Lock Management

- Lock and unlock requests are handled by the lock manager (see Sec. 17.2.1)

- Lock table entry:
  - Lock name/identifier
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests

- Locking and unlocking have to be atomic operations (need mutex protection)

- Lock table entries are kept in order, to prevent starvation (lots of reads preventing a writer from ever getting a lock, etc.)
Lock Manager Data structure: a multilist

- Need access to lock entry by lock name or transaction id
- Some of these transactions are blocked on the lock
Multiple-Granularity Locks

- Hard to decide what granularity to lock
  - tuples vs. pages vs. files
  - Inefficient to have a million row locks to scan a relation
- Shouldn’t have to decide once and for all!
- Data containers are nested:
New Lock Modes, Protocol

- Allow transactions to lock at each level, but with a special protocol using new intention locks

- Before locking an item, must set intention locks on ancestors

- To lock an item with an S lock (X lock), need an IS (IX) lock or stronger on ancestors

- For unlock, go from specific to general (i.e., bottom-up).

- **SIX mode**: Like S & IX at the same time.

```
- - IS IX S X
--- √ √ √ √ √
IS √ √ √ √ √
IX √ √ √
S √ √ √
X √
```
New Lock Modes, Protocol

- Lock manager doesn’t care: just make up lock names with table name or item id, use new lock compatibility table
- Protocol makes client check higher level(s) first, then target level, so lock manager itself (or its kernel part) has no responsibility to know relationship between locks

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<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>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>
New Lock Modes, strength of locks

- Before locking an item, must set intention locks (IS/IX) on ancestors, or stronger locks
- IS is the weakest lock: it only blocks an X-locker (of a different transaction)
- IX is stronger than IS because it blocks an S-locker or an X-locker
- X is stronger than any other lock: it blocks all locks attempts by other transactions
- IX and S are not comparable this way
- SIX: blocks all but IS locks

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<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>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>
Multiple Granularity Lock Protocol

- Each transaction starts from the root of the hierarchy

- To get S or IS lock on a node, must hold IS on parent node, or the stronger S or IX or X locks

- To get X or IX or SIX on a node, must hold IX or the stronger SIX or X on parent node.

- Must release locks in bottom-up order
Examples: two levels, relation and tuples

- **T1** scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.

- **T2** uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R. If overlapping with T1, gets the IS lock on R, but may block on X-locked tuples.

- **T3** reads all of R:
  - T3 gets an S lock on R. If overlapping with T1, will block until T1’s SIX lock is released.
  - OR, T3 could behave like T2; can use lock escalation to decide which.

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<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>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>
Isolation Levels in Practice

- Databases default to RC, read-committed, so many apps run that way, can have their read data changed, and phantoms
- Web apps (JEE, anyway) have a hard time overriding RC, so most are running at RC
- The 2PL locking scheme we studied was for RR, repeatable read: transaction takes long term read and write locks
- Long term = until commit of that transaction
Read Committed (RC) Isolation

- 2PL can be modified for RC: take long-term write locks but not long term read locks.
- Reads are atomic as operations, but that’s it.
- Lost updates can happen in RC: system takes 2PC locks only for the write operations:
  \[ R1(A)R2(A)W2(B)C2W1(B)C1 \]
  \[ R1(A)R2(A)X2(B)W2(B)C2X1(B)W1(B)C1 \] (RC isolation)
- Update statements are atomic, so that case of read-then-write is safe even at RC.
- Update T set A = A + 100 (safe at RC isolation)
- Remember to use update when possible!
Syntax for SQL

**SET TRANSACTION ISOLATION LEVEL**
- SERIALIZABLE READ WRITE
- REPEATABLE READ
- READ ONLY

- **Note:**
  - READ UNCOMMITTED cannot be READ WRITE
More on setting transaction properties

