12. Transaction Processing

Most of content here is borrowed from book Elmasri/Navathe

Concurrency Control Techniques

Objectives Concurrency Control Objectives is
to ensure "serializable" and "recoverable" schedules

Concurrency control and Recovery system go hand in hand; here, we plan to have abstract understanding of following techniques and algorithms

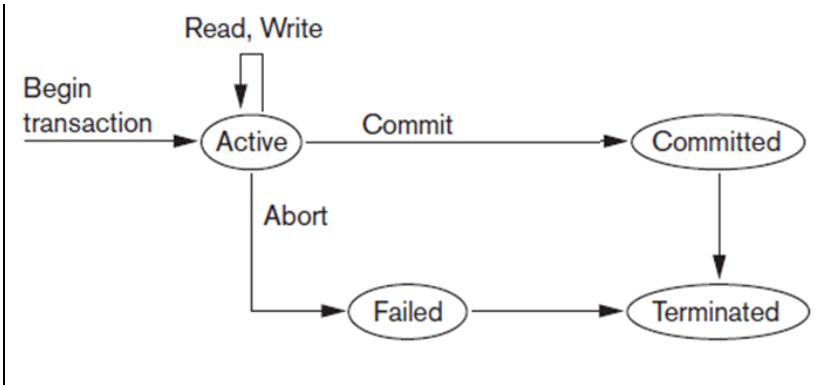
Concurrency Control Techniques (Algorithms)

- 2PL protocol
- Timestamp ordering based Protocols
- Multi-version Concurrency Control
- Optimistic Concurrency Control
- Snapshot Isolation

Recovery (Algorithms)

- Write Ahead Logging Protocol
- ARIES recovery algorithm

Life cycle of a transaction



Recall: Schedule, Serializable Schedule, Recoverable Schedule!

Two Phase Locking protocol (2PL)

Concept of Locking:

You can think of lock as an entry that indicates that a data item is "locked", and access to locked item is restricted or denied to other transactions

Typically we use two types of locks

- Read (or Shared) Locks, and
- Write (or Exclusive) Lock

Read Lock: a transaction acquires Read lock (shared lock) before performing read; many transactions can have read or shared lock onto a data item concurrently

Write Lock: Any transaction requiring modifying a data item requires exclusive lock, so that no other transaction can access this item for reading and writing while transaction updates it.

Locking Rules:

- A transaction must acquire a *read_lock* or *write_lock* before it can read or write an item
- A transaction must acquire a *write_lock* before it can write an item
- Must unlock after all read/write are complete
- Transaction having lock on an item can only unlock the item.

Lock Compatibility Matrix		
	Shared	e**X**lusive
If No Lock	YES	YES
If shared	YES	NO
If exclusive	NO	NO

It should be easy to appreciate that "how long a lock is to be retained", merely locking for the duration of operation does not yield the required outcome; early releasing of lock still may lead having non-serializable schedule.

Two phase locking protocol basically puts restrictions on locking and unlocking operations, that is once a transaction starts unlocking, it is no more performing any operation on unlocked items, and no more acquiring any additional lock.

Two phases here are - "Locking Phase" also referred as *growing phase*; and "Unlocking Phase" or *shrinking phase*. Once a transaction enters into shrinking phase; it can no more acquire any additional lock.

It essentially means, once a transaction has released an item, it no more can perform any operation on that data item, that is, it can no more have operation that may lead to a cycle in precedence graph.

Rigorous 2PL

A most commonly used variation of 2PL protocol is Rigorous 2PL – this does not allow a transaction to release any of lock (read/write) until it commits or aborts.

Consider a situation in which a transaction releases write lock before it commits. If it aborts, other transaction will also have to aborted and may have cascaded aborts.

How does 2PL guarantee serializability?

Scenario-1

T1: r1(X); w1(X);

T2: r2(X); w2(X);

- T1 requests exclusive lock on X; gets granted; T1 reads(X)
- T2 requests shared lock on X; cannot get; asked to wait
- T1 attempts to write X; allowed to write.
- T1 releases lock of X
- T2 gets locks and executes its conflicting operations

This leads T2 executing operations are T1 has done all conflicting operations.
Equivalent to "T2; T1"

Had T1 released lock early, as shown following, there would be a cycle? (2PL rescues)

- T1 requests shared lock on X; gets granted; T1 reads(X)
- T1 releases lock on X
- T2 requests shared lock on X; gets; T2 reads X
- T2 releases lock on X
- T1 requests exclusive lock on X; gets granted
- T1 writes X
- Has resulted a CYCLE?

