Correctness criteria: The ACID properties

- **Atomicity:** All actions in the Xact happen, or none happen.
- **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation:** Execution of one Xact is isolated from that of other Xacts.
- **Durability:** If a Xact commits, its effects persist.
Formal Properties of Schedules

- **Serial schedule**: Schedule that does not interleave the actions of different transactions.

- **Equivalent schedules**: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Conflicting Operations

- Need a tool to decide if 2 schedules are equivalent
- Use notion of “conflicting” operations

- **Definition**: Two operations conflict if:
  - They are by different transactions,
  - they are on the same object,
  - and at least one of them is a write.
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “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 |

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  | T1:  | W(A), W(B), C |
  | T2:  | W(A), W(B), C |
Conflict Serializable Schedules

- **Definition**: Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions, and
  - every pair of conflicting actions is ordered the same way

- **Definition**: Schedule S is conflict serializable if:
  - S is conflict equivalent to some serial schedule.

- Note, some “serializable” schedules are NOT conflict serializable
  - A price we pay to achieve efficient enforcement.

Conflict Serializability – Intuition

- A schedule S is conflict serializable if:
  - You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

\[
\begin{align*}
R(A) \ W(A) \ W(B) & \quad \Leftrightarrow \quad R(B) \ W(B) \\
R(A) \ W(A) & \quad R(B) \ W(B) \\
R(A) \ W(A) \ R(B) \ W(B) & \quad \equiv \quad R(A) \ W(A) \ R(B) \ W(B)
\end{align*}
\]
A Serializable (but not Conflict Serializable) Schedule

S1: \( \text{W(A=1)} \quad \text{W(A=0)} \)
S2: \( \text{W(A=2)} \quad \text{W(A=0)} \)

- Why is this not conflict serializable?
- Why is this serializable?

Dependency Graph

- **Dependency graph**:  
  - One node per Xact  
  - Edge from \( T_i \) to \( T_j \) if:  
    - An operation \( O_i \) of \( T_i \) conflicts with an operation \( O_j \) of \( T_j \)  
    - \( O_i \) appears earlier in the schedule than \( O_j \).

- **Theorem**: Schedule is conflict serializable *if and only if* its dependency graph is acyclic.
Two-Phase Locking (2PL)

rules:
- Xact must obtain a S *(shared)* lock before reading, and an X *(exclusive)* lock before writing.
- Xact cannot get new locks after releasing any locks.

<table>
<thead>
<tr>
<th>Lock Compatibility Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

Two-Phase Locking (2PL), cont.

2PL guarantees conflict serializability

But, does *not* prevent **Cascading Aborts**.
Strict 2PL

- **Problem:** Cascading Aborts
- **Example:** rollback of T1 requires rollback of T2!

| T1: R(A), W(A), R(B), W(B), Abort | T2: R(A), W(A) |

- **Strict Two-phase Locking (Strict 2PL) protocol:**
  - Same as 2PL, except:
  - **Locks released only when transaction completes**
  - i.e., either:
    - (a) transaction has committed (commit record on disk),
    - or
    - (b) transaction has aborted and rollback is complete.

---

Strict 2PL (continued)

```
# locks held
acquisition phase
release all locks at end of xact
```
Venn Diagram for Schedules

Which schedules does Strict 2PL allow?
Lock Management

• Lock and unlock requests handled by Lock Manager

• LM keeps an entry for each currently held lock.
  • Entry contains:
    – List of xacts currently holding lock
    – Type of lock held (shared or exclusive)
    – Queue of lock requests

Lock Management, cont.

• When lock request arrives:
  – Does any other xact hold a conflicting lock?
    • If no, grant the lock.
    • If yes, put requestor into wait queue.

• Lock upgrade:
  – xact with shared lock can request to upgrade to exclusive
Deadlock Detection

- Create and maintain a “waits-for” graph
- Periodically check for cycles in graph

Example:

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)

Diagram:

- T1
- T2
- T3
- T4

T1 → T2
T4 → T3
T2 → T3
Deadlock Prevention

- Assign priorities based on timestamps.
- Say Ti wants a lock that Tj holds
  Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits

- Why do these schemes guarantee no deadlocks?

- Important detail: If a transaction re-starts, make sure it gets its original timestamp. -- Why?

Multiple-Granularity Locks

- Shouldn’t have to make same decision for all transactions!
- Data “containers” are nested:
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new “intention” locks:
- Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time. Why useful?

Multiple Granularity Lock Protocol

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

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.
Lock Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>six</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>IX</td>
<td>√</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>six</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>√</td>
<td>-</td>
<td>-</td>
<td>√</td>
<td>-</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **IS** – Intent to get S lock(s) at finer granularity.
- **IX** – Intent to get X lock(s) at finer granularity.
- **SIX mode**: Like S & IX at the same time.

RECOVERY
Motivation

• **Atomicity:**
  – Transactions may abort (“Rollback”).

