Concurrency Control Without Locking

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

Application 1 ↔ DBMS Interface ↔ Application 2

DBMS Interface:
- Connections, Security, Utilities, ...
- Query Processor
  - Query Parser
  - Query Optimizer
- Query Rewriter
- Query Executor
- Storage Manager
  - Data Access
  - Buffer Manager
  - Transaction Manager
  - Recovery Manager

Data

[RG, Sec. 19.5]
Outlook

- **Optimistic** concurrency control
- **Timestamp** concurrency control
- **Multi-version** concurrency control
- **Snapshot** isolation
Optimistic CC Motivation

• Locking itself leads to **overheads**
  • E.g., overheads due to **lock management**
  • Possibly overheads due to **deadlocks**

• Locking prevents conflicts **proactively**
  • **Pessimistic** assumption: conflicts are likely

• **Optimistic** concurrency control
  • Conflicts are **rare**, no need to avoid proactively
Optimistic CC Bookkeeping

- Need to keep read set and write set for each transaction
  - **Read set**: objects that the transaction read
  - **Write set**: objects that the transaction wrote
Execution Phases

- **Read**
  - *Read* relevant data from database
  - *Execute* transaction on private copy

- **Validate**
  - *Check* for conflicts with other transactions

- **Write**
  - *Publish* local changes if no conflicts
Validation Phase

• Assign transactions to unique *timestamps* at validation
  
  • Will try to *serialize* transactions in timestamp order

• Two transactions *cannot* have conflicted if
  
  • T1 completes *before* T2

  • T1 completes before T2 starts *writing*, Writes(T1) *disjunct* with Reads(T2)

  • T1 completes reads before T2 completes *reads*, Writes(T1) *disjunct* with Reads(T2) and Writes(T2)
Simplification: Combine Validation and Write Phase

• Only one transaction can be in validation+write phase

• Only need to consider conflict cases 1 and 2
  • Write phases cannot overlap
Optimistic CC Overheads

- Must **record** read and write sets
- Transaction **restarts** if validation fails
- **Critical** section during validation/writes
Optimistic CC Overheads

- Must **record** read and write sets
- Transaction **restarts** if validation fails
- **Critical** section during validation/writes

*Good if probability of conflicts is low*
Timestamp CC Overview

- We associate transactions with timestamps
- Want to serialize transactions in timestamp order
- Also, we associate each object with timestamps
  - **Read timestamp**: time of last read
  - **Write timestamp**: time of last write
Timestamp CC Rules

• **TS(T)** is timestamp of transaction T

• **RTS(A), WTS(A)**: read & write timestamp of object A

• Transaction T wants to **read** database object A
  • Abort & restart if **TS(T) < WTS(A)**

• Transaction T wants to **write** database object A
  • Abort & restart if **TS(T) < RTS(A)**

• **What if TS(T) < WTS(A) ... ?**
Thomas Write Rule

- Transaction T wants to write A but $TS(T) < WTS(A)$

- **Conflicts** with serialization order, could abort

- **Thomas Write Rule** ignores outdated writes instead
  
  - E.g., consider $R1(A) \ W2(A) \ C2 \ W1(A) \ C1$
  
  - Not conflict serializable but view-serializable
  
  - Simplifies to $R1(A) \ C2 \ W1(A) \ C1$
Timestamp CC Overheads

- **Restarting** overheads for aborted transactions
- Need to keep track of **object timestamps**
  - Means **space** consumption increases
  - Overheads for **updating timestamps**
    - **Requires write** for each operation
Multi-version CC (MVCC) Overview

- Idea: keep multiple versions of database objects

- Doing so helps for instance in the following situation

- \( R1(A) \) \( W1(A) \) \( R2(A) \) \( W2(B) \) \( R1(B) \) \( W1(C) \)

- Not conflict-serializable as written

- Could fix by moving \( R1(B) \) before \( W2(B) \)

- Making \( R1(B) \) read old version of \( B \) has same effect
MVCC Protocol

• Each transaction receives **timestamp** when entering
  • Will try to **serialize** transactions in this order
• Each **write** creates a new version of an object
  • Perform **write check** and abort if not valid
  • Version has **timestamp** of writing transaction
• **Read** mapped to last version before transaction timestamp
  • Transaction with timestamp i reads version with **largest timestamp** k such that k<i
Write Check

- Want to be **consistent** with transaction timestamps

- Can transaction with timestamp I **write object A**?
  - Assume transaction with **timestamp > I**
  - Cannot read **earlier** version of A than I
  - Must **abort** if this has already happened

  - Track **read timestamps** for versions!
Abort-Related Behavior

- Aforementioned protocol guarantees *serializability*
- Need additional mechanisms for *abort* properties
- E.g., delay commits for *recoverability*
Snapshot Isolation Overview

- Each transaction operates on database snapshot
- This snapshot is taken once transaction starts
  - Uses last committed value for each object
- Maintains multiple object versions internally
  - Different from MVCC: no uncommitted values
Handling Writes

- Check before commit for *overlapping writes*
- Everything OK if target objects *unchanged*
- Otherwise *abort* & restart transaction
Example with SI

• Consider **tables A and B** with one integer column each

• Consider **two transactions** that execute one update each
  
  • T1: **Insert into B select count(*) from a;**
  
  • T2: **Insert into A select count(*) from b;**

• **What happens if both transaction start at same time?**

  • *Is the result equivalent to a serial execution?*
Write Skew

T1: Insert into B select count(*) from A;
T2: Insert into A select count(*) from B;

<table>
<thead>
<tr>
<th>Execution</th>
<th>Content of A</th>
<th>Content of B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1; T2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T2; T1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Snapshot Isolation</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Serializability vs. SQL Definition

- SQL-92 standard defines isolation via anomalies
- The write skew anomaly is missing, drawing criticism
- Careful, may get SI when choosing serializable isolation