Dirk Habich

Scalable Data Management (SDM)

Traditional Transaction Processing

TECHNISCHE UNIVERSITÄT DRESDEN
Dresden Database Research Group

Transactions Overview

Dresden Database Research Group

Transactions

- A **transaction** is the execution of a **sequence** of one or more **operations** (e.g., SQL queries) on a shared database to perform some higher-level function
- It is the basic unit of change in a DBMS: Partial transactions are not allowed!

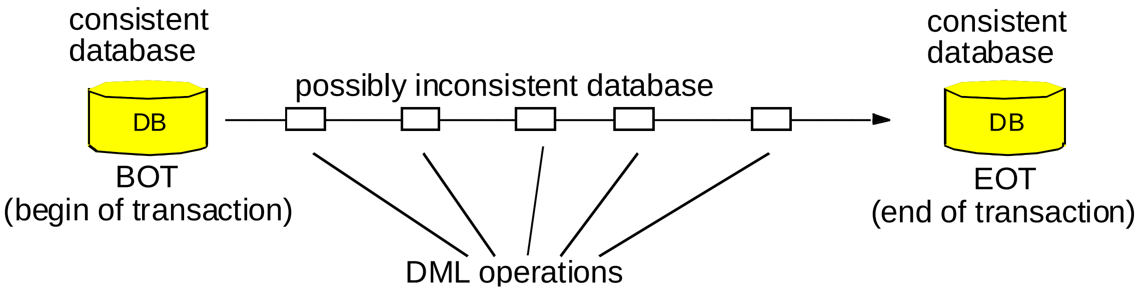
Example

- Move $100 from Andy’s bank account to his bookie’s account
- Transaction
  - Check whether Andy has $100
  - Deduct $100 from his account
  - Add $100 to his bookie’s account

Transaction Management

Principle of a transaction

- sequence of successive DB operations that transform a database from a consistent state into another consistent state
  surrounded by: BOT …….. EOT (Commit / Abort)



- properties
  - ACID: Atomicity, Consistency, Isolation, Durability
  - A transaction will always come to an end
  - Normal (commit): changes are permanently stored within the DB
  - Abnormal (abort/rollback): already composed changes are undone
- note: EOT state is not necessarily different from BOT state

Strawman System

Execute each transaction one-by-one (i.e., serial order) as they arrive at the DBMS

- One and only one transaction can be running at the same time in the DBMS

Before a transaction starts, copy the entire database to a new file and make all changes to that file

- If the transaction completes successfully, overwrite the original file with the new one
- If the transaction fails, just remove the dirty copy

→ Better approach: allow concurrent execution of independent transactions

- Question: Why do we want that?
  - Utilization/throughput (“hide” waiting for I/Os)
  - Increased response times to users
- But we also would like
  - Correctness
  - Fairness

Problem Statement

Arbitrary interleaving can lead to

- Temporary inconsistency (ok, unavoidable)
- Permanent inconsistency (bad!)

→ **Need formal correctness criteria**

Definitions

Dresden Database Research Group

Definitions

- A transaction may carry out many operations on the data retrieved from the database
- However, the DBMS is only concerned about what data is read/written from/to the database
  - Changes to the “outside world” are beyond the scope of the DBMS

Database

- A fixed set of named data objects (A, B, C, …)

Transaction

- A sequence of read and write operations (R(A), W(B), …)
  - DBMS’s abstract view of a user program

Transactions in SQL

- A new transaction starts with the **begin** command
- The transaction stops with either **commit** or abort:
  - If **commit**, all changes are saved
  - If **abort**, all changes are undone so that it’s like as if the transaction never executed at all
- Transaction can abort itself or the DBMS can abort it

