Scheduling and Concurrency Control

● Objectives

- atomic execution of transactions on shared data by controlling the interleaving of concurrent accesses

● Conflicts

- a request to access a data object meets other request from another transaction
- one of the requests is a write access request
- RW conflict, WR conflict, WW conflict

● Algorithms

- two-phase locking
- timestamp ordering
- certifier schemes
- integrated schemes
- hybrid schemes
Scheduling Approaches

Transaction Manager ↔ Scheduler ↔ Data Manager

- Options for a scheduler
  1) immediately schedule it
  2) delay it (insert it into a queue)
  3) reject it (causing abort)

- Aggressive vs conservative approaches
  - optimistic vs pessimistic
  - aggressive favors immediate action (option 1);
    if impossible to finish T, abort some (option 3)
  - conservative favors option 2
  - performance trade-offs between the two

- Syntactic vs semantic correctness
Two-Phase Locking (2PL)

Assumption: each data object has a lock associated with it.

- Two locking modes
  - shared (read) lock
  - exclusive (write) lock

- Well-formed transaction
  - locks data object before accessing it
  - does not lock the same data object twice
  - unlocks all the locked objects before completion

- Some notations
  rl(x): read lock on x
  ru(x): unlock (release) x
  wl(x): write lock on x
  wu(x): unlock (release) x
Basic 2PL

1. For a request $p_i(x)$, check if $pl_i(x)$ conflicts with other $ql_j(x)$ that already exists.
   - if so, delay $p_i(x)$, forcing $T_i$ to wait
   - if not, set $pl_i(x)$ and send $p_i(x)$ to data manager

2. Once $pl_i(x)$ is set, it is not released until after data manager acknowledges that $p_i(x)$ is processed

3. Two-phaseness
   - growing phase and shrinking phase cannot be mixed
   - once a transaction $T_i$ starts releasing a lock, it cannot set another lock on any data object
   - to guarantee all pairs of conflicting operations of two transactions are scheduled in the same order (to guarantee consistency)
Example of Simple Locking and 2PL

\[ T_1: \begin{align*} &A + 100 \rightarrow A \\
&\quad B + 100 \rightarrow B \end{align*} \quad T_2: \begin{align*} &A \times 2 \rightarrow A \\
&\quad B \times 2 \rightarrow B \end{align*} \]

correctness assertion: \( A = B \)

- Well-formed, not two-phased version of \( T_1 \): \( T_1' \)
  
  lock A
  
  A + 100 \rightarrow A
  
  unlock A
  
  lock B
  
  B + 100 \rightarrow B
  
  unlock B

- Well-formed two-phased version of \( T_1 \) and \( T_2 \)

\[ T_1: \begin{align*} &\text{lock A} \\
&A + 100 \rightarrow A \\
&\text{lock B} \\
&\text{unlock A} \\
&B + 100 \rightarrow B \\
&\text{unlock B} \end{align*} \quad T_2: \begin{align*} &\text{lock A} \\
&A \times 2 \rightarrow A \\
&\text{lock B} \\
&A \times 2 \rightarrow A \\
&\text{unlock A} \\
&B \times 2 \rightarrow B \\
&\text{unlock B} \]
Inconsistent Execution

\[ T_1': \text{ lock } A \]
\[ T_1': \text{ A + 100 } \rightarrow A \]
\[ T_1': \text{ unlock } A \checkmark \]
\[ T_2: \text{ lock } A \checkmark \]
\[ T_2: \text{ lock } B \]
\[ T_2: \text{ A } \times 2 \rightarrow A \]
\[ T_2: \text{ B } \times 2 \rightarrow B \]
\[ T_2: \text{ unlock } A \]
\[ T_2: \text{ unlock } B \checkmark \]
\[ T_1': \text{ lock } B \checkmark \]
\[ T_1': \text{ B + 100 } \rightarrow B \]
\[ T_1': \text{ unlock } B \]

A: \( T_1' \rightarrow T_2 \)

B: \( T_2 \rightarrow T_1' \)
Consistent Execution

\[ T_1: \text{lock } A \]
\[ T_1: A + 100 \rightarrow A \]
\[ T_1: \text{lock } B \]
\[ T_1: \text{unlock } A \]
\[ T_2: \text{lock } A \]
\[ (T_2 \text{ waits on lock } B) \]
\[ T_1: B + 100 \rightarrow B \]
\[ T_1: \text{unlock } B \]
\[ T_2: \text{lock } B \]
\[ T_2: A \times 2 \rightarrow A \]
\[ T_2: B \times 2 \rightarrow B \]
\[ T_2: \text{unlock } A \]
\[ T_2: \text{unlock } B \]

- **Locked point**
  - the point at the end of the growing phase at which the transaction owns all the locks

- **Equivalence**
  - an execution \( L \) is equivalent to a serial execution \( L' \) in which every transaction executes at its locked point
Correctness of Schedulers

- Need to prove
  - all schedules representing executions that could be produced by the scheduler are serializable (SR)

- How to prove it?
  - enumerate all the possible schedules and check SR is infeasible
  - two step approach
    - characterize properties of its schedules
    - prove that any schedule with such properties are serializable

