Transactions & Update Correctness

April 11, 2017
Correctness

• Data Correctness (Constraints)

• Query Correctness (Plan Rewrites)

• Update Correctness (Transactions)
What could go wrong?

• **Parallelism**: What happens if two updates modify the same data?

• Maximize use of IO / Minimize Latencies.

• **Persistence**: What happens if something breaks during an update?

• When is my data safe?
What does it mean for a database operation to be correct?
What is an Update?

- INSERT INTO ...?
- UPDATE ... SET ... WHERE ...?
- Non-SQL?

Can we abstract?
Abstract Update Operations
What does it mean for a database operation to be correct?
Transaction Correctness

- Reliability in database transactions guaranteed by ACID

- A - Atomicity ("Do or Do Not, there is nothing like try") - usually ensured by logs

- C - Consistency ("Within the framework of law") - usually ensured by integrity constraints, validations, etc.

- I - Isolation ("Execute in parallel or serially, the result should be same") - usually ensured by locks

- D - Durability ("once committed, remain committed") - usually ensured at hardware level
Atomicity

- A transaction completes by committing, or terminates by aborting.
- **Logging** is used to undo aborted transactions.
- **Atomicity**: A transaction is (or appears as if it were) applied in one ‘step’, independent of other transactions.
- All ops in a transaction commit or abort together.
Isolation

T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END

• Intuitively, T1 transfers $100 from A to B and T2 credits both accounts with interest.

• What are possible interleaving errors?
Example: Schedule

Time

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100</td>
<td>A=1.06*A</td>
</tr>
<tr>
<td>B=B−100</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

OK!
Example: Schedule

Time

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A = A + 100</td>
<td>A = 1.06 * A</td>
</tr>
<tr>
<td>B</td>
<td>B = B - 100</td>
<td>B = 1.06 * B</td>
</tr>
</tbody>
</table>

Not OK!
Example: The DBMS’s View

Time

T1

R(A)
W(A)

T2

R(A)
W(A)
R(B)
W(B)

R(B)
W(B)

Not OK!
What went wrong?
What could go wrong?

Reading uncommitted data
(write-read/WR conflicts; aka “Dirty Reads”)

T1: R(A), W(A),
T2: R(A),W(A),CMT,
T1: R(B),W(B),ABRT
T2: R(A),W(A),CMT,

Unrepeatable Reads
(read-write/RW conflicts)

T1: R(A),
T2: R(A),W(A),CMT,
T1: R(A),W(A),CMT
T2: R(A),W(A),CMT,
What could go wrong?

Overwriting Uncommitted Data
(write-write/WW conflicts)

T1: \( W(A), W(B), CMT \)
T2: \( W(A), W(B), CMT \)
Schedule
An ordering of read and write operations.

Serial Schedule
No interleaving between transactions at all

Serializable Schedule
Guaranteed to produce equivalent output to a serial schedule
Conflict Equivalence

Possible Solution: Look at read/write, etc… conflicts!

Allow operations to be reordered as long as conflicts are ordered the same way.

Conflict Equivalence: Can reorder one schedule into another without reordering conflicts.

Conflict Serializability: Conflict Equivalent to a serial schedule.
Conflict Serializability

- **Step 1:** Serial Schedules are *Always Correct*
- **Step 2:** Schedules with the same operations and the same conflict ordering are *conflict-equivalent*.
- **Step 3:** Schedules *conflict-equivalent to* an always correct schedule are also correct.
  - … or *conflict serializable*
Example

Time

\[ T_1 \]

\[ W(B) \]

\[ R(B) \]

\[ W(A) \]

\[ R(A) \]

Conflict

\[ T_2 \]

\[ W(B) \]

\[ R(B) \]

\[ W(A) \]

\[ R(A) \]

\[ T_1 \]

\[ T_2 \]

VS.
Example

Time

1: T2 → T1
2: T1 → T2

≠

1: T2 → T1
2: T2 → T1
Equivalence

• Look at the actual effects
  • Can’t determine effects without running
• Look at the conflicts
  • Too strict
• Look at the possible effects
Example

Time

T1  T2  T3

R(A)
W(A)
W(A)
W(A)
Example

Write order irrelevant (T3 overwrites either way)
Information Flow
Information Flow

T1 → T2 → T3

Important

Not Important

R(…)

R(…)

R(…)

Important

Not Important
Information Flow

Multiple Transactions

R(…)

R(…)

R(…)
View Serializability

Possible Solution: Look at data flow!

