Transactions

Carsten Schürmann

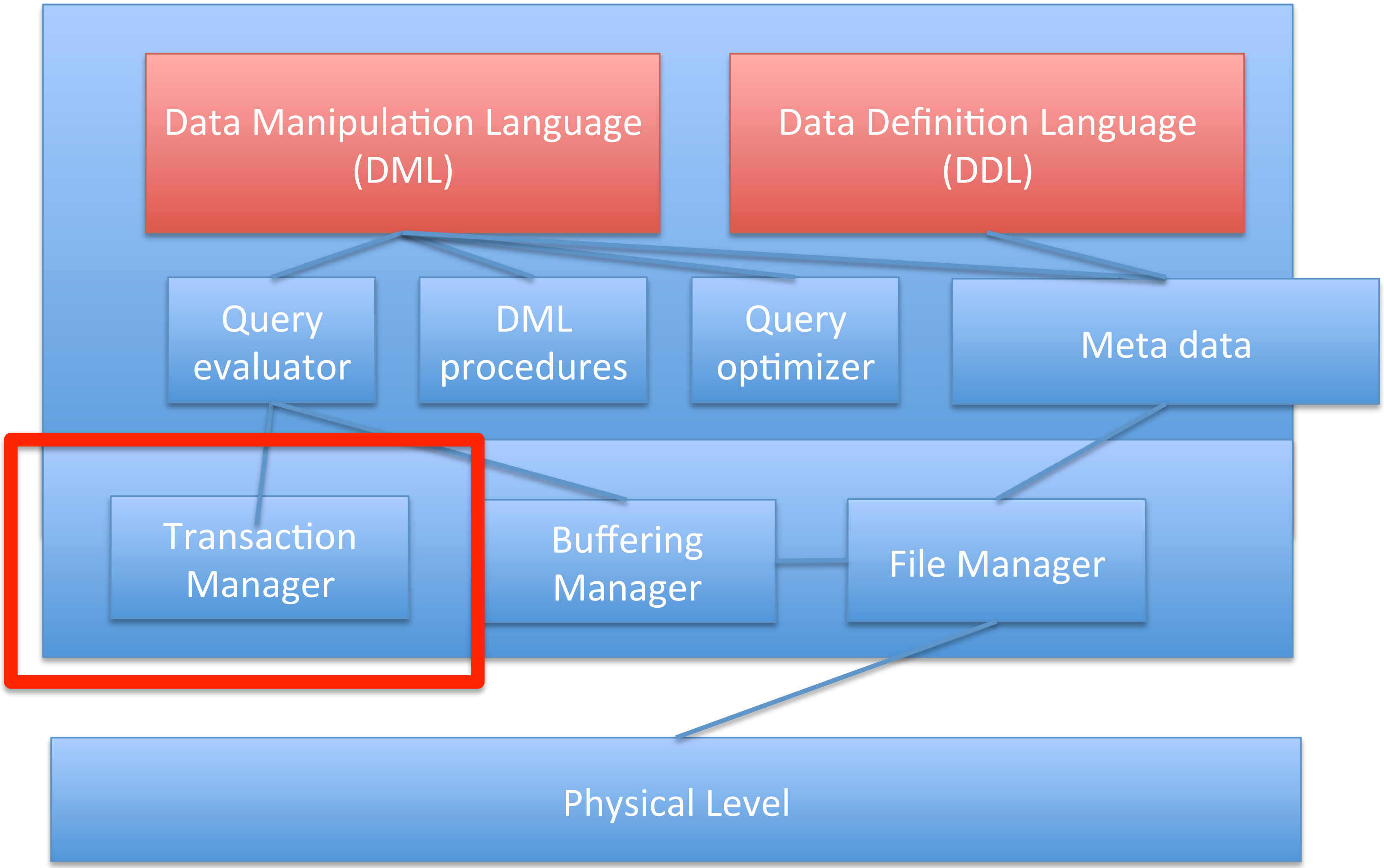
Motivation

- Tw update the same record (“Jesper”)
  - (“Jesper”, 10), (“Jesper”, 20)
  - Race condition!

- Transfer 100 k… …ount to another
  - po… …happens?

DBMSs automatically handle both issues

Database System Architecture



Transactions

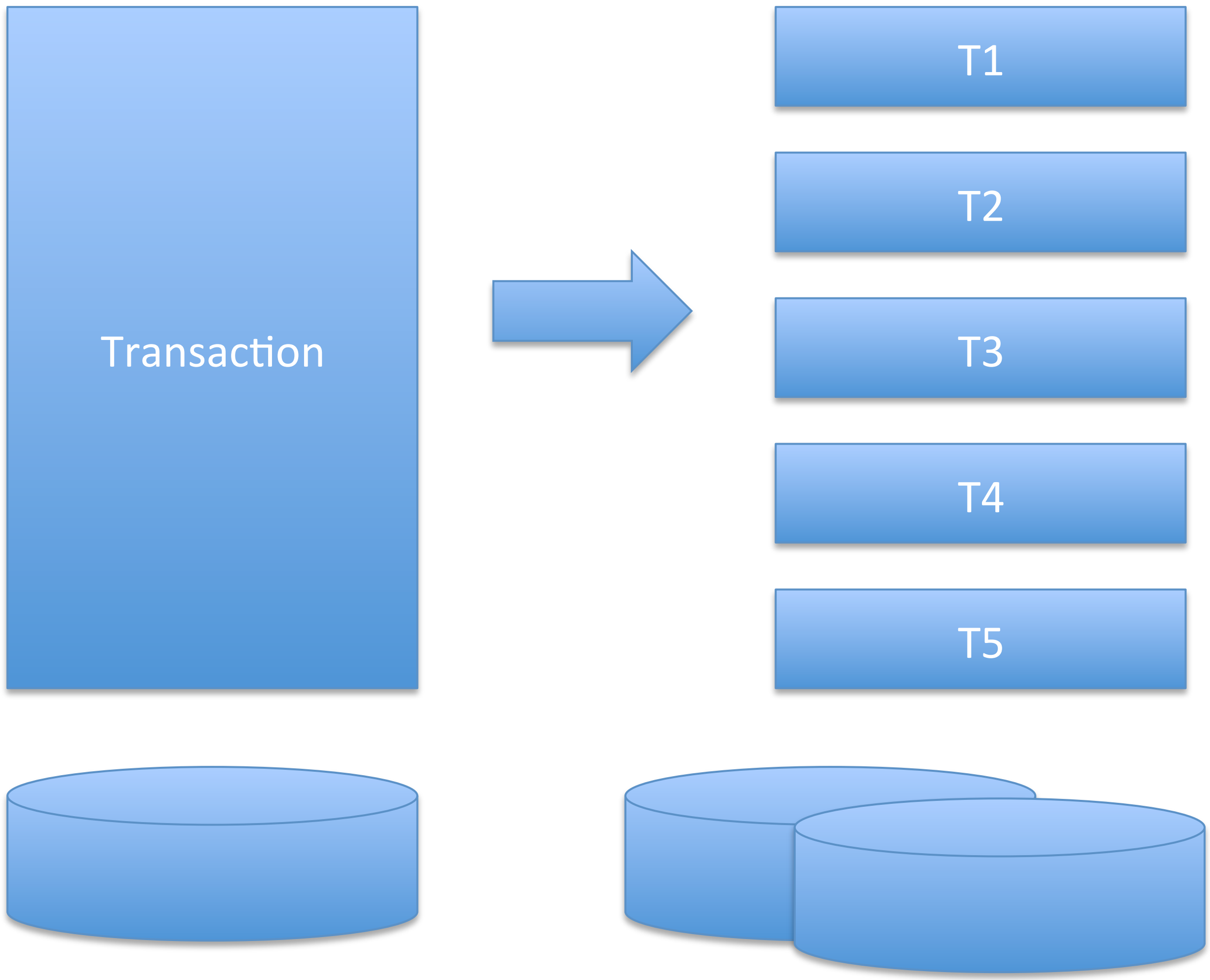
- Execution of one or more operations
- Implements a higher-level function
- May temporarily break consistency of database
- Transactions form unit
  - Either all steps are executed
  - Or no steps are executed