Correctness Criteria: ACID

Atomicity

- All actions in the transaction happen, or none happen → *"all or nothing"*

Consistency

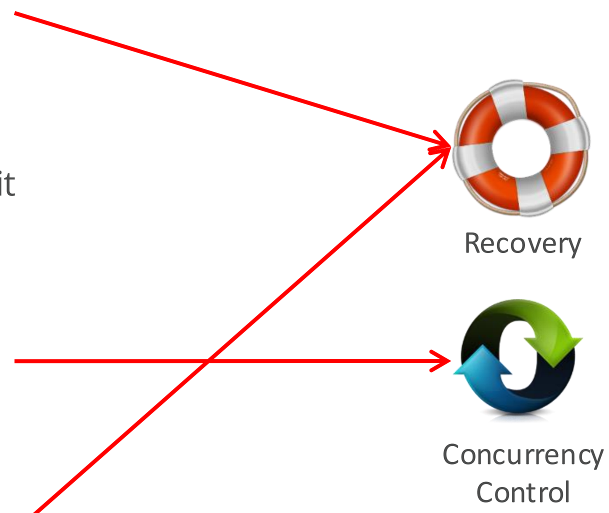
- If each transaction is consistent and the DB starts consistent, then it ends up consistent → *"it looks correct to me"*

Isolation

- Execution of one transaction is isolated from that of other transactions → *"as if alone"*

Durability

- If a transaction commits, its effects persist → "survive failures"



Atomicity of Transactions

Dresden Database Research Group

Two possible outcomes of executing a transaction:

- Transaction might commit after completing all its actions
- or it could abort (or be aborted by the DBMS) after executing some actions

DBMS guarantees that transaction are atomic

- From user's point of view: transaction always either executes all its actions, or executes no actions at all

Mechanisms for Ensuring Atomicity

- Logging
  - DBMS logs all actions so that it can undo the actions of aborted transactions
- Shadowing
  - DBMS makes copies of pages and transactions make changes to those copies
  - Only when the transaction commits the page is made visible to others

We take $100 out of Andy's account but then there is a power failure before we transfer it to his bookie.

When the database comes back on-line, what should be the correct state of Andy's account?

Consistency

Dresden Database Research Group

Database 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 transactions starts (running alone), it also will be after
- Transaction consistency is the application's responsibility
  - We won't discuss this further…

Isolation

Dresden Database Research Group

Users submit transactions, and each transaction executes as if it was running by itself

Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions

How do we achieve this? → Many methods - two main categories

- **Pessimistic** – Don’t let problems arise in the first place
- **Optimistic** – Assume conflicts are rare, deal with them after they happen

Durability

Dresden Database Research Group

All results of successful transactions have to be made persistent

Example

- If a flight booking reports that a seat has successfully been booked, then the seat will remain booked even if the system crashes

Durability can be achieved by flushing the transaction's log records to non-volatile storage before acknowledging commitment

- Write-Ahead Log (WAL): Record the changes made to the database in a log before the change is made

Isolation

Example

Two transactions

- T_1 transfers \$100 from B's account to A's
- T_2 credits both accounts with 6% interest

Assume at first A and B each have \$1000

What are the legal outcomes of running T_1 and T_2?

- Many! But A+B should be: \$2000*1.06=\$2120
- There is no guarantee that T_1 will execute before T_2 or vice-versa, if both are submitted together
- But, the net effect must be equivalent to these two transactions running serially in some order
- Legal outcomes
  - A=1166, B=954
  - A=1160, B=960
- The outcome depends on whether T_1 executes before T_2 or vice versa

T_1	T_2
BEGIN	**BEGIN**
A=A+100	A=A*1.06
B=B-100	B=B*1.06
COMMIT	**COMMIT**

Isolation

Example

Serial Execution

- T_1 transfers $100 from B's account to A's
- T_2 credits both accounts with 6% interest

Schedule

time

T_1	T_2
BEGIN A=A+100 B=B-100 **COMMIT**	
	BEGIN A=A*1.06 B=B*1.06 **COMMIT**

A=1166, B=954

time

T_1	T_2
	BEGIN A=A*1.06 B=B*1.06 **COMMIT**
BEGIN A=A+100 B=B-100 **COMMIT**	

A=1160, B=960

Interleaving Transactions

Dresden Database Research Group

We can also interleave the transactions in order to maximize concurrency

- Slow disk/network I/O
- Multi-core CPUs

Interleaving Examples

time

T_1	T_2
BEGIN	
A=A+100	
	BEGIN
	A=A*1.06
B=B-100	
COMMIT	
	B=B*1.06
	COMMIT

A=1166, B=954

≡

time

T_1	T_2
BEGIN	
A=A+100	
B=B-100	
COMMIT	
	BEGIN
	A=A*1.06
	B=B*1.06
	COMMIT

A=1166, B=954

≠

time

T_1	T_2
BEGIN	
A=A+100	
	BEGIN
	A=A*1.06
	B=B*1.06
	COMMIT
B=B-100	
COMMIT	

The bank lost $6!

