**Review: Data Mining**

- **Mining process**
- **Unsupervised Algorithms**
  - Clustering
  - Association Rules
- **Supervised Algorithms**
  - Decision Trees
  - Regression Trees
  - Bayesian Networks

**Review: Concurrency Control**

- **We want DBMSs to have ACID properties**
- **These properties supported by:**
  - Transactions: unit of atomicity
  - Log: information to undo/redo transactions
  - Scheduler: limit reads/writes of Xactions to:
  - reduce anomalies
  - enhance concurrency
- **Scheduling**
  - A serial execution of transactions is safe but slow
  - Try to find schedules equivalent to serial execution
  - One solution for serializable schedules is 2PL

**Review: Anomalies**

- **Reading Uncommitted Data ("WR", dirty reads):**
  
  | T1: R(A), W(A), R(B), W(B), Abort |
  | T2: R(A), W(A), C |

- **Unrepeatable Reads ("RW" Conflicts):**

  | T1: R(A), R(A), W(A), C |
  | T2: R(A), W(A), C |

- **Overwriting Uncommitted Data ("WW", lost update):**

  | T1: W(A), W(B), C |
  | T2: W(A), W(B), C |

**Review: Anomalies (cont)**

- If DBMS changes during transaction, result may not reflect consistent DBMS state

  - **E.g., Consider T1 - “Find oldest sailor for each rating”**
    - T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (say, age = 71).
    - Next, T2 inserts a new sailor; rating = 1, age = 96.
    - T2 also deletes oldest sailor with rating = 2 (and, say, age = 80), and commits.
    - T1 now locks all pages containing sailor records with rating = 2, and finds oldest (say, age = 63).
Review: Anomalies (cont.)

• Some anomalies might be acceptable sometimes
• SQL 92 supports different “Isolation Levels” for a transaction (Lost Update not allowed at any level)

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable Reads</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Review: Precedence Graphs

• Anomalies can be related to “conflicts”
  - 2 Xacts accessing same object, at least one write
• Precedence graphs show conflicts
• Cycle in precedence graph indicates anomaly

T1: R(A), R(A), W(A), C
T2: R(A), W(A), C

Review: Schedule Characteristics

• Want schedule to optimize concurrency vs anomaly
• Many criteria to evaluate schedules

Review: Locking Issues

• When a transaction needs a lock, it either...
  - blocks until the lock is available
  - or aborts, starts again later
• Locking has significant overhead
• Locking approaches are subject to Deadlock
  - must either prevent or detect deadlock
• Locking also subject to Convoys
  - With pre-emptive multitasking, transaction with lock may be pre-empted many times to allow blocked transactions to execute, but they get no work done
  - Chain of block Xactions called a Convoy

Review: Precedence Graphs

• Anomalies can be related to “conflicts”
  - 2 Xacts accessing same object, at least one write
• Precedence graphs show conflicts
• Cycle in precedence graph indicates anomaly

T1: R(A), R(A), W(A), C
T2: R(A), W(A), C

Review: Schedule Characteristics

• Want schedule to optimize concurrency vs anomaly
• Many criteria to evaluate schedules

Review: Locking Issues

• When a transaction needs a lock, it either...
  - blocks until the lock is available
  - or aborts, starts again later
• Locking has significant overhead
• Locking approaches are subject to Deadlock
  - must either prevent or detect deadlock
• Locking also subject to Convoys
  - With pre-emptive multitasking, transaction with lock may be pre-empted many times to allow blocked transactions to execute, but they get no work done
  - Chain of block Xactions called a Convoy

Review: Locking in indexes

• What should we lock?
  - Allow locking of DBMS, Tables, Pages, Tuples
  - To get lock at any level, use intention locks at higher level
• Locking in indexes
  - don’t want to lock a B-tree root for a whole transaction!
  - actually do non-2PL “latches” in B-trees
Multi-Granularity Example

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock, must hold IS or IX on parent.
- To get X or IX or SIX, must hold IX or SIX on parent.
- Must release locks in bottom-up order.

Tuple 1 wants to read all tuples, change a few
Tuple 2 wants to change Tuple 4
- T1 gets IX lock on DBMS
- T1 gets SIX lock on Sailor, Pages
- T1 gets X lock on each approp. Tuple
- T2 gets IX lock on DBMS, tries to get IX lock on Sailor, but this conflicts with T1’s SIX lock, so T2 blocks.

Today

- CC w/out locking
  - “optimistic” concurrency control
  - “timestamp” and multi-version concurrency control
  - locking usually better, though

Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts commit.

