Last Class

- A *concurrency control* scheme uses locks and aborts to ensure correctness.
- Conflict vs. View Serializability
- (Strict) 2PL is popular.
- We need to handle deadlocks in 2PL:
  - Detection: *Waits-for* graph
  - Prevention: Abort some txns, defensively

Last Class Assumption

- We assumed that the database was *fixed* collection of *independent* objects.
  - No objects are added or deleted.
  - No relationship between objects.
  - No indexes.
Today’s Class

- Lock Granularities
- Locking in B+Trees
- The Phantom Problem
- Transaction Isolation Levels

Lock Granularities

- When we say that a txn acquires a “lock”, what does that actually mean?
  - On a field? Record? Page? Table?
- Ideally, each txn should obtain fewest number of locks that is needed...

Database Lock Hierarchy
Example

• **T1:** Get the balance of Christos’ shady off-shore bank account.
• **T2:** Increase all account balances by 1%.
• **Q:** What locks should they obtain?

Example

• **Q:** What locks should they obtain?
• **A:** Multiple
  – **Exclusive + Shared** for leafs of lock tree.
  – Special **Intention** locks for higher levels

Intention Locks

• Intention locks allow a higher level node to be locked in **S** or **X** mode without having to check all descendent nodes.
• If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.
Intention Locks

- **Intention-Shared (IS):** Indicates explicit locking at a lower level with shared locks.
- **Intention-Exclusive (IX):** Indicates locking at lower level with exclusive or shared locks
- **Shared+Intention-Exclusive (SIX):** The subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</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>S</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>X</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Multiple Granularity Protocol

- **Weaker**
- **Stronger**

Privileges

- IS
- IX
- S
- SIX
- X
Locking Protocol

- Each txn obtains appropriate lock at highest level of the database hierarchy.
- To get S or IS lock on a node, the txn must hold at least IS on parent node.
  - What if txn holds SIX on parent? S on parent?
- To get X, IX, or SIX on a node, must hold at least IX on parent node.

Example – Two-level Hierarchy

Example – Threesome

- Assume three txns execute at same time:
  - T1: Scan R and update a few tuples.
  - T2: Scan a portion of tuples in R.
  - T3: Scan all tuples in R.
Example – Threesome

• **T1**: Get an **SIX** lock on $R$, then get **X** lock on tuples that are updated.
• **T2**: Get an **IS** lock on $R$, and repeatedly get an **S** lock on tuples of $R$.
• **T3**: Two choices:
  – T3 gets an **S** lock on $R$.
  – OR, T3 could behave like T2; can use lock escalation to decide which.

Lock Escalation

• Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired.
Multiple Lock Granularities

- Useful in practice as each txn only needs a few locks.
- Intention locks help improve concurrency:
  - Intention-Shared (IS): Intent to get S lock(s) at finer granularity.
  - Intention-Exclusive (IX): Intent to get X lock(s) at finer granularity.
  - Shared+Intention-Exclusive (SIX): Like S and IX at the same time.

Today’s Class

- Lock Granularities
- Locking in B+Trees
- The Phantom Problem
- Transaction Isolation Levels

Locking in B+Trees

- Q: What about locking indexes?
- A: They are not quite like other database elements so we can treat them differently:
  - It’s okay to have non-serializable concurrent access to an index as long as the accuracy of the index is maintained.
**Example**

- T1 wants to insert in H
- T2 wants to insert in I
- Q: Why not plain 2PL?
- A: Because txns have to hold on to their locks for too long!

**Lock Crabbing**

- Improves concurrency for B+Trees.
- Get lock for parent; get lock for child; release lock for parent if “safe”.
- **Safe Nodes**: Any node that won’t split or merge when updated.
  - Not full (on insertion)
  - More than half-full (on deletion)

**Lock Crabbing**

- **Search**: Start at root and go down; repeatedly,
  - S lock child
  - then unlock parent
- **Insert/Delete**: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
  - If child is safe, release all locks on ancestors.
**Example #1 – Search 38**

We know that C will not need to merge with F, so it’s safe to release A + B.

**Example #2 – Delete 38**

We know that C will not need to merge with F, so it’s safe to release A + B.

**Example #3 – Insert 45**

We know that if C needs to split, B has room so it’s safe to release A. C won’t split, so we can release B + C.
Example #4 – Insert 25

We need to split H so we need to keep the lock on its parent node.

Problems

Q: What was the first step that all of the update examples did on the B+Tree?

A: Locking the root every time becomes a bottleneck with higher concurrency.