Isolation

A=1166, B=960

Schedule versus DBMS View

Dresden Database Research Group

Schedule

time

T_1	T_2
BEGIN	
A=A+100	
	BEGIN
	A=A*1.06
	B=B*1.06
	COMMIT
B=B-100	
COMMIT	

DBMS's View

time

T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Isolation

Correctness and Serializability

Dresden Database Research Group

How do we judge that a schedule is correct?

- If it is equivalent to some serial execution

Serial Schedule

- A schedule that does not interleave the actions of different transactions

Equivalent Schedules

- For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule → **no matter what the arithmetic operations are!**

Serializable Schedule

- A schedule that is equivalent to **some serial execution** of the transactions
- Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Serializability

- Less intuitive notion of correctness compared to transaction initiation time or commit order, but it provides the DBMS with significant additional **flexibility** in scheduling operations.

Isolation

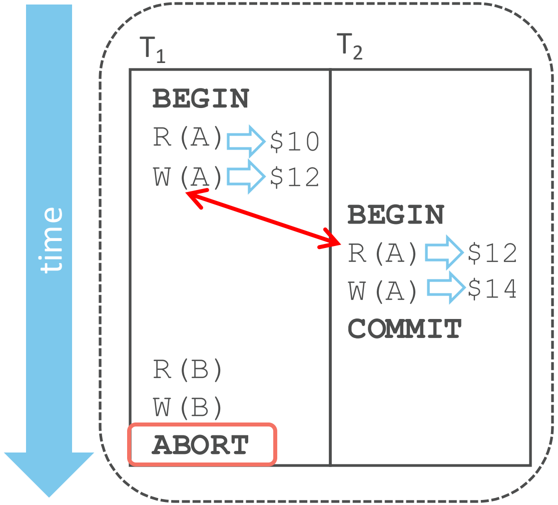
Anomalies

Interleaved Execution Anomalies

Dresden Database Research Group

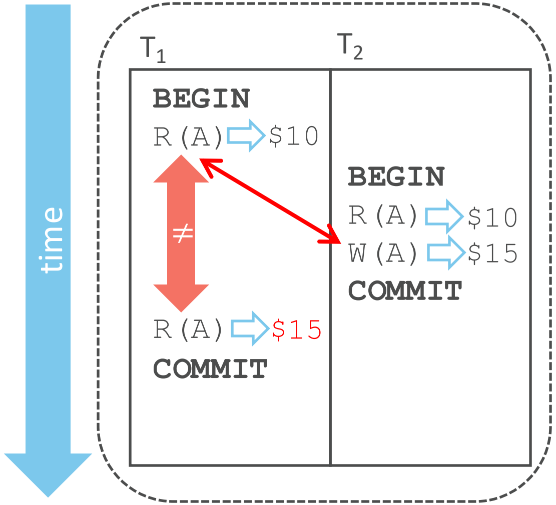
Write-Read conflicts (W-R)

- Reading Uncommitted Data → "Dirty Reads"



Read-Write Conflicts (R-W)

- Unrepeatable Reads

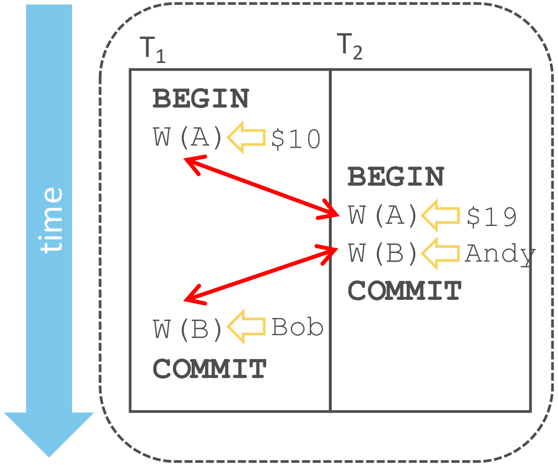


Interleaved Execution Anomalies

Dresden Database Research Group

Write-Write Conflicts (W-W)

- Overwriting uncommitted data → lost update



How could you guarantee that all resulting schedules are correct (i.e., serializable)?

→ Use locks!