Transaction Example

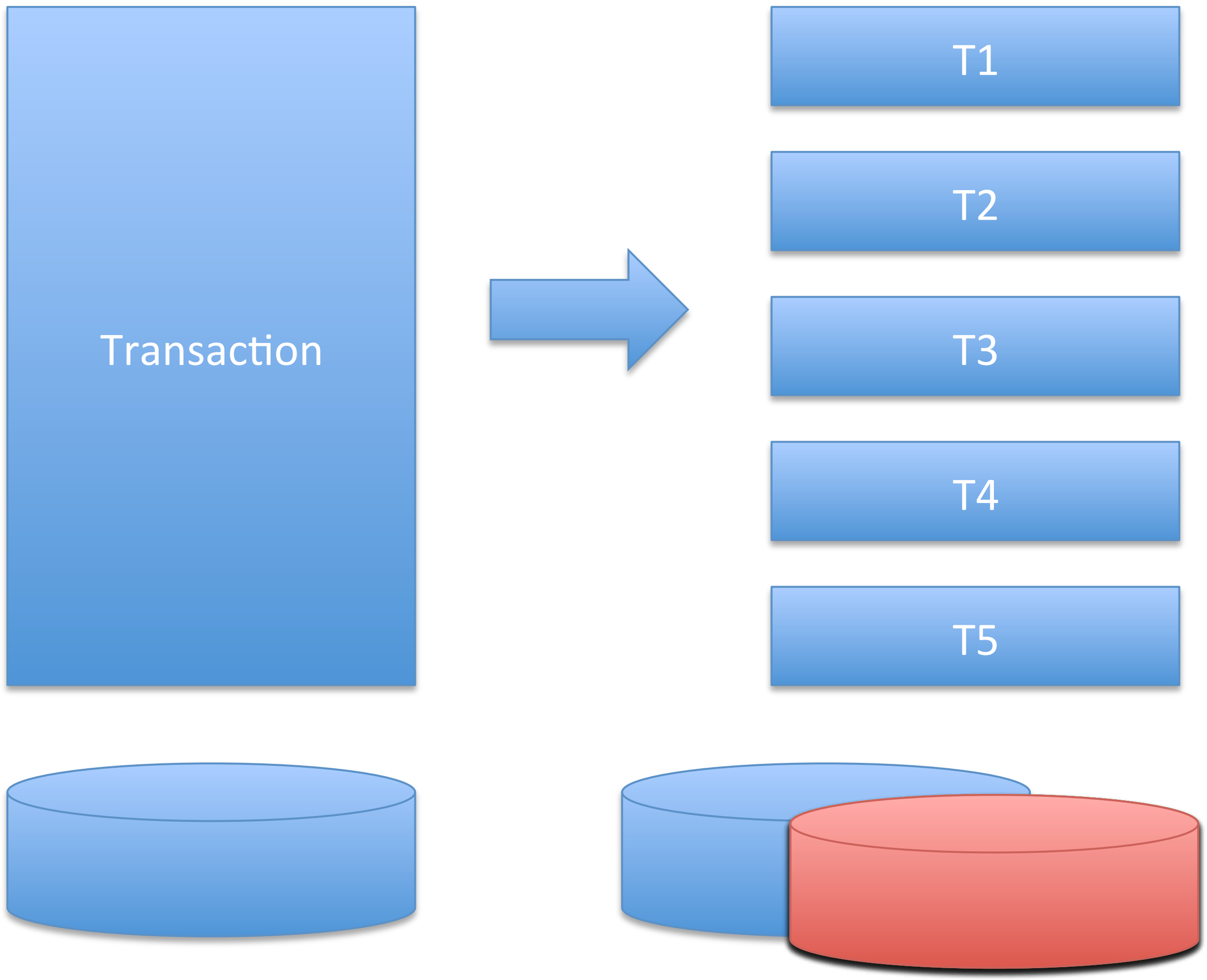
Move 1000 kr from Jesper’s account to mine!

