Isolation via Concurrency Control

Immanuel Trummer
itrummer@cornell.edu
www.itrummer.org
Database Management Systems (DBMS)

Application 1 ↔ DBMS Interface ↔ Application 2

Connections, Security, Utilities, ...

Query Processor
  - Query Parser
  - Query Rewriter
  - Query Optimizer
  - Query Executor

Storage Manager
  - Data Access
  - Buffer Manager
  - Transaction Manager
  - Recovery Manager

[RG, Sec. 19]
Reminder: Isolation

- Make users think that transactions execute **sequentially**
- That's challenging because in reality they **don't**...
Why Interleave Steps?

- Motivation 1: long running transactions
  - Imagine user submitting very short transaction
  - User may have long wait if scheduled behind long transaction
  - Better: alternate between transaction steps
  - Long transaction barely slower, short transaction quick
- Motivation 2: idle time e.g. by disk access
  - Assume transaction 1 needs data from disk for next step
  - Could load data while executing step from other transaction
Notation

• We introduce a short notation for transactions steps

• Will use letters (A, B, C, ...) for objects read or written

• Will use numbers to distinguish transactions

• Will use R for reads, W for writes, RW for reads+writes
  • E.g., R1(A) means transaction 1 reads object A

• Will use C for commits and A for aborts
  • E.g., C2 means transaction 2 commits
Example Transaction 1

UPDATE Accounts
SET Amount = Amount - 50
WHERE Name = 'Bob'

UPDATE Accounts
SET Amount = Amount + 50
WHERE Name = 'Alice'

(Transfers money from Bob to Alice)
Example Transaction 2

UPDATE Accounts
SET Amount = Amount * 1.1
WHERE Name = 'Bob'

UPDATE Accounts
SET Amount = Amount * 1.1
WHERE Name = 'Alice'

(Yearly bonus for everyone)
Example Schedule

UPDATE Accounts
SET Amount = Amount * 1.1
WHERE Name = 'Bob'

UPDATE Accounts
SET Amount = Amount - 50
WHERE Name = 'Bob'

UPDATE Accounts
SET Amount = Amount + 50
WHERE Name = 'Alice'

UPDATE Accounts
SET Amount = Amount * 1.1
WHERE Name = 'Alice'
Do We Have Isolation?
Do We Have Isolation?

- Assume Alice and Bob both have $100 initially
- Two possible transaction orders if executing sequentially
- T1 (transfer), T2 (bonus): Bob has $55, Alice $165 finally
- T2 (bonus), T1 (transfer): Bob has $60, Alice $160 finally
- Interleaving as shown: Bob has $60, Alice $165 finally
- **Destroys** the illusion of sequential execution!
Isolation Anomalies

- **Anomaly**: may destroy illusion of sequential execution
- **Dirty reads**: read data from unfinished transaction
- **Unrepeatable reads**: data changes while working with it
- **Lost updates**: unsaved changes are overridden
Dirty Reads

• We read data written by uncommitted transaction

• E.g., what if writing transaction aborts?
  • Need to undo all effects of aborted transaction

• Strange effects even if writing transaction commits

• Anomaly signature with short notation: Wx(A) Ry(A)
Unrepeatable Reads

- **Reading committed** data may be problematic, too
- We read data **twice**, changed from outside in between
- Means we read **different values** without changing value
  - E.g., **check** if at least one item stored (read 1), proceed
  - Other transaction **reduces** item count to zero
  - Now try to reduce item count by one (read 2 & write)
- Anomaly signature in short notation: $Rx(A) \ Wy(A) \ Cy \ Rx(A)$
Lost Updates

- We **override** value written by ongoing transaction

- E.g., want to pay **same** salary for all employees

- Have **two transactions** updating salary to different values

- **Constraint holds** if transactions execute sequentially

- But may not hold if **interleaving** transactions

- Anomaly signature in short notation: $W_x(A) \ W_y(A)$
(Phantom Problem)

- Read is unrepeateable because rows were inserted
  - E.g., we query twice for rows satisfying a predicate
  - Another transaction inserts new rows in between
- Problem is not related to an update but to insertion
- Therefore difficult to represent with current notation
- Will come back to this anomaly later ...
# SQL Isolation Levels

