Concurrency Control Algorithms

- Two-phase locking
- Optimistic concurrency control
- Timestamp ordering

Locking

- Use exclusive locks on a resource
  - A transaction obtains a read/write lock for an object before performing a read/write operation
  - A lock for an object is granted to a transaction if no conflicting locks are held by other transactions
  - A transaction releases a lock on an object when the operation on the object has been completed
- Need a lock manager – a server that issues locks on resources
  - A client can request a lock and then release a lock
- To ensure that conflicting operations of two transactions are executed in the same order, a transaction is not allowed to get any new locks after it has released a lock
Two-Phase Locking (2PL)

- A transaction executes in two phases
  - Phase 1 (Growing phase): The transaction acquires the locks it needs
  - Phase 2 (Shrinking phase): The transaction releases its locks

In 2PL, a transaction executes in two phases
\(T_a\)
\[
\begin{align*}
x &= 0 \\
x &= x + 1
\end{align*}
\]

\(T_b\)
\[
\begin{align*}
x &= 0 \\
x &= x + 2
\end{align*}
\]

\(x = 1\) or 2 in a legal schedule

Acquire x.lock
w(x)0
Release x.lock

Acquire x.lock
w(x)0
Release x.lock

The transaction gets a new lock after it releases a lock

Acquire x.lock
r(x)0
Release x.lock

Acquire x.lock
r(x)0
Release x.lock

Acquire x.lock
w(x)0
Release x.lock

Acquire x.lock
w(x)0
r(x)0
w(x)1
Release x.lock

Release x.lock

Acquire x.lock
r(x)1
Release x.lock

Acquire x.lock
w(x)1
Release x.lock

Release x.lock

Acquire x.lock
w(x)3
Release x.lock

Acquire x.lock
w(x)3
Release x.lock

This schedule is illegal (x = 3).

2PL produces a legal schedule (x = 2)
Strict Two-Phase Locking

- 2PL can lead to cascading aborts
  - Suppose T2 reads the result of a write of T1, if T1 is aborted then T2 must be aborted
- To avoid cascading aborts, we require that locks can only be released when the transaction commits or aborts. This is known as strict two-phase locking
Problems with Locking

• Locks have an overhead: a lock manager is needed to keep track of locks
• Locks can result in deadlock: need to detect or avoid deadlock
  – A solution: Adding timeouts to locks. When a lock times out, the transaction holding the lock is aborted
• Locks may reduce concurrency by having transactions hold the locks until the transaction commits or aborts (strict two-phase locking)

Optimistic Concurrency Control

• In most applications the chance of two transactions accessing the same object is low
• Allow transactions to proceed without obtaining locks
• Check for conflicts at commit time: if there is a conflict, abort and restart some transaction
Three Phases

- **Working phase**: A transaction operates in a private workspace, so its modifications are not visible to other transactions until it commits.
- **Validation phase**: When transaction T is ready to commit, check if any object read by T has been modified by transactions that completed after T started.
  - Yes: abort T (the client needs to restart T)
  - No: commit T
- **Update phase**: if a transaction is validated, its private workspace is copied to stable storage (i.e., all changes are made permanent).

Optimistic Concurrency Control: Pros and Cons

- **Advantages**
  - Deadlock free: no locks are used
  - Maximum concurrency
- **Disadvantages**
  - Restart transaction if aborts
  - Starvation can occur
  - Probability of conflict increases substantially at high loads
Timestamp Ordering

- **Basic idea**: if two transactions perform a pair of conflicting operations, the transaction that started earlier should perform its operation first
  - This produces serially equivalent schedules
- Each transaction T is assigned a timestamp ts(T) when it starts
- Two timestamps are maintained for each object x
  - **Read timestamp**, denoted by rts(x), is the maximum timestamp of a transaction that read the object
  - **Write timestamp**, denoted by wts(x), is the maximum timestamp of a transaction that modified the object

Timestamp Ordering Rule

- A transaction’s request to read an object is valid only if that object was last written by an earlier transaction

When transaction T performs read(x):
- if ts(T) > wts(x), perform read(x) and rts(x) \(\leftarrow\) MAX(rts(x), ts(T))
- else abort T

- A transaction’s request to write an object is valid only if that object was last read and written by an earlier transaction

When transaction T performs write(x)
- if ts(T) > rts(x) and ts(T) > wts(x), perform write(x) and wts(x) \(\leftarrow\) ts(T)
- else abort T