Scenario-2

T1: r1(X); w1(X);

T2: r2(X); w2(X);

- T1 requests shared lock on X; gets granted; T1 reads(X)
- T2 requests shared lock on X; gets granted; reads X
- T1 to perform write X, it requires exclusive lock, it request but cannot be granted as T2 also has shared lock [note shared lock by self can be promoted to exclusive provided no other transaction is having shared lock on the data item]
- T1 has to wait till T2 releases read lock
- T1 gets exclusive lock when T2 releases it
- T1 writes. No Cycle formed! Equivalent to "T2; T1"

Scenario-3

T1: r1(X); w1(Y);

T2: r2(Y); w2(X);

- T1 seeks shared lock on X; gets granted; T1 reads X
- T2 seeks shared lock on Y; gets granted; T2 reads Y
- T1 request exclusive request on Y; cannot get; waits for T2 finish
- T2 requests shared lock on X; cannot get; waits for T1 to finish

This is deadlock!

2PL also reduces concurrency significantly!

Implementation of Locking

- Locking and Unlocking requests are handled by Lock Manager
- Locking can be granted or requesting transaction put in waiting based on locking rules.
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests.
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked
- Lock Manager should also handle the deadlocks (based on some strategies), selects transaction to abort, (and take care of starvation).

Deadlock prevention Techniques

Context: suppose that transaction Ti tries to lock an item X but is not able to do so because X is locked by some other transaction Tj with a conflicting lock.

- Cautious Waits: If Tj is not blocked (not waiting for some other locked item), then Ti is blocked and allowed to wait; otherwise abort Ti.
  Cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction.
- Deadlock Detection and Victim Selection:
  - Wait-For graph is popular technique for deadlock detection. It is directed graph, were we have node for every active transaction, and an edge from node Ti to Tj indicates that Ti is waiting for Tj to finish and release lock.
  - A deadlock is detected when there is cycle in the graph.
  - To break the deadlock, one of the transactions is to be aborted – a victim selection.
  - Normally a victim is the one that is at early stage in execution.

Concurrency control based on Time-Stamp Ordering

Deadlock is the main problem with 2PL. Concurrency control with timestamp ordering doesn't use locks, therefore deadlocks cannot occur.

What is Timestamps?

- Unique identifiers generated by DBMS to identify transactions
- Also indicate start order of transactions
- Let Timestamps of transaction T, is denoted by TS(T)
- Time stamps can be generated in several ways: Starting "timestamp" of a transaction; Transaction counter, increased every time new transaction is started, and so.

Timestamp Ordering – the intuition

- Operations from multiple transactions are included in schedule in their timestamp order: this leads to making schedule equivalent a "serial schedule" as "$T_{older};T_{younger}$".
- Intuition of Time stamp based concurrency control is that conflicting operations in a schedule occur in the order of their transaction's timestamp. The time ordering algorithm ensures this "serialization rules".
- Based on this principle, and request from a participating transaction; decision for transaction is taken - if its operation is immediately included in schedule, or should wait, or no chance of inclusion, therefore it should abort, etc.

Timestamp Ordering - Algorithm

The intuition -

- Older transaction cannot write a data item if any younger transaction has read/written it.
- Older transaction cannot read a data item if any younger transaction has written it.
- If this is violated, then the transaction is aborted, and resubmitted with new timestamp.

Let us have two timestamps values for data items

- **RTS(X)**: largest timestamp of data item X, that has successfully **read** item X. It is timestamp of youngest transaction that has read item X.
- **WTX(X):** largest timestamp of data item X, that has successfully **written** X - timestamp of youngest transaction that has written X.