Isolation

Locking

Locks

Dresden Database Research Group

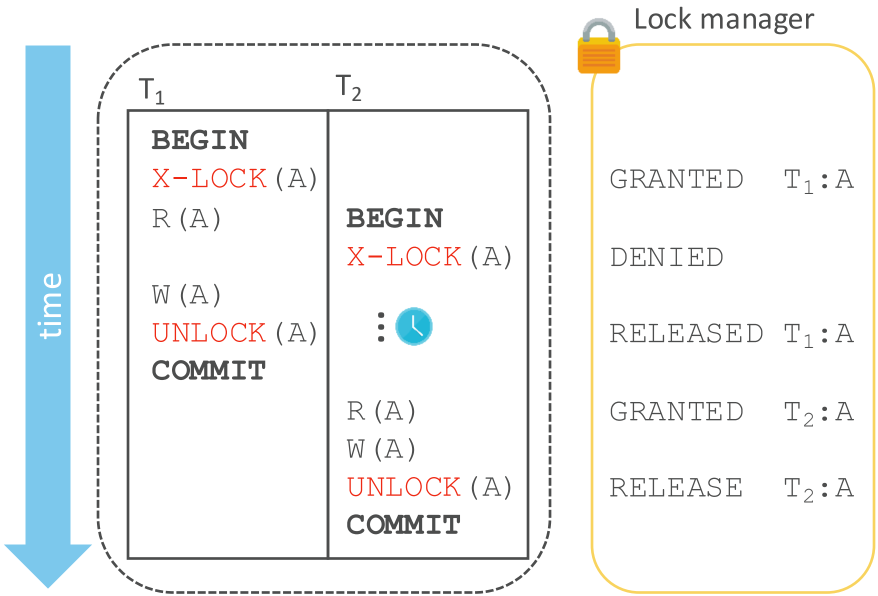
Basic Lock Types

- S-LOCK – Shared Locks (reads)
- X-LOCK – Exclusive Locks (writes)

Compatibility Matrix

	shared	exclusive
shared	✓	✗
exclusive	✗	✗

Example



Locks (2)

Dresden Database Research Group

Executing with Locks

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

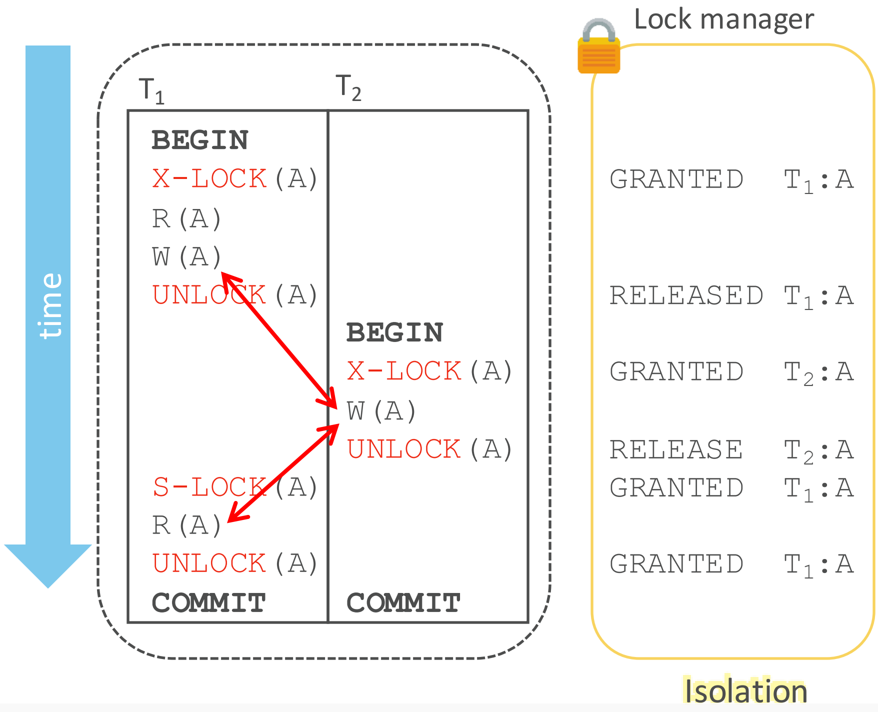
→ But this is not enough…

- We need to use a well-defined protocol that ensures that transactions execute correctly

One possible solution

- Two-Phase Locking (2PL)