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read committed</td>
<td>Impossible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

Slides by Immanuel Trummer, Cornell University
Isolation in Postgres

- Setting default isolation level for future transactions:
  - Set session characteristics as <isolation-spec>

- Setting isolation level for the current transaction:
  - Set transaction <isolation-spec>

- <isolation-spec> ::= isolation level <i-level>

- <i-level> is one of SERIALIZABLE, REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED
Isolation in Postgres

• Setting default isolation level for future transactions:
  • Set session characteristics as <isolation-spec>

• Setting isolation level for the current transaction:
  • Set transaction <isolation-spec>

• <isolation-spec> ::= isolation level <i-level>

• <i-level> is one of SERIALIZABLE, REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED

Careful, default is READ COMMITTED!
Isolation in Postgres

• Setting default isolation level for future transactions:
  • Set session characteristics as <isolation-spec>

• Setting isolation level for the current transaction:
  • Set transaction <isolation-spec>

• <isolation-spec> ::= isolation level <i-level>

• <i-level> is one of SERIALIZABLE, REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED

Careful, default is READ COMMITTED!
Concurrency Control

Transactions

Concurrency Control

Schedule
(Ordered Transaction Steps)

Picks cheapest schedule among good ones
Selecting Schedules

- **Schedule**: ordered steps from multiple transactions
- A **good schedule** preserves the illusion of isolation
  - E.g., none of aforementioned anomalies
- Want to select **cheapest** schedule among good ones
- However, want to minimize selection **overheads**
- Need **sufficient** "goodness" criterion, **quick** to verify
Comparing Schedules

- Will define *good* schedules by comparison with reference:
  - **Serial** schedule (one transaction after the other)

- Introduce multiple *equivalence* schedule criteria next
  - **Final state** equivalence
  - **View** equivalence
  - **Conflict** equivalence
Final State Equivalence

- **Compare** two schedules based on final database state
- Equivalent schedules if DB **content** equal after execution
- Must hold for arbitrary **initial** database content
- E.g., the following two schedules are equivalent
  - $W_1(A)\ W_2(A)\ W_1(B)\ W_2(B)\ C_1\ C_2$
  - $W_1(A)\ W_1(B)\ C_1\ W_2(A)\ W_2(B)\ C_2$
Final State Serializability

- A schedule S is **final state serializable** if
  - There is a **serial** schedule ...
    - ... that is final state **equivalent** to S.
  - May have **unrepeatable reads** with final state serializability
    - Can be bad even if it does not influence db state
      - E.g., R1(A) W2(A) R1(A) is final state serializable
  - Probably want a **stronger** criterion!
View Equivalence

• **View equivalence** is stronger than final state equivalence

• Two schedules S1 and S2 are view equivalent iff

  • If transaction X reads the *initial value* for some object in S1, it also does so in S2

  • If transaction X reads a *value written* by transaction Y in S1, it also does so in S2

  • If transaction X writes the *final value* written by transaction Y in S1, it also does so in S2
View Serializability

- Schedule is view serializable if view equivalent to a **serial** schedule

- E.g., consider schedule R1(A) W2(A) R1(A) C1 C2
  - R1(A) R1(A) C1 W2(A) C2 - not view equivalent as **second read** now returns initial value
  - W2(A) C2 R1(A) R1(A) C1 - not view equivalent as **first read** does not return initial value
  - **Not equivalent** to any of two possible serial schedules

- Verifying view serializability is **NP-hard**! Too much overhead ...
Overview of Classes of Schedules

- **Final State Serializable**
- **View Serializable**
- **Serial**

All Schedules
Overview of Classes of Schedules

- Final State Serializable
- View Serializable
- Serial Slow Execution!

All Schedules
Overview of Classes of Schedules

- Final State
- Serializable

- Slow Execution!
- Concurrency Control!
- View
- Serializable
Overview of Classes of Schedules

- **Serial**: View Serializable
- **Concurrency Control!**: Final State Serializable, Anomalies!
- **Slow Execution!**: Serial Serializable
Overview of Classes of Schedules

Anomalies!
All Schedules
Overview of Classes of Schedules

- Serial
- View
- Serializable
- Final State
- Anomalies!