View Equivalence: All reads read from the same writer
Final write in a batch comes from the same writer

View Serializability: View Equivalent to a serial schedule.
View Equivalence

- For all Reads \( R \)
  - If \( R \) reads old state in \( S_1 \), \( R \) reads old state in \( S_2 \)
  - If \( R \) reads Ti’s write in \( S_1 \), \( R \) reads the the same write in \( S_2 \)
- For all values \( V \) being written.
  - If \( W \) is the last write to \( V \) in \( S_1 \), \( W \) is the last write to \( V \) in \( S_2 \)
- If these conditions are satisfied, \( S_1 \) and \( S_2 \) are view-equivalent
View Serializability

- **Step 1:** Serial Schedules are *Always Correct*

- **Step 2:** Schedules with the same information flow are *view-equivalent*.

- **Step 3:** Schedules *view-equivalent* to an always correct schedule are also correct.

  - … or *view serializable*
Enforcing Serializability

• Conflict Serializability:
  • Does locking enforce conflict serializability?

• View Serializability
  • Is view serializability stronger, weaker, or incomparable to conflict serializability?

• What do we need to enforce either fully?
How to detect conflict serializable schedule?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(b)</td>
<td></td>
</tr>
<tr>
<td>W(d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(d)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(d)</td>
<td></td>
</tr>
</tbody>
</table>

Precedence Graph

- T1 → T2
- T2 → T3
- T3 → T1

Cycle!
Not Conflict serializable
Not conflict serializable but view serializable

Satisfies 3 conditions of view serializability

Every view serializable schedule which is not conflict serializable has blind writes.
How can conflicts be avoided?

- Optimistic Concurrency Control
- Conservative Concurrency Control
Conservative Concurrency Control

• How can bad schedules be detected?

• What problems does each approach introduce?

• How do we resolve these problems?
Two-Phase Locking

• Phase 1: Acquire (do not release) locks.
• Phase 2: Release (do not acquire) locks.

Why?

Can we do even better?
Example

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Acyclic - Conflict Serializable
2PL exists
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(a) W(a)</td>
<td>L(b) R(b)</td>
<td>L(d) R(d)</td>
</tr>
<tr>
<td>W(d) R-L(d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L(d) R-L(b)</td>
<td>L(b) R-L(a)</td>
<td>R(d) R-L(d)</td>
</tr>
<tr>
<td>W(b) R-L(b)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Need for shared and exclusive locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(a)</td>
<td></td>
<td>L(d)</td>
</tr>
<tr>
<td>W(a)</td>
<td></td>
<td>R(d)</td>
</tr>
<tr>
<td>L(b)</td>
<td>L(b)</td>
<td></td>
</tr>
<tr>
<td>W(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(d)</td>
</tr>
</tbody>
</table>

It is conflict Serializable but requires granular control of locks.

Precedence Graph:

It is serializable because it has an acyclic graph and 2PL because locks can be assigned as follows (many similar solutions are possible):

T1

T2

T3

Every non-serializable schedule can not be 2PL or strict 2PL.

E.

It can not be strict 2PL because T2 will have to unlock(B) at the very end and hence it will be impossible for T1 to w(B).
## Need for shared and exclusive locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SL(d)</td>
<td>R(d)</td>
<td></td>
</tr>
<tr>
<td>XL(a)</td>
<td>SL(d)</td>
<td>R(d)</td>
<td></td>
</tr>
<tr>
<td>W(a)</td>
<td>SL(b)</td>
<td>R-SL(b)</td>
<td></td>
</tr>
<tr>
<td>SL(b)</td>
<td>R(b)</td>
<td>R-SL(b)</td>
<td></td>
</tr>
<tr>
<td>XL(b)</td>
<td>W(b)</td>
<td>R-XL(b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(d)</td>
<td>R-SL(d)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R-XL(d)</td>
<td>XL(d)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(d)</td>
<td>R-XL(d)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lock held in mode</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Yes</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Reader/Writer (S/X)

• When accessing a DB Entity…
  • Table, Row, Column, Cell, etc…

• Before reading: Acquire a Shared (S) lock.
  • Any number of transactions can hold S.