Kung-Robinson Model

- Xacts have three phases:
  - READ: Xacts read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.

Optimistic Idea

- Each transaction gets a timestamp
- Optimistically hope that all conflicts follow timestamp order
  - i.e. as if Xactions ran serially, in timestamp order
- If out-of-order conflict occured, abort Xaction

Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
  - Each Xact is assigned a numeric id.
    - Just use a timestamp.
  - Xact ids assigned at end of READ phase, just before validation begins. (Why then?)
- ReadSet(Ti): Set of objects read by Xact Ti.
- WriteSet(Ti): Set of objects modified by Ti.
Test 1

• For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

Test 2

• For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase
  - WriteSet(Ti) ∩ ReadSet(Tj) is empty.

Test 3

• For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does
  - WriteSet(Ti) ⊆ WriteSet(Tj) is empty
  - WriteSet(Ti) ⊆ ReadSet(Tj) is empty.

Validation Notes

• Only permit one Xact to validate at a time
  - often do validation inside critical section
  - otherwise might miss conflicts

• Often do write phase inside critical section also
  - if writes happen serially, don’t have to check rule 3

Applying Tests 1 & 2: Serial Validation

• To validate Xact T, compare to all earlier overlapping Ts
  
  valid = true;
  // S = set of Xacts that committed after Begin(T)
  foreach Ts in S do |
    if ReadSet(T) does not intersect WriteSet(Ts)
      then valid = false;
  }
  if valid then { install updates; // Write phase
    Commit T }
  else Restart T 

Comments on Serial Validation

• Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.

• Assignment of Xact id, validation, and the Write phase are inside a critical section
  - i.e., Nothing else goes on concurrently.
  - If Write phase is long, major drawback.

• Optimization for Read-only Xacts:
  - Don’t need critical section (because there is no Write phase).
Overheads in Optimistic CC

- **Must record read/write activity in ReadSet and WriteSet per Xact.**
  - Must create and destroy these sets as needed.

- **Must check for conflicts during validation, and must make validated writes "global."**
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.

- **Optimistic CC restarts Xacts that fail validation.**
  - Work done so far is wasted; requires cleanup.

``Optimistic” 2PL

- If desired, we can do the following:
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of Xact, make writes global, then release all locks.

- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in Xacts being blocked, waiting for locks.
  - However, no validation phase, no restarts (modulo deadlocks).

Timestamp CC

- **Idea:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.

When Xact T wants to read Object O

- **If** TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again! Contrast use of timestamps in 2PL for ddlk prevention.)

- **If** TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))

  Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

When Xact T wants to Write Object O

- **If** TS(T) < RTS(O), this violates timestamp order of T w.r.t. writer of O; abort and restart T.

- **If** TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - **Thomas Write Rule:** We can safely ignore such outdated writes; need not restart T! (T’s write is effectively followed by another write, with no intervening reads.)
  - Allows some serializable but non-conflict serializable schedules:
    - Else, allow T to write O.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A) Commit</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>W(A) Commit</td>
<td></td>
</tr>
</tbody>
</table>

Timestamp CC and Recoverability

- Unfortunately, unrecoverable schedules are allowed:

- **Timestamp CC can be modified to allow only recoverable schedules:**
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A) W(B) Commit</td>
<td></td>
</tr>
</tbody>
</table>
**Multiversion Timestamp CC**

- Idea: Let writers make a “new” copy while readers use an appropriate “old” copy:
  - Readers are always allowed to proceed.
    - But may be blocked until writer commits.

- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are “too old to be of interest”.
- Each Xact is classified as Reader or Writer.
  - Writer may write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.

**Reader Xact**

- For each object to be read:
  - Finds newest version with WTS < TS(T).
    (Starts with current version in the main segment and chains backward through earlier versions.)

- Assuming that some version of every object exists from the beginning of time, Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.

**Writer Xact**

- To read an object, follows reader protocol.
- To write an object:
  - Finds newest version V s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Xacts can see TS values but can’t read version CV.)
  - Else, reject write.

**Summary**

- Optimistic CC minimizes CC overheads in an environment where reads common, writes rare.
  - Optimistic CC has its own overheads however;
  - most real systems use locking.

- Timestamp CC is another alternative to 2PL:
  - allows some serializable schedules that 2PL does not (although converse is also true).
  - Ensuring recoverability with Timestamp CC requires ability to block Xacts, which is similar to locking.

- Multiversion Timestamp CC is a variant
  - ensures that read-only Xacts never restarted; can always read a suitable older version.
  - Additional overhead of version maintenance.