Introduction to Data Management

Transaction: Isolation Levels

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Recap

- Schedules under 2PL are conflict serializable
  - Locking phase → unlocking phase

- Conflict serializable schedules follow the isolation principle of ACID
  - No dirty read (WR)
  - No unrepeatable read (RW)
  - No lost update (WW)

- Schedules under strict 2PL additionally provide recoverability
  - Locking phase → unlock with commit or rollback

- Deadlocks can still occur
Outline

- Shared/Exclusive locks
- Isolation levels
- Phantom reads
- Transactions demo
Practicality of Binary Locks

- Binary Locks offer full control or no control
- Leads to excessive deadlock
Thrashing

Throughput (txns/sec = TPS) vs. # of active transactions

thrashing
Observation: Reads don’t conflict with each other

Simple 3-tier lock hierarchy:
- Exclusive/Write Lock \( X_i(A) \)
  - Full control
  - No other locks may exist concurrently
- Shared/Read Lock \( S_i(A) \)
  - Shared control
  - May exist concurrently with other shared locks
- Unlocked
Lock Compatibility Matrix

Is the requested lock granted...

...if the element lock is in this state?

<table>
<thead>
<tr>
<th></th>
<th>unlocked</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Practicality of Serializability

▪ **Easy to reason about**
  • Application programming is easier under serializability assumptions

▪ **Expensive to use**
  • Slow
  • Resource intensive

▪ **Applications often don’t need serializability**
  • Application functionality may not depend on serializability
  • Financial/User experience cost is low enough for tradeoff considerations
Isolation Levels

- SET TRANSACTION ISOLATION LEVEL ...
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
  - SNAPSHOT ISOLATION
  - …

- Default isolation level and configurability depends on the DBMS (read the docs)
- **Writes**  △  Strict 2PL write locks
- **Reads**  △  No locks needed
- **Dirty reads are possible**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td>ABORT U(A)</td>
<td></td>
</tr>
</tbody>
</table>
READ COMMITTED

- **Writes**: Strict 2PL write locks
- **Reads**: Short-duration read locks
  - Acquire lock before reading and release lock after (not 2PL)
- **Dirty reads are prevented**

---

**TABLE 1**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>ABORT U(A)</td>
<td></td>
</tr>
</tbody>
</table>

---

**TABLE 2**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(A) blocked…</td>
</tr>
<tr>
<td>ABORT U(A)</td>
<td>…granted S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT U(A)</td>
</tr>
</tbody>
</table>
- Unrepeatable reads are possible

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) blocked…</td>
<td>S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>…granted X(A)</td>
<td>U(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>S(A) blocked…</td>
</tr>
<tr>
<td>COMMIT U(A)</td>
<td>…granted S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT U(A)</td>
</tr>
</tbody>
</table>
### REPEATABLE READ

- **Writes** □ Strict 2PL write locks
- **Reads** □ Strict 2PL read locks
- **Unrepeatable reads are prevented**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(A) blocked…</td>
<td>S(A)</td>
<td>X(A) blocked…</td>
<td>S(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>…granted X(A)</td>
<td>L(A)</td>
<td>…granted X(A)</td>
<td>L(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>S(A) blocked…</td>
<td>W(A)</td>
<td>S(A) blocked…</td>
</tr>
<tr>
<td>COMMIT U(A)</td>
<td>…granted S(A)</td>
<td>COMMIT U(A)</td>
<td>…granted S(A)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT U(A)</td>
<td>COMMIT U(A)</td>
<td>COMMIT U(A)</td>
</tr>
</tbody>
</table>
REPEATABLE READ

- Writes □ Strict 2PL write locks
- Reads □ Strict 2PL read locks
- **Phantom reads are possible...**
Phantom Reads

- Conflict serializability implying serializability assumes a **static database**
  - Conflicts only matter for the same element
  - Inserting a new element (tuple-level granularity) means that the conflict model no longer is able to encapsulate it

```
SELECT * FROM Table;
R(A)
R(B)
I(C)
INSERT INTO Table VALUES (C...);

SELECT * FROM Table;
R(A)
R(B)
R(C)
```
Phantom Reads

- **Dynamic database** serializability needs either:
  - Table locking (prevent insertions) or
  - Predicate locking (lock based on query filters)
SERIALIZABLE

- Write Lock □ Strict 2PL
- Read Lock □ Strict 2PL
- Plus predicate locks and/or table locks
## Isolation Level Summary

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>READ COMMITED</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No anomalies!
Applications generally need to
- Check/Set isolation levels
- Specify operations as transactions

Common mistakes/misconceptions:
- You do not need to implement locking. The DBMS takes care of it.
- You must **close all explicit transactions** with COMMIT or ROLLBACK. Not doing so will cause the application to hang (wait due to unfinished locking).
Applications generally need to
  • **Check/Set isolation levels**
  • **Specify operations as transactions**

Common mistakes/misconceptions:
  • You do not need to implement locking. The DBMS takes care of it.
  • You must **close all explicit transactions** with COMMIT or ROLLBACK. Not doing so will cause the application to hang (wait due to unfinished locking).

One should *execute* per transaction, may have multiple in code:

```java
if (failure_case) {
    rollback();
}
...
commit();
```
Case Study: SQLite

- SQLite uses a relatively shared lock system

- Lock types
  - READ LOCK (to read)
  - RESERVED LOCK (to write)
  - PENDING LOCK (wants to commit)
  - EXCLUSIVE LOCK (to commit)
Case Study: SQLite

**Step 1:** when a transaction begins

- Acquire a **READ LOCK** (aka "SHARED" lock)
- All these transactions may read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock
**Step 2:** when one transaction wants to write

- Acquire a **RESERVED LOCK**
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a RESERVED LOCK
Step 3: when writer transaction wants to commit, it needs exclusive lock, which can’t coexists with read locks

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKs are accepted
- Wait for all read locks to be released
Case Study: SQLite

**Step 4**: when all read locks have been released

- Acquire the **EXCLUSIVE LOCK**
- Nobody else can touch the database now
- All updates are written permanently to the database file
- Release the lock and COMMIT
Case Study: SQLite

Isolation Levels

None → READ LOCK → RESERVED LOCK → PENDING LOCK → EXCLUSIVE LOCK

- begin transaction
- first write
- commit requested
- no more read locks

commit
commit executed
Demo