Basic Algorithm-

If transaction T issues a **write(X)** operation

//then to let it write, no younger transaction should have read or written X

If **RTS(X) > TS(T) or WTS(X) > TS(T)**

abort T

// This is required because of younger transaction has already read or updated X before T,
// therefore if we allow T to update X, will violate TO

Otherwsie

execute WRITE(X)

WTS(X) = TS(T)

If transaction T issues a **read(X)** operation

//then to let it read, no younger transaction should have written X

If WTS(X) > TS(T)

abort T

// This is because of younger transaction has already updated X before T could read it,
// therefore if we allow T to read X, will violate TO

Otherwsie

execute READ(X)

RTS(X) = max(TS(T), RTS(X))

Multi-version Concurrency Control Schemes

- Basic Time Stamp ordering algorithm may leaded to many aborts.
- Multi-version TO schemes help in increasing concurrency.
- Multi-version schemes keep old versions of data items, and instead of asking a requester to wait/abort, "old copies" of data item is handed over.
- When read request comes, an "appropriate version" is returned immediately - typically version created by youngest among older (including self)
- Each successful write results in the creation of a new version of the data item written.
- Both type of MVCC algorithms are available
    - Timestamp ordering – MV Timestamp Ordering Schemes
    - Locking – Multi-version 2PL protocol

Multi-version based on Timestamp Ordering

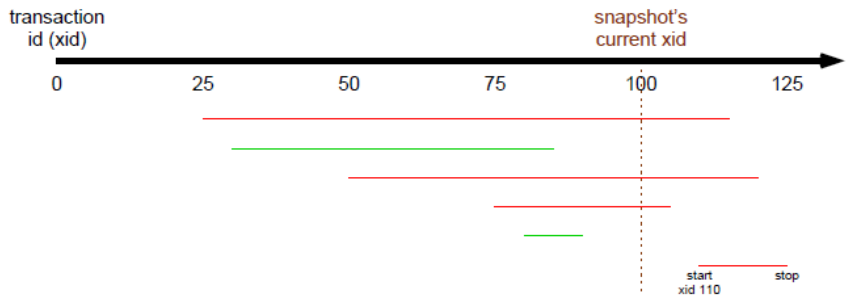
- Let us say there are versions X1, X2, … Xm for a data item X. Let suffix i of version Xi be TS of transaction Ti that created Xi. Let us also have RTS(Xi) and WTS(Xi) defined as following -
  - RTS(Xi): is timestamps of youngest transactions that have successfully read version Xi. When transaction T is allowed to read the value of version Xi, the value of RTS(Xi) is set to **`max(RTS(Xi), TS(T))`**
  - WTS(Xi): timestamp of the transaction that created Xi

Multi-version TS Ordering – Algo intuition

- There are multiple versions of a data item; say as shown in figure here; there are three versions of account 101. Let each version be identified by TS of transaction that created the version.

ver	wts	rts	AcNo	Balance
x5	5	6	101	20000
x8	8	11	101	22000
x16	16	16	101	35000

- Each version maintains WTS, and RTS (timestamp of transaction that created the version, and timestamp of highest (youngest) transactions that read the version)
- A transaction sees versions that are created by older transactions only.
- A transaction reads a version created by youngest among older transactions.
- If allowed to write, every write creates a new version. A transaction T is not allowed to write (and create a new version) if a younger (than T) transaction has read the version created by nearest older (backward) - ensuring this is necessary for basic time stamp ordering principle.



Green is visible. Red is invisible.

Only transactions completed before transaction id 100 started are visible.

For example a version x5 created by transaction T5 has been read by T7 and T9. Now if T8 attempts to write X; it cannot be allowed to create new version as younger transaction has already read X! Allowing this will violate TO principle T8 writes after T11 has read (Global serializability to be formed). Suppose T11 requests reading X again it will read X9 (if T9 was allowed to write)

Multi-version TS Ordering – Algorithm

When a transaction T issues **write(X)** request

Find version Xi, that is created by youngest among older transactions (i.e. the one having highest WTS(X*i*) among those with WTS(X*i*) <= TS(T)

If any younger (than T) transaction has read Xi, i.e. RTS(X*i*) > TS(T) then

abort T

Else,

create new version X*j* and set

WTS(X*j*) = RTS(X*j*) = TS(T)

When T issues **read(X)** – always succeeds

Find version Xi, that is created by youngest among older transactions (i.e. the one having highest WTS(X*i*) among those with WTS(X*i*) <= TS(T) , and

return value of X*i* to T, and

set RTS(X*i*) ← max(TS(T), RTS(X*i*))

Worked Example:

- Consider three versions of an account tuple.
- Say T9 attempt to read the tuple; which version should we return?
- To ensure TS ordering, it should be older; therefore version created by youngest among older is returned, i.e. x8. RTS of x8 remains unchanged as it is already 11.

ver	wts	rts	AcNo	Balance
x5	5	6	101	20000
x8	8	11	101	22000
x16	16	16	101	35000

- Now suppose, T9 attempt to write the tuple with salary=25000; however this cannot allowed. Nearest older version x8, has been read by younger (than T9) transaction T11. T9 needs to be aborted.
- Now, suppose T12 attempts reading tuple, version x8 is again returned and RTS is set to 12
- If T13 attempts writing tuple with salary=45000, then version x13 is created with WTS=RTS=13

ver	wts	rts	AcNo	Balance
x5	5	6	101	20000
x8	8	~~11~~ 12	101	22000
x16	16	~~16~~ 18	101	35000
x13	13	~~13~~ 15	101	45000

- If T15 attempts to read the tuple, x13 is returned, and set RTS(x13)=15
- If T18 attempts to read the tuple, x16 is returned, and set RTS(x16)=18

MVCC: Conclude

- Increases concurrency; reads are never blocked
- MVCC is one of the popular techniques used in practice
- Additional storage for multiple versions seems to be major drawback of this approach; however versions that are no more relevant, can be garbage collected. For instance if T13 is oldest active transaction, then versions before x13 will never be needed and can be removed.

Validation based optimistic techniques

- This is also called as optimistic approach; where we assume that majority of transactions are read only; or chances of multiple transactions updating same data items are low
- If this assumption is correct, then this validation based protocols are quiet efficient.
- In this approach, we allow transaction to execute in interleaved fashion without any check. We check serialization rules at the time of commit: see if commit would violate any TO principle, then we abort the transaction, otherwise allow committing.

In this category of techniques, execution of transaction Ti is done in three phases.

1. Read and execution phase: Transaction Ti writes only to temporary local variables
2. Validation phase: Transaction Ti performs a "validation test'' to determine if data from local variables can be written to database without violating TO principle.
3. Write phase (update phase): If Ti is validated, the updates are applied to the database; otherwise, Ti is rolled back.

Validation process:

If a transaction Ti is in validation phase, then conflicting operations needs to be checked with every transaction Tj that is either active at the time of starting of Ti, or started later, and Tj has either committed or in validation phase then one of following is true then Ti can be validated-

(1) Ti starts its write phase after Tj completes its write phase, and read-set of Ti has no common items with write set of Tj
(2) Both read-set and write-set of Ti have no common item with write-set of Tj, and Tj has completed its read/execution phase before Ti.

Note that evaluation of condition (2) is done only when condition (1) is false – it may be noted that evaluation of (1) is simpler. If none of the condition is true then transaction is invalidated and aborted!

Snapshot Isolation [1]

Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows - results poor performance

Snapshot Isolation is primarily a optimistic variation of Multi Version Concurrency Control (MVCC), and proposed by Berenson et al, SIGMOD 1995. Many DBMS like Oracle, PostgreSQL, SQL Server use its variations.

T1	T2	T3
W(Y, 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X,2) W(Z,3) Commit
	R(Z) → 0 R(Y) → 1 W(X, 3) **Commit**	

The technique in nutshell through a simple example

- A transaction T2 executing with Snapshot Isolation takes snapshot of committed data at start;
- A transaction always reads/modifies data in its own snapshot
- updates of concurrent transactions are not visible to T2
- writes of T2 complete when it commits
- First-committer-wins rule:
  Commits only if, no other concurrent transaction has already written data that T2 intends to write.
- In the case here T2 cannot be committed, therefore aborts.

In snapshot isolation, reading is *never* blocked, and also doesn't block other transaction activities, even updates.

Problem with Snapshot Isolation

Has a noted problem called "skew write" – explained through example below

T1: Read Y; X=Y; Write(X)

T2: Read X; Y=X; Write(Y)

Initially x = 3 and y = 17

Serial execution should: x = ?, y = ?

If both transactions start at the same time, with snapshot isolation: x = ? , y = ?

More recent variation of snapshot isolation is Serializable Snapshot Isolation (SSI) is implemented in newer versions of RDBMS, and you may not experience skew write.

[1] Source: Lecture slides of Silberschatz-Korth-Sudarshan, 6th ed