• **Durability:**
  – What if DBMS stops running? (Causes?)
  
  Desired state after system restarts:
  - T1 & T3 should be **durable**.
  - T2, T4 & T5 should be **aborted** (effects not seen).

Assumptions

• Concurrency control is in effect.
  – **Strict 2PL**, in particular.

• Updates are happening “in place”.
  – i.e. data is overwritten on (deleted from) the actual page copies (not private copies).

• Can you think of a simple scheme (requiring no logging) to guarantee Atomicity & Durability?
  – What happens during normal execution (what is the minimum lock granularity)?
  – What happens when a transaction commits?
  – What happens when a transaction aborts?
Buffer Management plays a key role

<table>
<thead>
<tr>
<th>No Steal</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Force</td>
<td>Fastest</td>
</tr>
<tr>
<td>Force</td>
<td>Slowest</td>
</tr>
</tbody>
</table>

Performance Implications

Logging/Recovery Implications

### Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update before the corresponding data page gets to disk.
  2. Must force all log records for a Xact before commit.

- #1 (with UNDO info) helps guarantee Atomicity.
- #2 (with REDO info) helps guarantee Durability.
- This allows us to implement Steal/No-Force

- Exactly how is logging (and recovery!) done?
  - We’ll look at the ARIES algorithm from IBM.
WAL & the Log

- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each data page contains a pageLSN.
  - The LSN of the most recent log record for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: For a page i to be written must flush log at least to the point where:
  \[ \text{pageLSN}_i \leq \text{flushedLSN} \]

The Big Picture: What’s Stored Where

- LogRecords
  - LSN
  - prevLSN
  - XID
  - type
  - pageID
  - length
  - offset
  - before-image
  - after-image
- DB
  - Data pages each with a pageLSN
- Master record
  - recLSN
- Xact Table
  - lastLSN
  - status
- Dirty Page Table
  - flushedLSN
- RAM
Crash Recovery: Big Picture

- Start from a **checkpoint** (found via **master** record).
- Three phases. Need to do:
  - **Analysis** - Figure out which Xacts committed since checkpoint, which failed.
  - **REDO** all actions.
    - (repeat history)
  - **UNDO** effects of failed Xacts.

---

Recovery: The Analysis Phase

- Re-establish knowledge of state at checkpoint.
  - **via transaction table and dirty page table** stored in the checkpoint
- **Scan** log forward from checkpoint.
  - **End** record: Remove Xact from Xact table.
  - **All Other records**: Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.
  - also, for **Update** records: If page P not in Dirty Page Table, Add P to DPT, set its recLSN=LSN.
- **At end of Analysis**...
  - transaction table says which xacts were active at time of crash.
  - DPT says which dirty pages **might not** have made it to disk
Phase 2: The REDO Phase

- We *Repeat History* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- Scan forward from log rec containing smallest `recLSN` in DPT. Q: why start here?
- For each update log record or CLR with a given LSN, *REDO* the action unless:
  - Affected page is not in the Dirty Page Table, or
  - Affected page is in D.P.T., but has `recLSN > LSN`, or
  - `pageLSN` (in DB) ≥ LSN. (this last case requires I/O)
- To *REDO* an action:
  - Reapply logged action.
  - Set `pageLSN` to LSN. No additional logging, no forcing!

Phase 3: The UNDO Phase

- A Naïve solution:
  - The xacts in the Xact Table are losers.
  - For each loser, perform simple transaction abort.
  - Problems?
Phase 3: The UNDO Phase

\( \text{ToUndo} = \{ \text{lastLSNs of all Xacts in the Xact Table} \} \)

a.k.a. "losers"

Repeat:
  – Choose (and remove) largest LSN among ToUndo.
  – If this LSN is a CLR and undonextLSN==NULL
    • Write an End record for this Xact.
  – If this LSN is a CLR, and undonextLSN != NULL
    • Add undonextLSN to ToUndo
  – Else this LSN is an update. Undo the update, write a CLR, add prevLSN
to ToUndo.

Until ToUndo is empty.

NOTE: This is simply a performance optimization on the naïve solution to do it in
one backwards pass of the log!

Summary of Logging/Recovery

• Recovery Manager guarantees Atomicity & Durability.

• Use WAL to allow \texttt{STEAL/NO-FORCE} w/o sacrificing correctness.

• LSNs identify log records; linked into backwards chains per transaction (via prevLSN).

• pageLSN allows comparison of data page and log records.
Summary, Cont.

• **Checkpointing**: A quick way to limit the amount of log to scan on recovery.

• Recovery works in 3 phases:
  – **Analysis**: Forward from checkpoint.
  – **Redo**: Forward from oldest recLSN.
  – **Undo**: Backward from end to first LSN of oldest Xact alive at crash.

• Upon Undo, write CLRs.

• Redo “repeats history”: Simplifies the logic!