1. Check whether Jesper’s got 1000 kr
2. Deduct 1000 kr from Jesper’s account
3. Deposit 1000 kr on Carsten’s account

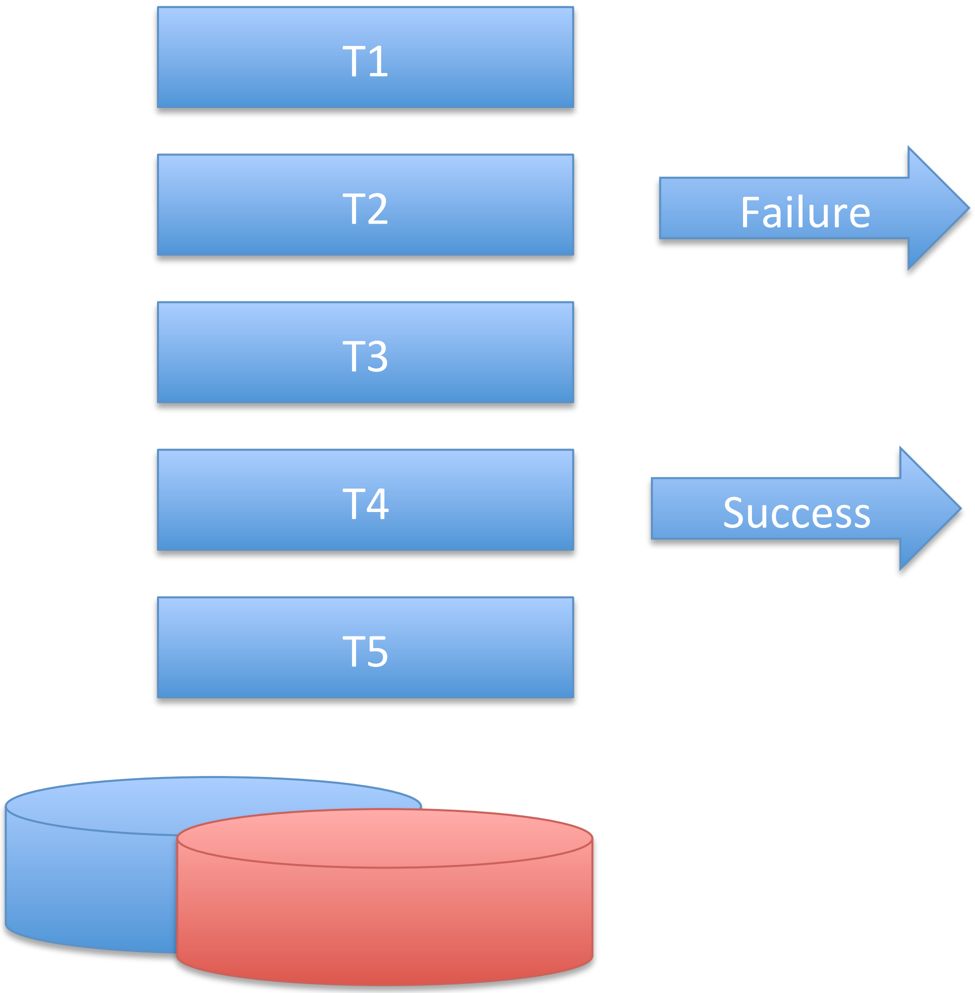
Perfect System



Perfect System



Perfect System



Thoughts

- Sequentiality is good
- Concurrency is better
- Parallel execution of independent transactions
- Interleaving of transactions
- Correctness
- Fairness
- Persistency

ACID

Atomicity: All actions in the defining the transaction happen, or none happen.

Consistency: If each transaction is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions

Durability: If a transaction finishes successfully, its effects persist.

ACID

Atomicity: *all or nothing*

Consistency: *it looks correct to me*

Isolation: *as if alone*

Durability: *survive failures*

Basic Database Operations

Database: Fixed set of names (A, B, C…)

- Read: R(A)
- Write: W(A)

Transaction commands

1. begin (start the transaction)
2. commit (make state persistent)
3. abort (roll state back to the beginning)

Atomicity

Assume first two step executes

Then power failure

Rewind the transactions

Recreate the database as it was before start

1. Check whether Jesper's got 1000 kr
2. Deduct 1000 kr from Jesper's account
3. Deposit 1000 kr on Carsten's account

Mechanisms: Logging!

(Audit trail and efficiency)

ACID

Atomicity: All actions in the defining the transaction happen, or none happen.

Consistency: If each transaction is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions

Durability: If a transaction finishes successfully, its effects persist.

Consistency

Data Consistency

Data in the DBMS is accurate in modeling the real world and follows *integrity constraints*!

Transaction Consistency

If the database is consistent before the transaction starts, it will be after also.

Level of Consistency

Strong Consistency

Guaranteed to see all writes immediately, but transactions will be slower

Weak Consistency

Writes will take time to become visible, but transactions will be faster

ACID

Atomicity: All actions in the defining the transaction happen, or none happen.

Consistency: If each transaction is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions

Durability: If a transaction finishes successfully, its effects persist.