**Embedded SQL**

```
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

**JDBC**

```java
conn.setAutoCommit(false);
conn.setTransactionIsolation(
    (Connection.TRANSACTION_Isolation_level.SERIALIZABLE);
```
Snapshot Isolation (SI)

- Multiversion Concurrency Control Mechanism (MVCC)
- This means the database holds more than one value for a data item at the same time

- Used in Oracle, PostgreSQL (as option), MS SQL Server (as option), others

- Readers never conflict with writers unlike traditional DBMS (e.g., IBM DB2)! Read-only transactions run fast.

- Does not guarantee “real” serializability, unless fixed up, i.e., has anomalies. “Serializable Snapshot Isolation” available now in Postgres. Oracle allows SI anomalies.

- But: avoids all anomalies in the ANSI table, so seems OK.
- We found in use at Microsoft in 1993, published as example of MVCC
Snapshot Isolation - Basic Idea:

- Every transaction reads from its own snapshot (copy) of the database (will be created when the transaction starts, or reconstructed from the undo log).
- Writes are collected into a writeset (WS), not visible to concurrent transactions.
- Two transactions are considered to be concurrent if one starts (takes a snapshot) while the other is in progress.
First Committer Wins Rule of SI

- At the commit time of a transaction its WS is compared to those of concurrent committed transactions.
- If there is no conflict (overlapping), then the WS can be applied to stable storage and is visible to transactions that begin afterwards.
- However, if there is a conflict with the WS of a concurrent, already committed transaction, then the transaction must be aborted.
- That’s the “First Committer Wins Rule“
- Actually Oracle uses first updater wins, basically same idea, but doesn’t require separate WS
**Write Skew Anomaly of SI**

- In MVCC, data items need subscripts to say which version is being considered
  - Zero version: original database value
  - T1 writes new value of X, $X_1$
  - T2 writes new value of Y, $Y_2$
- **Write skew anomaly schedule:**
  
  $R_1(X_0) \ R_2(X_0) \ R_1(Y_0) \ R_2(Y_0,) \ W_1(X_1) \ C_1 \ W_2(Y_2) \ C_2$

- Writesets $WS(T1) = \{X\}, WS(T2) = \{Y\}$, do not overlap, so both commit.
- So what’s wrong—where’s the anomaly?
Write Skew Anomaly of SI

\[ R_1(X_0) \ R_2(X_0) \ R_1(Y_0) \ R_2(Y_0) \ \mathcal{W}_1(X_1) \ C_1 \ \mathcal{W}_2(Y_2) \ C_2 \]

- **Scenario:**
  - \( X \) = husband’s balance, orig 100,
  - \( Y \) = wife’s balance, orig 100.
  - Bank allows withdrawals up to combined balance
  - Rule: \( X + Y \geq 0 \)
  - Both withdraw 150, thinking OK, end up with -50 and -50.
- Easy to make this happen in Oracle at “Serializable” isolation.
- See conflicts, cycle in PG, can’t happen with full 2PL
- Can happen with RC/locking
How can an Oracle app handle this?

- If $X+Y \geq 0$ is needed as a constraint, it can be “materialized” as sum in another column value.

- Old program: $R(X)R(X\text{-spouse})W(X)C$

- New program: $R(X)R(X\text{-spouse})W(\text{sum})W(X)C$

- So schedule will have $W(\text{sum})$ in both transactions, and sum will be in both Writesets, so second committer aborts.
Oracle, Postgres: new failure to handle

- Recall deadlock-abort handling: retry the aborted transaction
- With SI, get "can't serialize access"
  - ORA-08177: can't serialize access for this transaction
  - Means another transaction won for a contended write
- App handles this error like deadlock-abort: just retry transaction, up to a few times
- This only happens when you set serializable isolation level
Other anomalies under SI

- **Oldest sailors example**
  - Both concurrent transactions see original sailor data in snapshots, plus own updates
  - Updates are on different rows, so both commit
  - Neither sees the other’s update
  - So not serializable: one should see one update, other should see two updates.

- **Task Registry example:**
  - Both concurrent transactions see original state with 6 hours available for Joe
  - Both insert new task for Joe
  - Inserts involve different rows, so both commit
Fixing the task registry phantom problem

- Following the idea of the simple write skew, we can materialize the constraint “workhours <= 8”
- Add a workhours column to worker table
- Old program:
  - if sum(hours-for-x) + newhours <= 8
  - insert new task
- New program:
  - if workhours-for-x + newhours <= 8
  - { update worker set workhours = workhours + newhours…
  - insert new task
  - }
Fixing the Oldest sailor example

- If the oldest sailor is important to the app, materialize it!

Create table oldestsailor (rating int primary key, sid int)
Oracle Read Committed Isolation

- **READ COMMITTED** is the default isolation level for both Oracle and PostgreSQL.
- A new snapshot is taken for every issued SQL statement (every statement sees the latest committed values).
- If a transaction T2 running in READ COMMITTED mode tries to update a row which was already updated by a concurrent transaction T1, then T2 gets blocked until T1 has either committed or aborted.
- Nearly same as 2PL/RC, though all reads occur effectively at the same time for the statement.
ACID Properties

Transaction Management must fulfill four requirements:

1. **Atomicity**: either all actions within a transaction are carried out, or none is
   - Only actions of committed transactions must be visible
2. **Consistency**: concurrent execution must leave DBMS in consistent state
3. **Isolation**: each transaction is protected from effects of other concurrent transactions
   - Net effect is that of some sequential execution
4. **Durability**: once a transaction commits, DBMS changes will persist
   - Conversely, if a transaction aborts/is aborted, there are no effects
Recovery Manager

- Crash recovery
  - Ensure that atomicity is preserved if the system crashes while one or more transactions are still incomplete
  - Main idea is to keep a log of operations; every action is logged before its page updates reach disk (Write-Ahead Log or WAL)

- The Recovery Manager guarantees Atomicity & Durability
Motivation

- **Atomicity:**
  - Transactions may abort – must **rollback** their actions

- **Durability:**
  - What if DBMS stops running – e.g., power failure?

Desired Behavior after system restarts:

- T1, T2 & T3 should be durable
- T4 & T5 should be aborted (effects not seen)
Assumptions

- Concurrency control is in effect
  - Strict 2PL

- Updates are happening “in place”
  - Data overwritten on (deleted from) the disk

- A simple scheme is needed
  - A protocol that is too complex is difficult to implement
  - Performance is also an important issue