- How to characterize the properties?
  - from the specification of scheduling algorithms
Properties of Schedules by 2PL

1. If $o_i(x)$ is in the schedule,
   then $ol_i(x)$ and $ou_i(x)$ are also in the schedule
   and $ol_i(x) < o_i(x) < ou_i(x)$

2. If $p_i(x)$ and $q_j(x)$ ($i \neq j$) are conflicting operations in the schedule,
   then either $pu_i(x) < ql_j(x)$
     or $qu_j(x) < pl_i(x)$

3. If $p_i(x)$ and $q_i(y)$ are in the schedule,
   then $pl_i(x) < qu_i(y)$

   --- from two-phaseness
Correctness of 2PL

**Theorem:** 2PL is correct (i.e., SR)

Proof:

1. If \( T_i \rightarrow T_j \) in the schedule, then \( p_{u_i}(x) < q_{l_j}(x) \) for some \( x \).

2. If \( T_i \rightarrow T_j \rightarrow T_k \) in the schedule, then \( T_i \) releases some lock before \( T_j \) set the lock, and the same for \( T_j \) and \( T_k \). By induction, same for \( T_1 \) and \( T_n \) if \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \)

3. If the schedule has a cycle in the serialization graph \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \) then \( T_1 \) releases some lock before \( T_1 \) sets a lock

    --- violation of two-phasedness, cannot be a 2PL schedule

Hence a cycle cannot exist.
Deadlocks

- Unfortunate property of locking
  
  $T_1: r_1(X) \rightarrow w_1(Y) \rightarrow c_1$
  
  $T_2: w_2(Y) \rightarrow w_2(X) \rightarrow c_2$

  schedule: $r_{l1}(X) w_{l2}(Y)$ delay $w_{l2}(X)$ delay $w_{l1}(Y)$

- Four necessary conditions for deadlock
  
  - mutual exclusion: one request is in exclusive mode
  
  - wait-for condition: holding a resource while waiting
  
  - no preemption
  
  - circular wait

- Approaches
  
  - prevention
  
  - avoidance
  
  - detection and resolution
Issues in Deadlock Detection and Resolution

- Time-out
  - no detection (by guessing)
  - chances of aborting transactions not involved in deadlock

- Wait-for graph (WFG) maintenance
  - precise detection
  - large overhead
  - how often should we check for a cycle in WFG?

- Victim selection
  - select the one with minimum cost
  - avoid cyclic restart
Deadlock Prevention

- Priority-based scheme
  Allow $T_i$ to be blocked (wait for) $T_j$, if $T_i$ has higher priority than $T_j$.
  Otherwise, $T_i$ is aborted.
  - deadlock is impossible: $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_1$
    implies priority($T_1$) > priority($T_2$) > ... > priority($T_1$)

- Potential problem of livelock (cyclic restart)
  - if a transaction uses higher priority when restarted
  - livelock is different from deadlock in that it does not prevent a transaction from execution, but it prevents the transaction from completing because of continuous abort/restart

- Avoiding livelock
  - by ensuring that a transaction will eventually have a priority high enough to complete
Wait-Die and Wound-Wait

- Timestamp
  - monotonically increasing number
  - unique
  - finite number of smaller timestamps
  - priority of a transaction is the inverse of its timestamp:
    older transaction → higher priority

- Scenario: $T_i$ requests a lock on which $T_j$ has a conflicting lock
  - Wait-die: if $ts(T_i) < ts(T_j)$ then $T_i$ waits else abort $T_1$
  - Wound-wait: if $ts(T_i) < ts(T_j)$ then abort $T_j$ else $T_1$ waits

  - terms wound, wait, and die are used from $T_i$’s viewpoint
  - in both schemes, younger transaction is aborted
  - wait-die favors younger transaction, while wound-wait favors older transactions
Variations of 2PL

- **Conservative 2PL**
  - deadlock prevention using pre-declaration
  - obtain all locks before submitting ops
  - never aborts a transaction

- **Strict 2PL**
  - release all locks together when T terminates
  - almost all 2PL implementations use it - why?
  - practical reason: when scheduler can release lock?
  - additional benefit: strictness
  - actually, readlocks can be released earlier - when?
Other Issues in Locking

- Implementation issues
  - optimization for frequent operations: lock/unlock
  - atomicity of read and write operations
    - problem associated with disk block

- Phantom problem
  - what is it?
  - index locking
  - predicate locking - why not used in general?

- Multi-granularity locking
  - lock type graph and lock instance graph
  - implicit/explicit locking
  - intention lock and its compatibility
  - issue: determining the level of locking granularity
Performance and Tree-Locking

● Thrashing
  - resource contention
  - data contention

● Policy
  - blocking policy vs restart policy
  - low resource contention and severe data contention
    -> restart is a better policy (surprise?)
  - blocking is selfish; restart is self-sacrificing

● Impact of granularity on performance

● Impact of number of locks per transaction
  - reduced throughput and increase deadlocks - when?

● Tree locking - why important?