Isolation

A transaction runs as if it was running by itself

Pessimistic Isolation Strategy

Don't let problems arise in the first place

Optimistic Isolation Strategy

Since conflicts are rare, deal with them as they arise

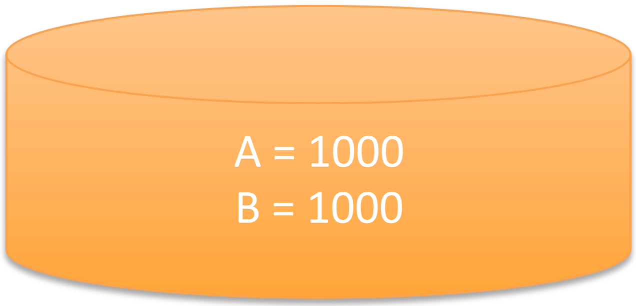
Invariant

A + B = 2120

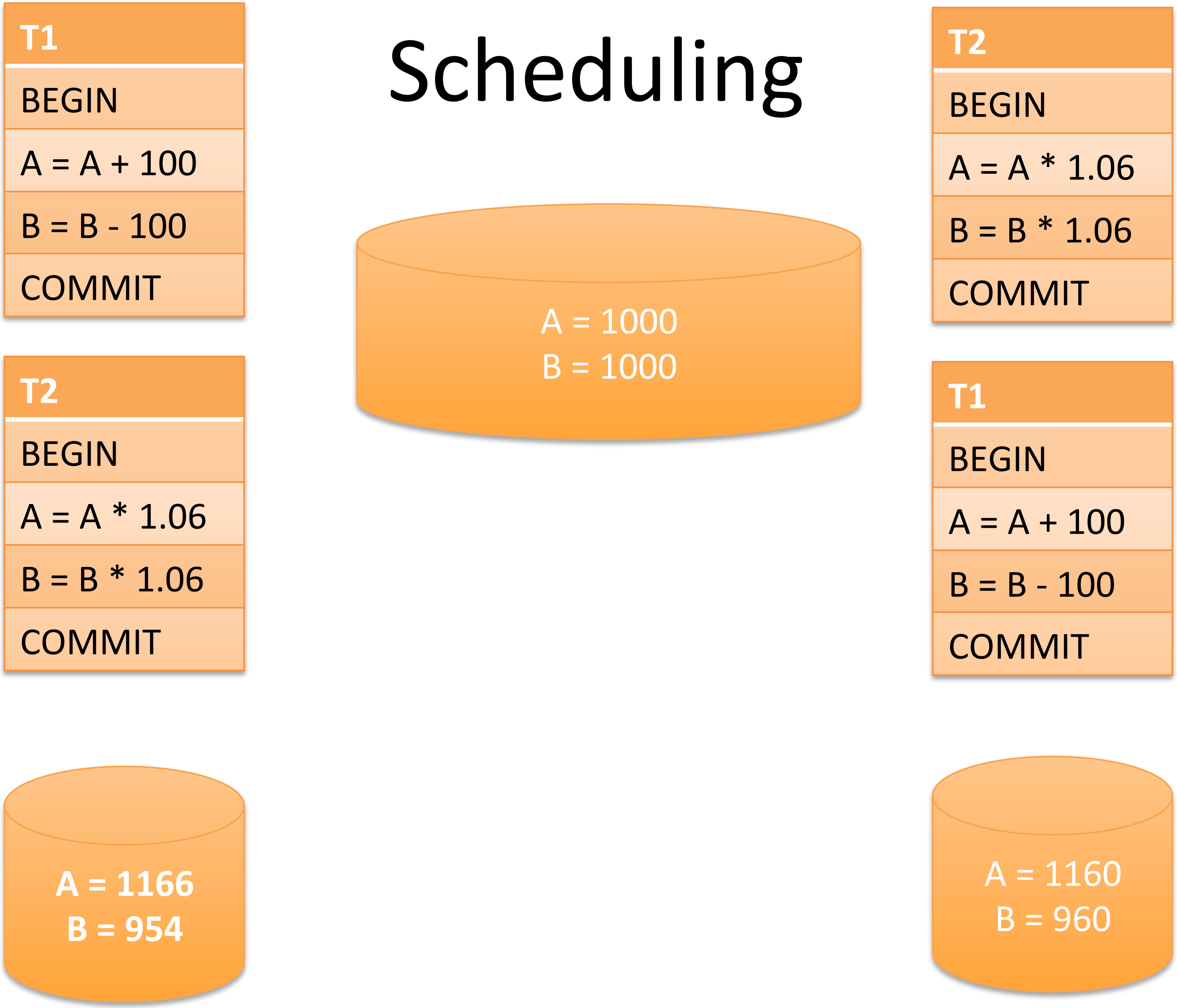
There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

The net effect must be equivalent to these two transactions running serially in some order.

T1	T2
BEGIN	BEGIN
A = A + 100	A = A * 1.06
B = B - 100	B = B * 1.06
COMMIT	COMMIT



Scheduling



T1
BEGIN
A = A + 100
B = B - 100
COMMIT
T2
BEGIN
A = A * 1.06
B = B * 1.06
COMMIT
A = 1000
B = 1000
T2
BEGIN
A = A * 1.06
B = B * 1.06
COMMIT
T1
BEGIN
A = A + 100
B = B - 100
COMMIT
A = 1166
B = 954
A = 1160
B = 960

Schedule

T1	T2
BEGIN	
A = A + 100	
B = B - 100	
COMMIT	
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT

Schedule

T1	T2
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT
BEGIN	
A = A + 100	
B = B - 100	
COMMIT	

Interleaving Schedule (Good)

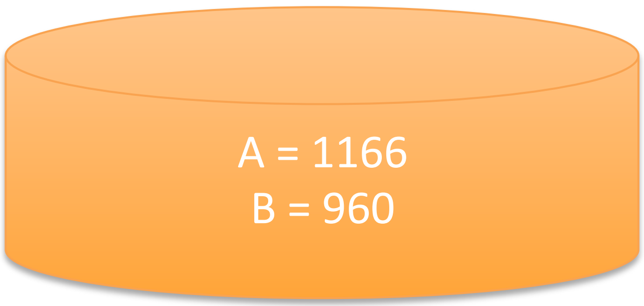
T1	T2
BEGIN	
A = A + 100	
	BEGIN
	A = A * 1.06
B = B - 100	
COMMIT	
	B = B * 1.06
	COMMIT

≡

T1	T2
BEGIN	
A = A + 100	
B = B - 100	
COMMIT	
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT

Interleaving Schedule (Bad)

T1	T2
BEGIN	
A = A + 100	
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT
B = B - 100	
COMMIT	



Someone lost 6kr.

Correctness

Serial schedule

A schedule that does not interleave the actions of different transactions

Equivalent schedule $S1 \equiv S2$

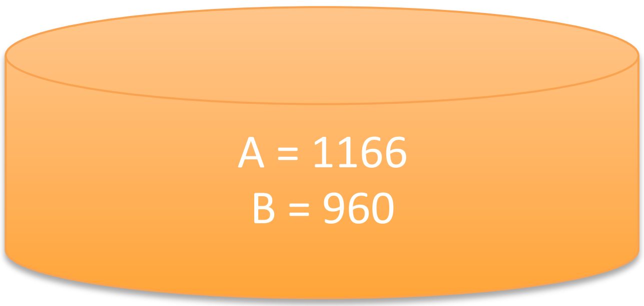
For any database state, the effect of executing the schedule S1 is identical to the effect of executing the schedule S2

Serializability

A schedule is *serializable* if an only if it is equivalent to a serial schedule.