Can we do better?
Better Tree Locking Algorithm

• Main Idea:
  – Assume that the leaf is ‘safe’, and use S-locks & crabbing to reach it, and verify.
  – If leaf is not safe, then do previous algorithm.

Better Tree Locking Algorithm

• **Search:** Same as before.
• **Insert/Delete:**
  – Set locks as if for search, get to leaf, and set X lock on leaf.
  – If leaf is not safe, release all locks, and restart txn using previous Insert/Delete protocol.
• Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful.

Example #2 – Delete 38

D will not need to coalesce, so we’re safe!
Example #4 – Insert 25

We need to split H so we have to restart and re-execute like before.

Another Alternative

• Textbook has a third variation, that uses lock-upgrades instead of restarting.
• This approach may lead to deadlocks.

Additional Points

• Q: Which order to release locks in multiple-granularity locking?
  • A: From the bottom up

• Q: Which order to release locks in tree-locking?
  • A: As early as possible to maximize concurrency.
Today’s Class

- Lock Granularities
- Locking in B+Trees
- The Phantom Problem
- Transaction Isolation Levels

Dynamic Databases

- Recall that so far we have only dealing with transactions that read and update data.
- But now if we have insertions deletions, we have new problems…

The Phantom Problem

Schedule

```
BEGIN
SELECT MAX(age) FROM sailors WHERE rating=1
COMMIT
```

```
BEGIN
INSERT INTO sailors (age=96, rating=1)
SELECT MAX(age) FROM sailors WHERE rating=1
COMMIT
```

```
BEGIN
SELECT MAX(age) FROM sailors WHERE rating=1
COMMIT
```

```
BEGIN
COMMIT
```

```
COMMIT
```

```
COMMIT
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

TIME

T1

T2
How did this happen?

- Because T1 locked only existing records and not ones under way!
- Conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed.
- Solution?

Predicate Locking

- Lock records that satisfy a logical predicate:
  - Example: \texttt{rating=1}.
- In general, predicate locking has a lot of locking overhead.
- \textbf{Index locking} is a special case of predicate locking that is potentially more efficient.

Index Locking

- If there is a dense index on the \texttt{rating} field then the \texttt{txn} can lock index page containing the data with \texttt{rating=1}.
- If there are no records with \texttt{rating=1}, the \texttt{txn} must lock the index page where such a data entry would be, if it existed.
Locking without an Index

- If there is no suitable index, then the txn must obtain:
  - A lock on every page in the table to prevent a record’s rating from being changed to 1.
  - The lock for the table itself to prevent records with rating=1 from being added or deleted.

Phantom Problem

Today’s Class

- Lock Granularities
- Locking in B+Trees
- The Phantom Problem
- Weaker Levels of Consistency
Weaker Levels of Consistency

- Serializability is useful because it allows programmers to ignore concurrency issues.
- But enforcing it may allow too little concurrency and limit performance.
- We may want to use a weaker level of consistency to improve scalability.

Isolation Levels

- Controls the extent that a txn is exposed to the actions of other concurrent txns.
- Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
  - Dirty Reads
  - Unrepeatable Reads
  - Phantom Reads

Isolation Levels

- **SERIALIZABLE**: No phantoms, all reads repeatable, no dirty reads.
- **REPEATABLE READS**: Phantoms may happen.
- **READ COMMITTED**: Phantoms and unrepeatable reads may happen.
- **READ UNCOMMITTED**: All of them may happen.
Isolation Levels

<table>
<thead>
<tr>
<th></th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ UNCOMMITTED</strong></td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>READ COMMITTED</strong></td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>REPEATABLE READ</strong></td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td><strong>SERIALIZABLE</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

CMU SCS

SQL-92 Isolation Levels

```
SET TRANSACTION ISOLATION LEVEL <isolation-level>;
```

- Default: **SERIALIZABLE**
- Not all DBMS support all isolation levels in all execution scenarios (e.g., replication).
Access Modes

- You can also provide hints to the DBMS about whether a txn will modify the database.
- Only two possible modes:
  - READ WRITE
  - READ ONLY

SQL-92 Access Modes

- Default: READ WRITE
- Not all DBMSs will optimize execution if you set a txn to in READ ONLY mode.

Transaction Demo
Summary

• Multiple granularity locking: leads to few locks, at appropriate levels
• Tree-structured indexes:
  – Lock crabbing and safe nodes
• Important distinction:
  – Multiple granularity locking releases locks bottom-up.
  – Tree-locking releases top-down to maximize concurrency.

Summary

• The Phantom Problem occurs if insertions/deletions
• Use Predicate locking to prevent this:
  – Index Locking
  – Table Locking