• Before writing: Acquire an Exclusive (X) lock.
  • If a transaction holds an X, no other transaction can hold an S or X.
What do we lock?

Is it safe to allow some transactions to lock tables while other transactions to lock tuples?
Even within the same application, there may be a need for locks at multiple levels of granularity. Database elements are organized in a hierarchy:

- Relations
- Blocks
- Tuples

```
relations
blocks
  B1   B2   B3   B4
  t1   t2   t3   t4   t5
  |     |     |     contained in
```
Hierarchical Locks

- Lock Objects Top-Down
  - Before acquiring a lock on an object, an xact must have at least an intention lock on its parent!

- For example:
  - To acquire a S on an object, an xact must have an IS, IX on the object’s parent (why not S, SIX, or X?)
  - To acquire an X (or SIX) on an object, an xact must have a SIX, or IX on the object’s parent.
## New Lock Modes

### Lock Mode(s) Currently Held By Other Xacts

<table>
<thead>
<tr>
<th>Lock Mode Desired</th>
<th>None</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>IS</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
</tr>
<tr>
<td>IX</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>S</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
<td>valid</td>
<td>fail</td>
</tr>
<tr>
<td>X</td>
<td>valid</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
</tr>
</tbody>
</table>
Example

• An I lock for a super-element constrains the locks that the same transaction can obtain at a subelement.

• If Ti has locked the parent element P in IS, then Ti can lock child element C in IS, S.

• If Ti has locked the parent element P in IX, then Ti can lock child element C in IS, S, IX, X.
Example

• T1 wants exclusive lock on tuple t2
Example

• T2 wants to request an X lock on tuple t3
Example

T2 wants to request an S lock on block B2

T1(IX)  R1  T2(IS)

B1

T1(IX)  B2  T2(S)  not granted!

B2

T1(X)  t2  t3  t4  t5

B3

B4
Deadlocks

- **Deadlock**: A cycle of transactions waiting on each other’s locks
  - Problem in 2PL; xact can’t release a lock until it completes
- **How do we handle deadlocks?**
  - **Anticipate**: Prevent deadlocks before they happen.
  - **Detect**: Identify deadlock situations and abort one of the deadlocked xacts.
Deadlock Detection

- **Baseline**: If a lock request can not be satisfied, the transaction is blocked and must wait until the resource is available.

- Create a waits-for graph:
  - Nodes are transactions
  - Edge from $T_i$ to $T_k$ if $T_i$ is waiting for $T_k$ to release a lock.
  - Periodically check for cycles in the graph.
## Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(B)</td>
<td></td>
<td>X(B)</td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>S(C)</td>
<td>R(C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X(C)</td>
<td></td>
<td>X(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X(A)</td>
<td></td>
</tr>
</tbody>
</table>
Example

Time \( T_1 \)  \( T_2 \)  \( T_3 \)  \( T_4 \)

\[ \begin{align*}
S(A) & \quad R(A) \\
X(B) & \quad W(B) \\
S(B) & \\
X(C) & \quad S(C) \quad R(C) \\
X(A) & \quad X(B) \\
\end{align*} \]
How do we avoid deadlock?

- Avoid Deadlock Situations
- React to Deadlock Situations
Deadlock Prevention

• Ensure that dependencies are monotonic (and consequently acyclic)

• Assign each transaction a priority based on the timestamp at which it starts.

• When a transaction fails to acquire a lock:
  • Wait if monotonicity would be preserved.
  • Kill one transaction otherwise.
Deadlock Prevention

- Policy 1 (Wait-Die): If $T_i$ has a higher priority, wait for $T_k$, otherwise $T_i$ aborts.

- Policy 2 (Wait-Wound): If $T_i$ has a higher priority, $T_k$ aborts, otherwise $T_i$ waits.

- Protect fairness by restarting the aborted transaction with its original timestamp.