Interleaving Schedule (Bad)

T1	T2
BEGIN	
A = A + 100	
	BEGIN
	A = A * 1.06
	B = B * 1.06
	COMMIT
B = B - 100	
COMMIT	



T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Interleaved Execution Anomalies

- **Read-Write** conflicts (R-W)
- **Write-Read** conflicts (W-R)
- **Write-Write** conflicts (W-W)

- Note, no **Read-Read** conflicts

Write Read Conflicts

Reading Uncommitted Data

“Dirty Reads”

	T1	T2	
	BEGIN		
50kr	R(A)		
100kr	W(A)		
		BEGIN	
		R(A)	100kr
		W(A)	
		COMMIT	
	R(B)		
	W(B)		
	ABORT		

Read Write Conflicts

"Unrepeatable reads"

	T1	T2	
	BEGIN		
50kr	R(A)		
		BEGIN	
		R(A)	50kr
		W(A)	100kr
		COMMIT	
100kr	R(A)		
	COMMIT		

Write Write Conflicts

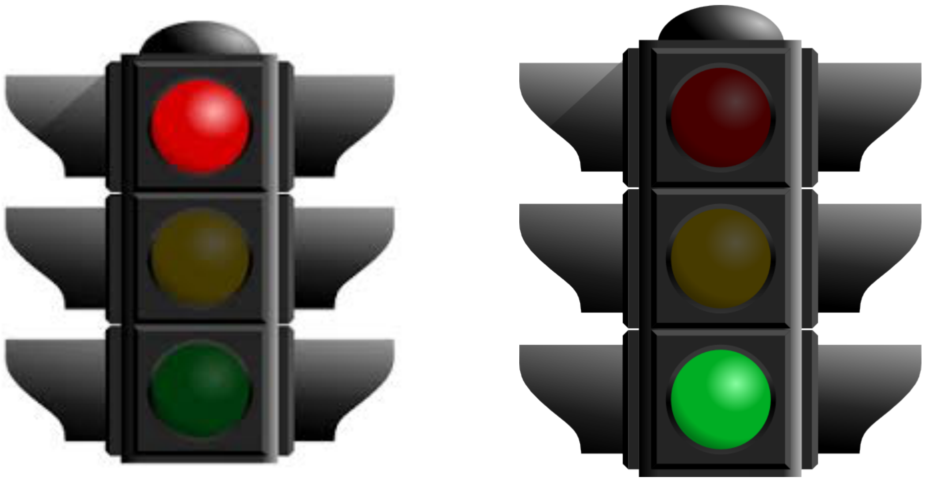
Overwriting uncommitted data

	T1	T2	
	BEGIN		
50kr	W(A)		
		BEGIN	
		W(A)	590kr
		W(B)	Jesper
		COMMIT	
Mette	W(B)		
	COMMIT		

Locks

Locks

- Protects a resource
  - Row in a table
  - Table in a database
- Operations
  - Lock
  - Unlock



Execution without Locks

T1	T2
BEGIN	
R(A)	
	BEGIN
	R(A)
W(B)	
COMMIT	
	W(B)
	COMMIT

Execution with Locks

T1	T2
BEGIN	
LOCK(A)	
R(A)	
	BEGIN
	LOCK(A)
W(A)	
UNLOCK(A)	
COMMIT	
	R(A)
	W(A)
	UNLOCK(A)
	COMMIT

Lock Manager
Granted (T1 → A)
Denied
Released (T1 → A)
Granted (T2 → A)
Released (T2 → A)

Types of Locks

- S-Lock   Shared lock (reads)
- X-Lock   Exclusive lock (writes)

Compatibility matrix

		Lock requested	
Lock granted		S	X
	S	Yes	No
	X	No	No

Execution with Locks

- Transactions request locks (or upgrades)
- Lock manager grants or blocks requests
- Transactions release locks
- Lock manager updates lock-table

But ....

Execution with Locks

Problem with Isolation!

T1	T2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager
Granted (T1 $\rightarrow$ A, X)
Released (T1 $\rightarrow$ A)
Granted (T2 $\rightarrow$ A, X)
Released (T2 $\rightarrow$ A)
Granted (T1 $\rightarrow$ A, S)
Released (T1 $\rightarrow$ A)

Concurrency Control

Locks can help but they need to be requested and released sensibly!

Two Phase Locking (2PL)

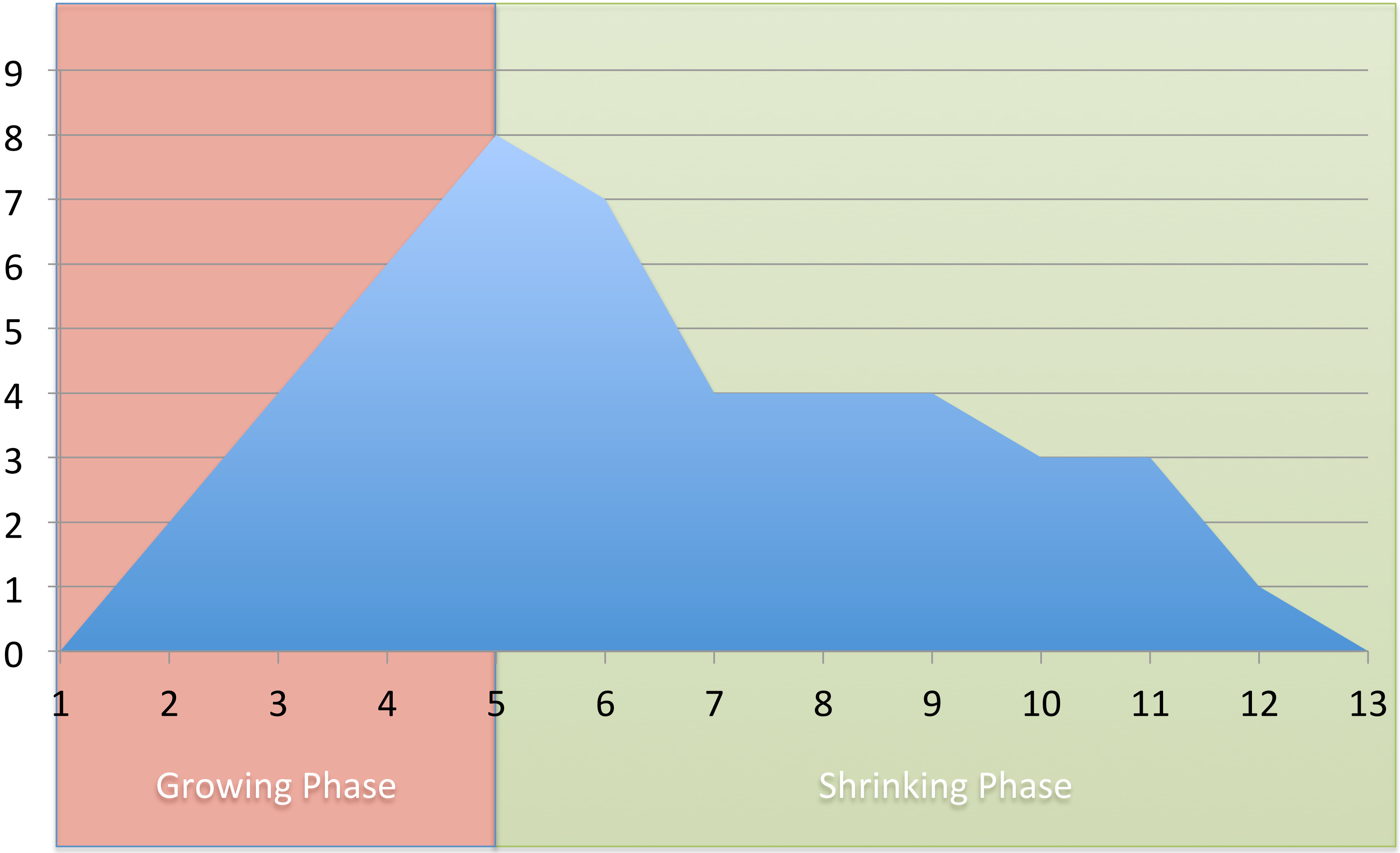
Phase 1: Growing

- Transaction requests the locks that it needs from the DBMS's lock manager.
- The lock manager grants/denies lock requests

