Concurrency control

12/1/17
Bag of words...

- Isolation
- Linearizability
- Consistency
- Strict serializability
- Durability
- Snapshot isolation
- Conflict equivalence
- Serializability
- Optimistic concurrency control
- Multiversion concurrency control
- Atomicity
- Two-phase locking
- Conflict serializability
ACID semantics

Relevant in the context of database transactions (txn)

Atomicity: Either all ops happen or no ops happen
Consistency: Application constraints are not violated
Isolation: Concurrent txns appear as if executed serially
Durability: Results of committed txns survive failures
Consistency disambiguation

Consistency in ACID refers to integrity constraints in applications
   e.g. Bank account balance should always be \( \geq 0 \)

Consistency in context of availability refers to linearizability
   Linearizability: once a write completes, all later reads should see that value

Consistency here describes guarantees about a single item
   e.g. CAP theorem, Dynamo
Isolation

How to ensure *correctness* when running concurrent txns?
Problems caused by concurrency?

Lost update: the result of a txn is overwritten by another txn

Dirty read: uncommitted results can be read by a txn

Non-repeatable read: two reads in the sametxn can return different results

Phantom read: later reads in the same txnn can return extra rows

BEGIN TRANSACTION
SELECT * FROM students
SELECT * FROM students
COMMIT

BEGIN TRANSACTION
UPDATE students SET gpa = 3.6 WHERE id = 1
INSERT INTO students VALUES (2, "Jack", 4.0)
COMMIT
Serial schedule — no problems

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

T2: R(A), W(A), Commit
Quiz: Which concurrency problem is this?

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

Dirty read
Quiz: Which concurrency problem is this?

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

Non-repeatable read
Quiz: Which concurrency problem is this?

T1: R(A), W(A)          W(B), Commit
T2: R(A)          W(A), W(B), Commit

Lost update
Quiz: Which concurrency problem is this?

T1: R(A), W(A)  W(A), Commit
T2: R(A), R(B), W(B) Commit

Dirty read
Levels of isolation

- Serializability
- Read committed
- Repeatable reads
- Read uncommitted

Stronger → Weaker
Levels of isolation

**Read uncommitted**: no restrictions on reads

**Read committed**: no dirty reads

**Repeatable reads**: rows returned by two reads in the sametxn are unchanged

**Serializability**: txns behave as if executed one after another (strongest)
## Levels of isolation

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty read</th>
<th>Nonrepeatable read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Read committed</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Snapshot</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Fixing concurrency problems

Strawman: Just run txns serially — prohibitive performance

Observation: Problems only arise when

1. Two txns touch the same table
2. At least one of these txns involve a write to the table

Key idea: Permit schedules whose effects are equivalent to serial schedules
Conflict serializability

Two operations conflict if

1. They belong to different txns
2. They operate on the same data
3. One of them is a write

Two operations are conflict equivalent if

1. They involve the same operations
2. All conflicting operations are ordered the same way

A schedule is conflict serializable if it is conflict equivalent to a serial schedule
Testing for conflict serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), \( W(A) \), Commit

T2: R(A), R(B), W(B) Commit
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: \( R(A), \) \( W(A) \), Commit

T2: \( R(A), R(B), W(B) \) Commit
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

Conflict serializable
Testing for conflict serializability

Intuition: Swap non-conflicting operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

- time
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), \textit{W(A)}, \quad \textit{W(B)}, \text{Commit}

T2: \quad \textit{R(B), W(B)}, \textit{R(A)} \text{ Commit}
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: \( R(A), W(A) \) \( W(B), \text{Commit} \)

T2: \( R(B), W(B), R(A) \) \( \text{Commit} \)
Testing for conflict serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: \( \text{R}(A), \text{W}(A), \text{W}(B), \text{Commit} \)

T2: \( \text{R}(B), \text{W}(B), \text{R}(A) \text{ Commit} \)

NOT conflict serializable
Testing for conflict serializability

Another way to test conflict serializability:

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between txns, the schedule is conflict serializable
Testing for conflict serializability

Another way to test conflict serializability:

Draw arrows between conflicting operations
Arrow points in the direction of time
If no cycles between txns, the schedule is conflict serializable

T1: R(A), W(A), Commit
T2: R(A), R(B), W(B) Commit
Testing for conflict serializability

Another way to test conflict serializability:
- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between txns, the schedule is conflict serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B), Commit

No cycles, conflict serializable
Testing for conflict serializability

Another way to test conflict serializability:

Draw arrows between conflicting operations

Arrow points in the direction of time

If no cycles between txns, the schedule is conflict serializable

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

Cycles exist, NOT conflict serializable
Implementing conflict serializability

Two-phase locking (2PL): acquire all locks before releasing any locks
Each txn acquires shared locks (S) for reads and exclusive locks (X) for writes

