also maintains some status flags in the tuple that indicate whether the tuple is “known committed” or “known aborted”. These status flags are updated by the first transaction that queries the `PG_LOG` table.

8 Multi-version Concurrency in Oracle

Oracle does not maintain multiple versions of data on permanent storage. Instead, it recreates older versions of data on the fly as and when required.

In Oracle, a transaction ID is not a sequential number; instead, it is a made of a set of numbers that points to the transaction entry (slot) in a Rollback segment header. A Rollback segment is a special kind of database table where “undo” records are stored while a transaction is in progress. Multiple transactions may use the same rollback segment. The header block of the rollback segment is used as a transaction table. Here the status of a transaction is maintained, along with its Commit timestamp (called System Change Number, or SCN, in Oracle).

Rollback segments have the property that new transactions can reuse storage and transaction slots used by older transactions that have committed or aborted. The oldest transaction's slot and undo records are reused when there is no more space in the rollback segment for a new transaction. This automatic reuse facility enables Oracle to manage large numbers of transactions using a finite set of rollback segments. Changes to Rollback segments are logged so that their contents can be recovered in the event of a system crash.

Oracle records the Transaction ID that inserted or modified a row within the data page. Rather than storing a transaction ID with each row in the page, Oracle saves space by maintaining an array of unique transactions IDs separately within the page, and stores only the offset of this array with the row.

Along with each transaction ID, Oracle stores a pointer to the last undo record created by the transaction for the page. The undo records are chained, so that Oracle can follow the chain of undo records for a transaction/page, and by applying these to the page, the effects of the transaction can be completely undone.

Not only are table rows stored in this way, Oracle employs the same techniques when storing index rows.

The System Change Number (SCN) is incremented when a transaction commits.

When an Oracle transaction starts, it makes a note of the current SCN. When reading a table or an index page, Oracle uses the SCN number to determine if the page contains the effects of transactions that should not be visible to the current transaction. Only those committed transactions should be visible whose SCN number is less than the SCN number noted by the current transaction. Also, Transactions that have not yet committed should not be visible. Oracle checks the commit status of a transaction by looking up the associated Rollback segment header, but, to save time, the first time a transaction is looked up, its status is recorded in the page itself to avoid future lookups.

If the page is found to contain the effects of "invisible" transactions, then Oracle recreates an older version of the page by undoing the effects of each such transaction. It scans the undo records associated with each transaction and applies them to the page until the effects of those transactions are removed. The new page created this way is then used to access the tuples within it.

Since Oracle applies this logic to both table and index blocks, it never sees tuples that are invalid.

Since older versions are not stored in the DBMS, there is no need to garbage collect data.

Since indexes are also versioned, when scanning a relation using an index, Oracle does not need to access the row to determine whether it is valid or not.

In Oracle's approach, reads may be converted to writes because of updates to the status of a transaction within the page.

Reconstructing an older version of the page is an expensive operation. However, since Rollback segments are similar to ordinary tables, Oracle is able to use the Buffer Pool to effectively ensure that most of the undo data is always kept in memory. In particular, Rollback segment headers are always in memory and can be accessed directly. As a result, if the Buffer Pool is large enough, Oracle to able create older versions of blocks without incurring much disk IO. Reconstructed versions of a page are also stored in the Buffer Pool.

An issue with Oracle's approach is that if the rollback segments are not large enough, Oracle may end up reusing the space used by completed/aborted transactions too quickly. This can mean that the information required to reconstruct an older version of a block may not be available. Transactions that fail to reconstruct older version of data will fail.

References

[BM00] Bruce Momijian. PostgreSQL Internals through Pictures. Dec 2001.

[TL01] Tom Lane. Transaction Processing in PostgreSQL. Oct 2000.

[MS87] Michael Stonebraker. The Design of the Postgres Storage System. Proceedings 13th International Conference on Very Large Data Bases (brighton, Sept, 1987). Also, Readings in Database Systems, Third Edition, 1998. Morgan Kaufmann Publishers.

[HB95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10.

[AF04] Alan Fekete, Elizabeth J. O'Neil, Patrick E. O'Neil: A Read-Only Transaction Anomaly Under Snapshot Isolation. SIGMOD Record 33(3): 12-14 (2004)

[JG93] Jim Gray and Andreas Reuter. Chapter 7: Isolation Concepts. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993.

[ZZ] Authors unknown. The Postgres Access Methods. Postgres V4.2 distribution.

[DH99] Dan Hotka. Oracle8i GIS (Geeky Internal Stuff): Physical Data Storage Internals. OracleProfessional, September, 1999.

[DH00] Dan Hotka. Oracle8i GIS (Geeky Internal Stuff): Index Internals. OracleProfessional, November, 2000.

[DH01] Dan Hotka. Oracle8i GIS (Geeky Internal Stuff): Rollback Segment Internals. OracleProfessional, May, 2001.

[RB99] Roger Bamford and Kenneth Jacobs, Oracle.US Patent Number 5,870,758: Method and Apparatus for providing Isolation Levels in a Database System. Feb, 1999.