Phase 2: Shrinking

- The transaction is allowed to only release locks that it previously acquired. It cannot acquire new locks.

Transaction Lifetime



Violates Two Phase Locking

T1	T2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager
Granted (T1 → A, X)
Released (T1 → A)
Granted (T2 → A, X)
Released (T2 → A)
Granted (T1 → A, S)
Released (T1 → A)

Violates Two Phase Locking

T1	T2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

Lock Manager
Granted (T1 → A, X)
Denied
Released (T1 → A)
Granted (T2 → A, X)
Released (T2 → A)

Two Phase Locking

- There are schedules that are serializable but would not be allowed by 2PL
- Locking limits concurrency
- May lead to deadlocks
- May still have "dirty reads"
- Solution: Strict 2PL

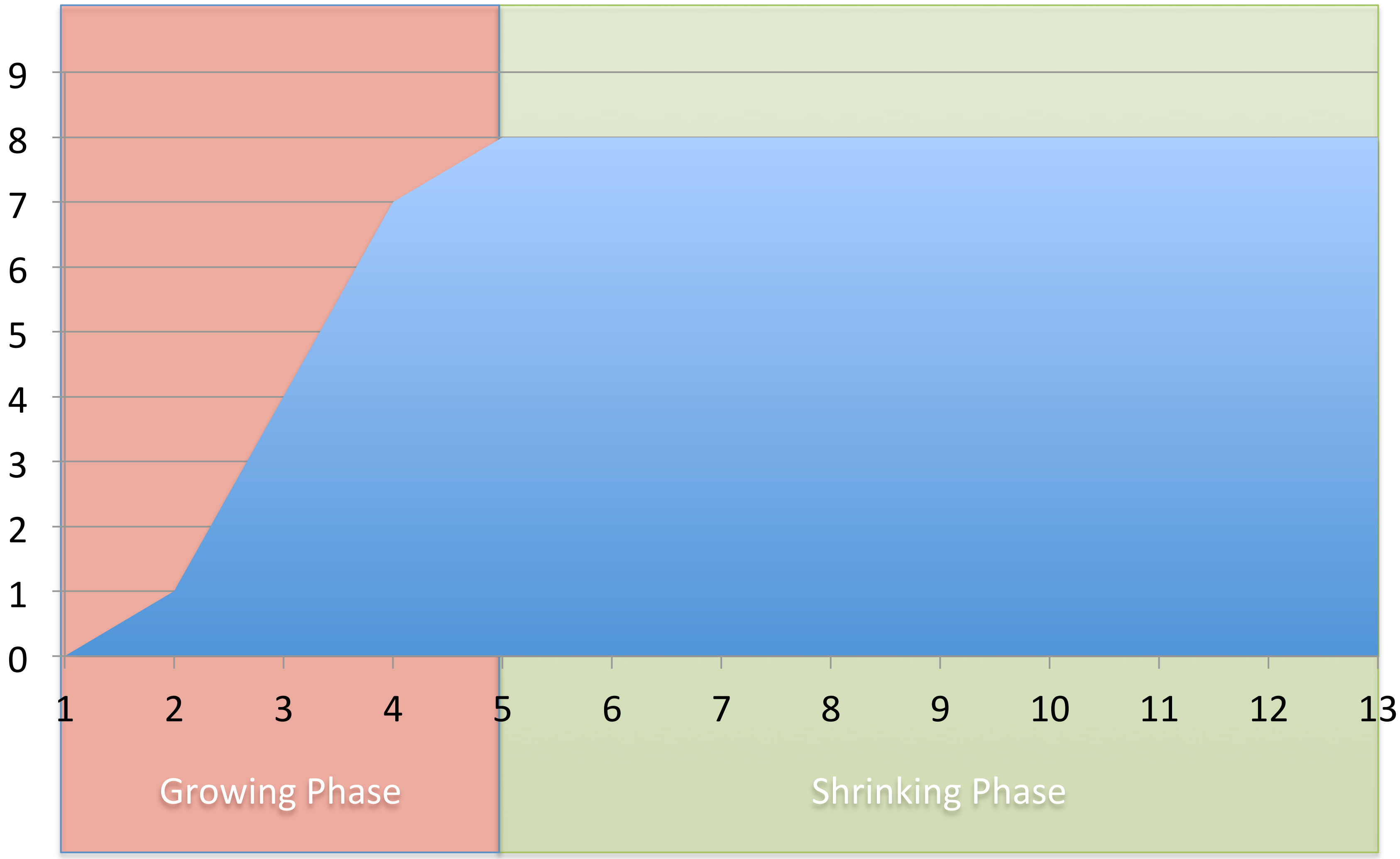
Strict Two Phase Locking

A schedule is *strict* if a value written by a transaction is not read or overwritten by other transaction until that transaction finishes.

Advantages:

- Recoverable
- Do not require cascading aborts
- Aborted transactions can be roll-backed by just restoring original values of modified tuples.

Transaction Lifetime



Strict Two Phase Locking

Transactions hold all of their locks until commit.