2PL guarantees conflict serializability by disallowing cycles
- Edge from Ti to Tj means Ti acquired lock first and Tj has to wait
- Edge from Tj to Ti means Tj acquired lock first and Ti has to wait
- Cycles mean DEADLOCK
Implementing conflict serializability

Two-phase locking (2PL): acquire all locks before releasing any locks

Each txn acquires shared locks (S) for reads and exclusive locks (X) for writes

T1: R(A), W(A), W(B), Commit
T2: R(B), W(B), R(A), Commit

DEADLOCK!
Implementing conflict serializability

Two-phase locking (2PL): acquire all locks before releasing any locks
Eachtxn acquires shared locks (S) for reads and exclusive locks (X) for writes

2PL guarantees conflict serializability by disallowing cycles
  Edge from Ti to Tj means Ti acquired lock first and Tj has to wait
  Edge from Tj to Ti means Tj acquired lock first and Ti has to wait
  Cycles mean DEADLOCK
  Deal with deadlocks by aborting one of the two txns (e.g. detect + timeout)
2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

```
Lock_X(A)  Unlock_X(A)
```

T1: \(R(A), W(A), \) Abort

T2: \(R(B), W(B), R(A)\) Abort

Rollback of T1 requires rollback of T2, since T2 read a value written by T1

Cascading aborts: the rollback of one txn causes the rollback of another
Strict 2PL

Release locks at the end of the txn

Variant of 2PL implemented by most databases in practice
Is this a 2PL schedule?

No, and it is not conflict serializable
Is this a 2PL schedule?
Yes, and it is conflict serializable

Is this a Strict 2PL schedule?
No, cascading aborts possible
<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A) &lt;granted&gt;</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Read(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>A: = A-50</td>
<td>Write(A)</td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B) &lt;granted&gt;</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := B +50</td>
</tr>
<tr>
<td>B := B +50</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>

Is this a 2PL schedule?
Yes, and it is conflict serializable

Is this a Strict 2PL schedule?
Yes, cascading aborts not possible
Recap

Isolation  Linearizability  Consistency

Strict serializability  Durability  Snapshot isolation

Conflict equivalence

Serializability

Atomicity  Optimistic concurrency control  Multiversion concurrency control

Two-phase locking  Conflict serializability
Recap

- Linearizability
  - Strict serializability
  - Conflict equivalence
  - Optimistic concurrency control
  - Two-phase locking

- Serializability
  - Snapshot isolation

- Conflict serializability

- Multiversion concurrency control
Recap

- Strict serializability
- Conflict equivalence
- Optimistic concurrency control
- Two-phase locking
- Conflict serializability
- Snapshot isolation
- Serializability
- Multiversion concurrency control
Recap

- Optimistic concurrency control
- Snapshot isolation
- Multiversion concurrency control
Two ways of implementing serializability

Issues with 2PL (pessimistic):

1. Assume conflict, always lock
2. High overhead for non-conflicting txn
3. Must check for deadlock

Optimistic concurrency control (OCC):

1. Assume no conflict
2. Low overhead for low-conflict workloads
3. Ensure correctness by aborting txns if conflict occurs
Optimistic concurrency control

Modify (Read): Read committed values, write changes locally

Verify: Check if a conflict would occur at commit

Commit (Write): If no conflict, commit, else abort
Test 1

For all $i$ and $j$ such that $T_i < T_j$, check that $T_i$ completes before $T_j$ begins.
Test 2

For all $i$ and $j$ such that $T_i < T_j$, check that:

- $T_i$ completes before $T_j$ begins its Write phase
- $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_j)$ is empty
Test 3

For all i and j such that Ti < Tj, check that:

- Ti completes Read phase before Tj does
- WriteSet(Ti) ∩ ReadSet(Tj) is empty
- WriteSet(Ti) ∩ WriteSet(Tj) is empty
Levels of isolation

- Serializability
- Repeatable reads
- Read uncommitted

- Stronger
- Snapshot isolation
- Read committed
- Weaker
Snapshot isolation

All reads see a consistent snapshot of the database

Commit only if no write-write conflicts with concurrent txns

Intuition: each write creates a new snapshot, and concurrent reads may return values from older snapshots
Snapshot isolation advantages

Super fast reads + most concurrency problems are solved

- No non-repeatable reads
- No dirty reads
- No lost updates
- (why?)
Snapshot isolation < serializability

Write skew problem: txns modify different items (hence no write conflict) but violate integrity constraints

Rare in practice!
Snapshot isolation implementation

Most popular implementation: multiversion concurrency control (MVCC)

Each txn T is assigned a timestamp TS

Reads return the latest value written before TS

Writes abort if another txn has updated the value in the same snapshot after TS

(Details in lecture)
Further reading

https://inst.eecs.berkeley.edu/~cs186/fa05/lecs/17TransIntro-6up.pdf

https://inst.eecs.berkeley.edu/~cs186/fa05/lecs/18cc-6up.pdf


https://db.in.tum.de/teaching/ws1314/transactions/pdf/SnapshotIsolation.pdf?lang=de
