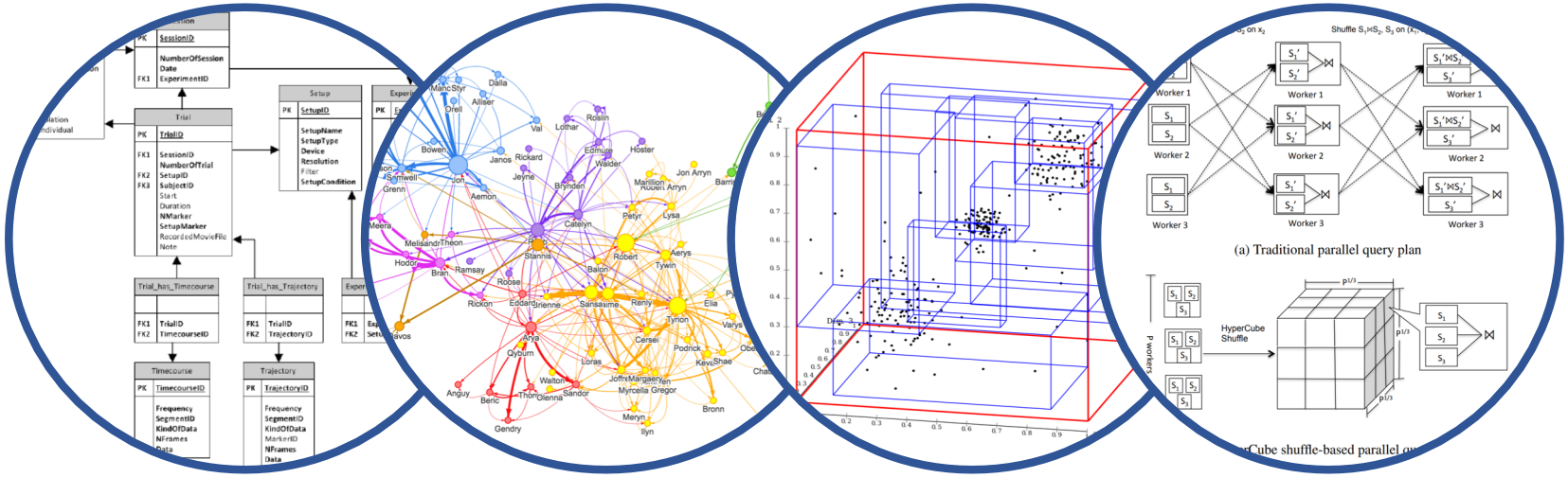


Introduction to Data Management

Isolation Levels

Jonathan Leang

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

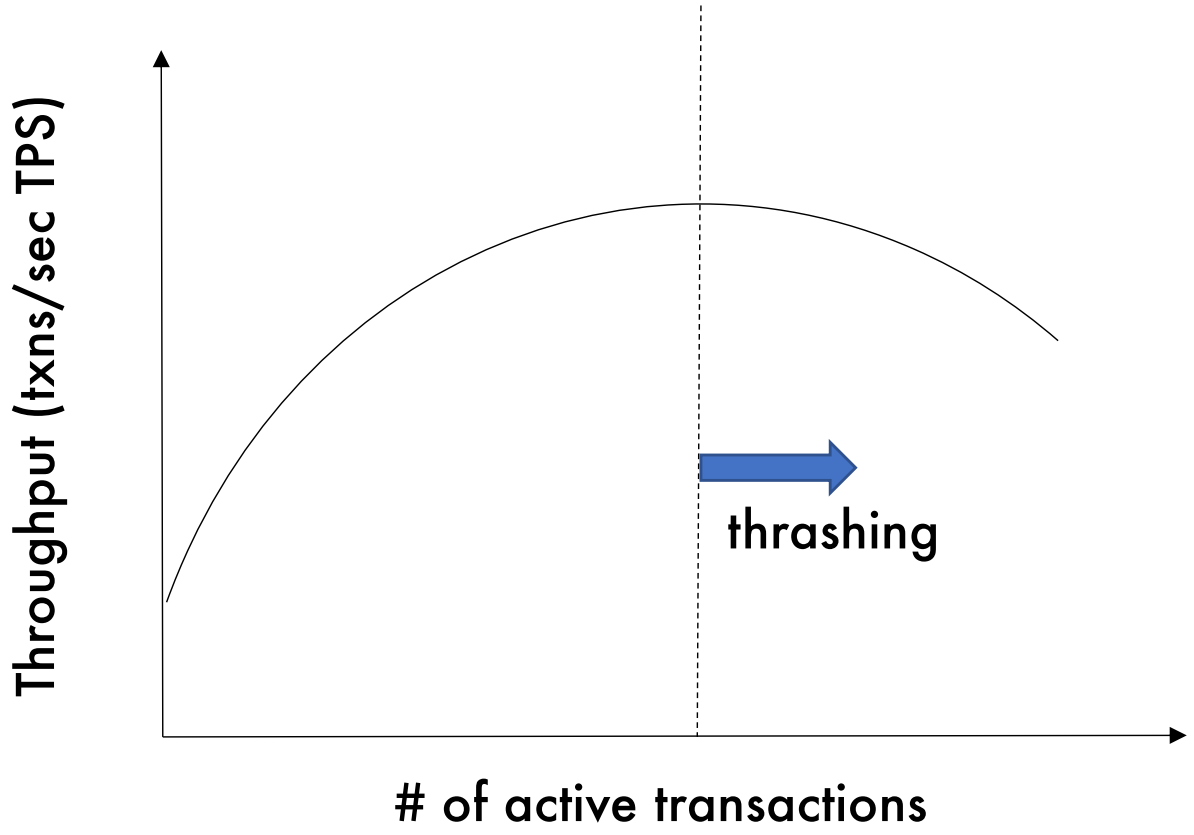
Outline

- Shared/Exclusive locks
- Isolation levels
- Implementing transactions in practice

Practicality of Binary Locks

- Binary Locks → full control or no control
- Leads to excessive deadlocking

Thrashing



Shared/Exclusive Locks

- Observation: Reads don't conflict with each other
- Simple 3-tier lock hierarchy:
  - Exclusive/Write Lock → $\mathbf{X_i(A)}$
    - Full control
    - No other locks may exist
  - Shared/Read Lock → $\mathbf{S_i(A)}$
    - Shared control
    - May exist with other shared locks
  - Unlocked

Shared/Exclusive Locks

Requested Lock	unlocked	S	X
S	Yes	Yes	No
X	Yes	No	No

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 UNCOMMITED**
  - **READ COMMITED**
  - **REPEATABLE READ**
  - **SERIALIZABLE**
  - SNAPSHOT ISOLATION
  - …
- Default isolation level and configurability depends on the DBMS (read the docs)

READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed
- Dirty reads are possible

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

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**

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

T1	T2
X(A) W(A)	
	S(A) blocked…
ABORT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

READ COMMITTED

- **Unrepeatable reads are possible**

T1	T2
X(A) blocked…	S(A)
	R(A)
…granted X(A)	U(A)
W(A)	S(A) blocked…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

REPEATABLE READ

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