Good:

- Avoids “dirty reads” etc

Bad:

- Limits concurrency even more
- And still may lead to deadlocks

Locking in Practice

- No need to set locks manually
- But DBMS may require hints to help improve concurrency

ACID

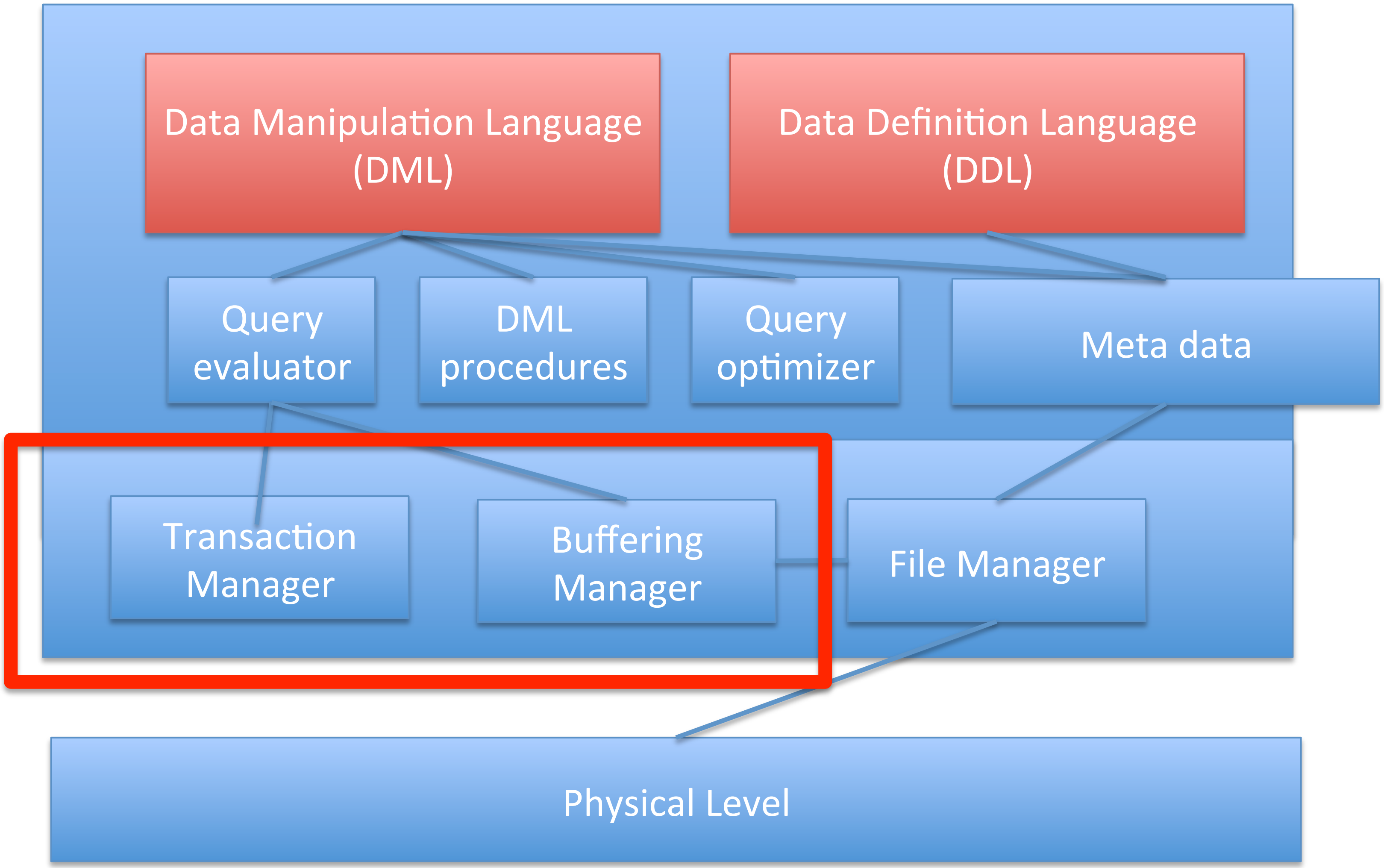
Atomicity: All actions in the defining the transaction happen, or none happen.

Consistency: If each transaction is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one transaction is isolated from that of other transactions

Durability: If a transaction finishes successfully, its effects persist.