- Slow
- Concurrency
- Control!
- View
- Serializable

- Serial
- Slow
- Execution!

Anomalies! All Schedules

Need Something Else ...
Conflict Equivalence

- Two operations of different transactions on the same object conflict if at least one of them is a write
  - No problem as long as transactions only read data
  - Three possible conflict types: RW, WR, and WW
- Swapping conflicting operations changes results/view
- Users do not notice swaps between non-conflicting ops
- Condition for schedules S1 and S2 being conflict-equivalent:
  - Can get from S1 to S2 by swapping non-conflicting operations
Conflict Serializability

• Conflict serializable: conflict equivalent to serial schedule

• Can test efficiently if schedule is conflict serializable
  
  • Draw conflict graph (see next)
  
  • Test if conflict graph has cycle
  
  • Conflict serializable if no cycle
Conflict Graph

- Draw conflict graph for schedule to test **serializability**
- Add one graph **node for each transaction** in schedule
- For each pair of **conflicting operations** O1 and O2
  - **Draw edge** from O1 transaction to O2 transaction
Conflict Graph Example

R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)
Conflict Graph Example

R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)
Conflicts Graph Semantics

- Semantics of having edge from node i to j:
  - Any conflict-equivalent schedule must order i before j

- Getting equivalent serial schedule for acyclic graph:
  - Start with node (transaction) without incoming edges
  - Add all operations of that transaction and commit
  - Continue with node where all predecessors treated
  - ...

Slides by Immanuel Trummer, Cornell University
Conflict Graph Example

R1(A) R2(A) R1(C) W1(A) C1 R3(C) W2(B) C2 W3(B) W3(C) C3

Equivalent Serial schedule
Conflict Graph Example

R1(A) R2(A) R1(C) W1(A) C1 R3(C) W2(B) C2 W3(B) W3(C) C3

Equivalent Serial schedule
R2(A) W2(B) C2
Conflict Graph Example

R1(A) R2(A) R1(C) W1(A) C1 R3(C) W2(B) C2 W3(B) W3(C) C3

Equivalent Serial schedule
R2(A) W2(B) C2 R1(A) R1(C) W1(A) C1
Conflict Graph Example

R1(A) R2(A) R1(C) W1(A) C1 R3(C) W2(B) C2 W3(B) W3(C) C3

Equivalent Serial schedule
R2(A) W2(B) C2 R1(A) R1(C) W1(A) C1 R3(C) W3(B) W3(C) C3
Overview of Classes of Schedules

- Final State
  - Serializable
  - Anomalies!

- Slow
  - Concurrency Control!
  - View
  - Serializable
  - Anomalies!

- Disallows some Good schedules

- Serial Slow Execution!
Handling Aborts

- **Exclude** aborted transactions for checking serializability
  - DBMS acts as if aborted transactions *never* happened
- Orthogonal **classification** of schedules based on aborts
Recoverable Schedules

• A schedule is **recoverable** if this condition holds:
  
  • Transaction commits only **after** all transactions it read from have committed as well

• Example for **non-recoverable** schedule:
  
  • \text{W1(A) R2(A) W2(B) C2 A1}

  • No trace of **aborted** transactions should remain

  • But write to B may have been **influenced** by read from A
ACA Schedules

- Can make schedule recoverable by delaying commits
- But still may have chain of aborting transactions
  - Transaction read from aborted transaction - tainted!
- ACA schedule: no transaction reads uncommitted data
  - ACA = Avoiding Cascading Aborts
- E.g., recoverable but does not avoid cascading aborts:
  - W1(A) R2(A) W2(B) C1 C2
Strict Schedules

- Definition of strict schedules:
  - No transaction reads or writes uncommitted data
- Otherwise cleanup after aborts can get tricky
  - Need to keep track of different object versions
  - Must check for each object whether undo required
- E.g., $W1(A) \ W2(A) \ W3(A)$ not strict (ACA & recoverable)
Classifying Schedules by Abort-Related Restrictions

All Schedules

Recoverable

ACA

Strict
Schedule Properties

• **Serializability**
  - Final state serializable
  - Conflict serializable
  - View serializable

• **Aborts**
  - Recoverable
  - Avoids cascading aborts
  - Strict