T1	T2
X(A) blocked…	S(A)
	R(A)
…granted X(A)	U(A)
W(A)	S(A) blocked…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

T1	T2
X(A) blocked…	S(A)
	R(A)
	R(A)
…granted X(A)	COMMIT U(A)
W(A)	
COMMIT U(A)	

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;

SELECT * FROM Table;

T1	T2
R(A)	
R(B)	
	I(C)
R(A)	
R(B)	
R(C)	

INSERT INTO Table
VALUES (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

READ UNCOMMITED → Dirty Read
READ COMMITED → Unrepeatable Read
REPEATABLE READ → Phantom Read
SERIALIZABLE → No Anomalies

Applying Transaction Logic

- 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).

Transaction Setup

```
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
conn.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);
conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
conn.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
conn.setAutoCommit(true);
conn.setAutoCommit(false);
```

DB Transaction Programming in Java

```
try {
  // Each Instance hold a unique conn
  PreparedStatement q = conn.prepareStatement("SELECT …");
  PreparedStatement i = conn.prepareStatement("INSERT …");
  // Make sure the statements don't execute separately
  conn.setAutoCommit(false);
  conn.execute("BEGIN TRANSACTION;");
  ResultSet rs = q.executeQuery();
  while(rs.next()) { … } // Read out tuples from the ResultSet
  i.executeUpdate();
  conn.execute("COMMIT;");
  conn.setAutoCommit(true);
  return "success"
} catch (SQLException ex) {
  try {
    conn.execute("ROLLBACK;");
    conn.setAutoCommit(true);
    return "failed"
  } catch (SQLException e) {
    return "failed";
  }
}
```