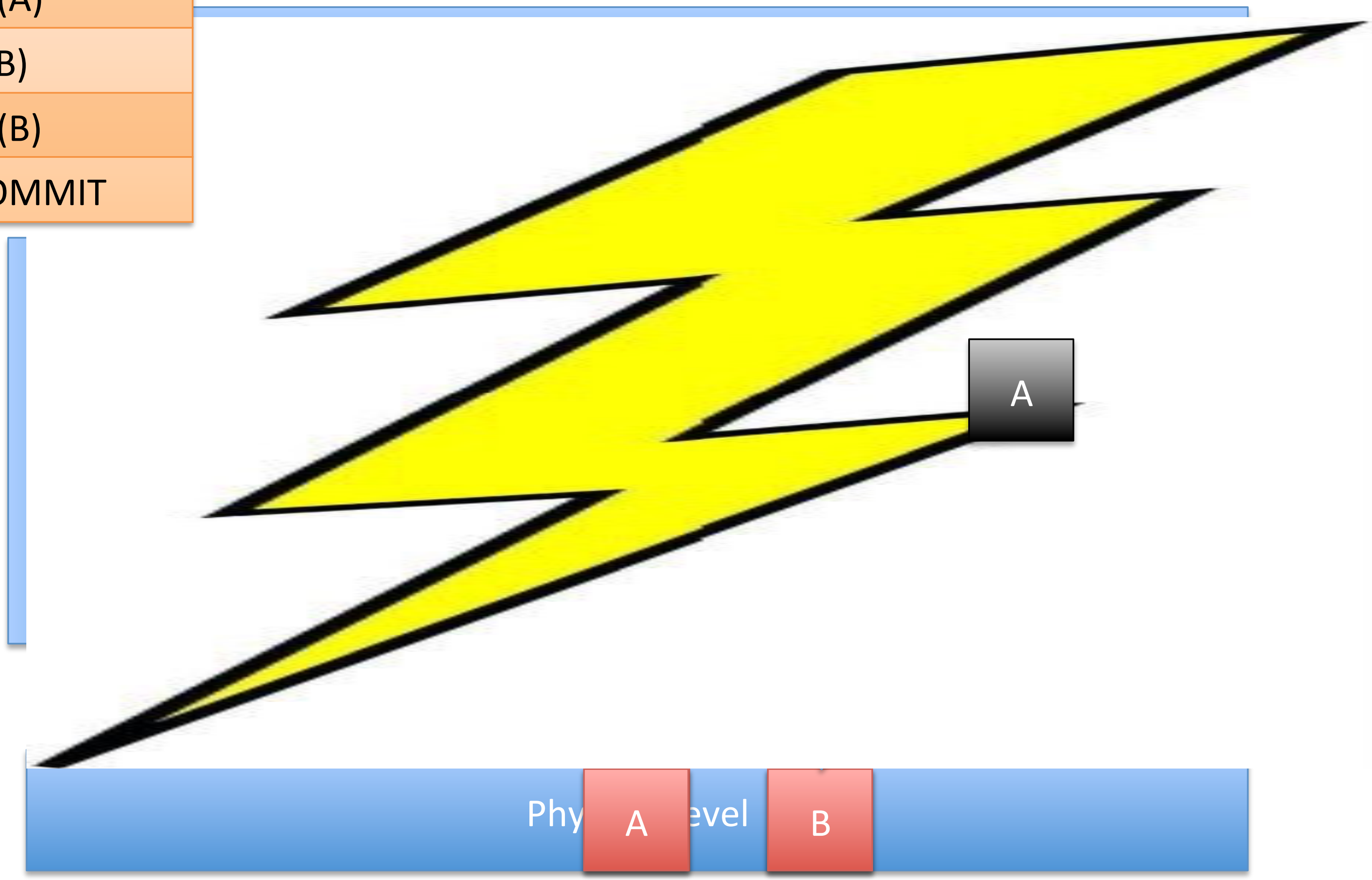
Database System Architecture



Buffer

T1
BEGI
A = A
B = B
COM

T1
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT



Transaction Durability

- Records stored on disk
- Buffer manager
  - Pages are copied into memory a…
    disk at the discretion of t…
  - One could forc…

This is too slow!

Write-Ahead Log

Log changes before the database is updated

Assume that the log lives on stable storage

Transaction finishes

- Write a commit to the log
- Make sure that all log messages are flushed before acknowledgment

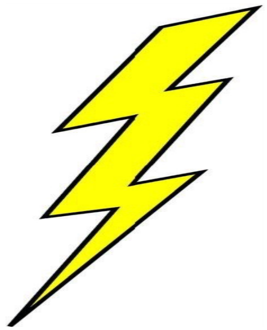
Transaction fails

- Undo uncommitted transactions
- Redo committed transactions

CRASH and REDO

T1
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

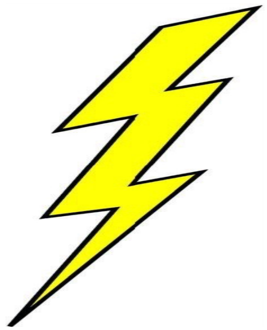
Write-Ahead Log
(T1, BEGIN)
(T1, A, 1000, 900)
(T1, B, 500, 600)
(T1, COMMIT)



CRASH and UNDO

T1
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

Write-Ahead Log
(T1, BEGIN)
(T1, A, 1000, 900)
(T1, B, 500, 600)



Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Concurrency control is automatic
  – System automatically inserts lock/unlock requests and schedules actions of different transactions.
  – Ensures that resulting execution is equivalent to executing the transaction one after the other in some order.

Transactions in Applicitions

JDBC connection

```
String url = "jdbc:mysql://localhost/";
String dbName = "imdb";
String driver = "com.mysql.jdbc.Driver";
String userName = "root";
String password = "";

try {
    Class.forName(driver);
    Connection conn = DriverManager.getConnection(url+dbName,userName,password);
    System.out.println("Connected to MySQL");
```

Database operations

```
     conn.close();
     System.out.println("Disconnected from MySQL");
 } catch (Exception e) {
     e.printStackTrace();
 }
```

JDBC dynamic SQL

```
try {
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT gender, count(*) FROM person GROUP BY gender");
    while (rs.next()) {
        System.out.println(rs.getString("gender")+": "+rs.getInt(2));
    }

    st.executeUpdate("DROP TABLE IF EXISTS JDBCtest");
    st.executeUpdate("CREATE TABLE JDBCtest(id int, string varchar(10))");
    st.executeUpdate("INSERT INTO JDBCtest VALUES (1,\"Tada!\")");
}
catch(SQLException s){
    System.out.println(s.toString());
}
```

Use caution when creating SQL based on user input!

JDBC static SQL

```java
PreparedStatement insertPerson =
conn.prepareStatement("INSERT INTO person VALUES (?,?,?,?,?,?)"); // Create prepare
insertPerson.setInt(1, 123456);
insertPerson.setString(2, "John Doe");
insertPerson.setString(3, "M");
insertPerson.setDate(4, new java.sql.Date(160617600000)); // Set date, given in mil
insertPerson.setNull(6,java.sql.Types.INTEGER); // Set to NULL
insertPerson.executeUpdate(); // Execute prepared statement with current parameters
```

Efficiency issues

Connection takes time to establish

– reuse for multiple operations

It takes time to parse dynamic SQL

– prepared statements execute faster

Understand the DBMSd

– `ORDER BY` may force creation of full result within the DBMS before any output reaches the application.

Cursors

Cursor allows the result to be traversed.

JDBC examples

- `Statement s = con.createStatement (ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)`
- `Statement s = con.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)`

Cursors

`TYPE_FORWARD_ONLY(CONCUR_READ_ONLY)`

Single (forward) pass through the result set.

`TYPE_FORWARD_ONLY(CONCUR_UPDATABLE)`

Single (forward) pass through the result set, to update rows.

`TYPE_SCROLL_INSENSITIVE`

Cursor moves forward and backward. Result set is not updated concurrently.

`TYPE_SCROLL_SENSITIVE`

Cursor moves forward and backward. Result set is updated concurrently.

Four examples

1. Movies by year – imperative way
2. Movies by year – SQL centric way
3. Iterating through a large result set
4. Iterating through a filtered result set

Transactions in JDBC

```
conn.setAutoCommit(false);
// Disable automatic commit
conn.commit();
// Commit all pending updates
conn.setSavepoint();
  // Something to roll back to
conn.rollback();
// Abort all pending updates
```

Isolation level syntax

Begin transaction with:

```
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED |
  READ COMMITTED |
  REPEATABLE READ |
  SERIALIZABLE }
```

SQL isolation levels

`READ UNCOMMITTED`
No locks are obtained

`READ COMMITTED`

Read locks are immediately released - read values may change during the transaction

`REPEATABLE READ`
2PL but no lock when adding new tuples

`SERIALIZABLE`:

2PL with lock when adding new tuples

Language integration

Little languages, database integration

E.g. Ruby on Rails

New query sublanguages for mainstream languages such as C#.

E.g. LINQ

Automatic translation to SQL.