Example



Two-Phase locking

Dresden Database Research Group

Motivation

- Example

time

T_1	T_2	Lock manager
BEGIN		
X-LOCK(A)		GRANTED T_1:A
R(A)		
W(A)		
UNLOCK(A)		RELEASED T_1:A
	BEGIN	
	X-LOCK(A)	GRANTED T_2:A
	W(A)	
	UNLOCK(A)	RELEASE T_2:A
S-LOCK(A)		GRANTED T_1:A
R(A)		
UNLOCK(A)		GRANTED T_1:A
COMMIT	**COMMIT**	

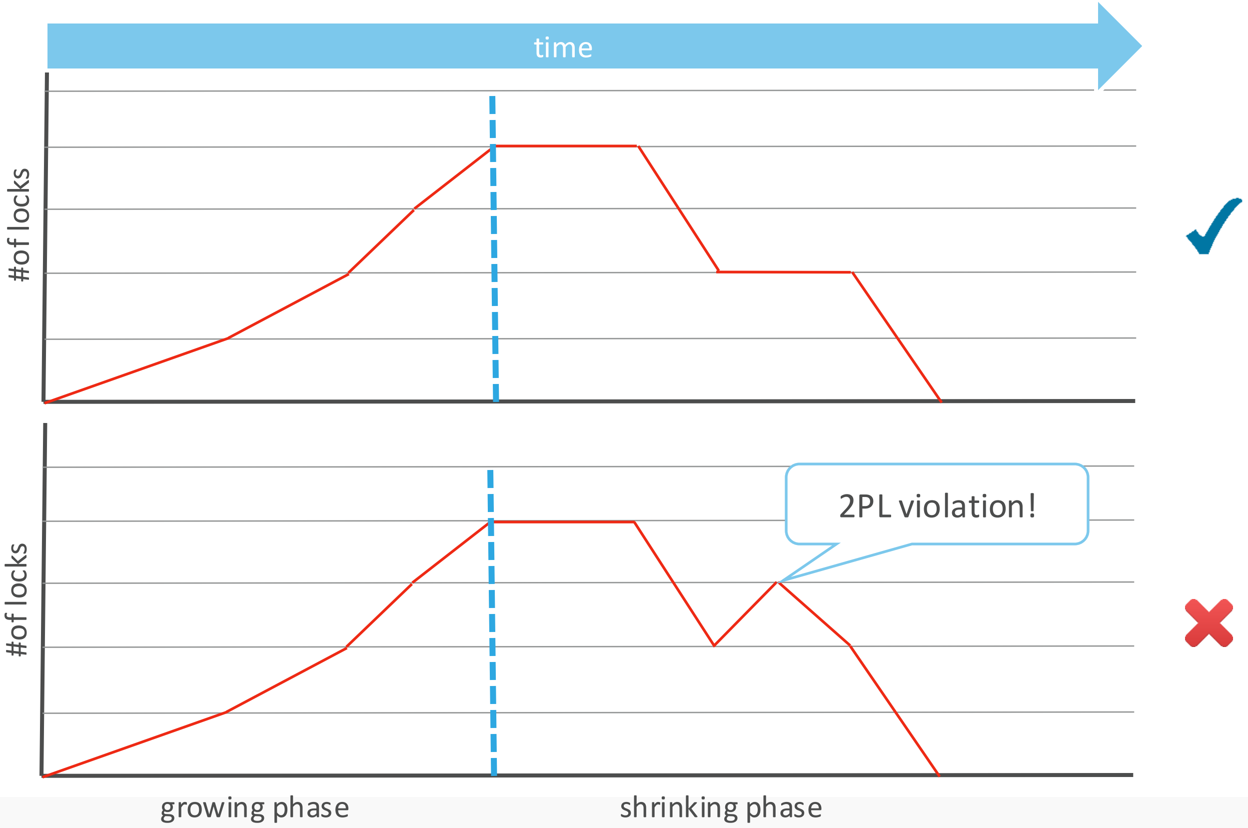
Phase 1: Growing

- Each 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

Two-Phase Locking



Example

time

T_1	T_2	Lock manager
BEGIN		
X-LOCK(A)		GRANTED T_1:A
R(A)		
W(A)		
	BEGIN	
	X-LOCK(A)	DENIED!
R(A)		
UNLOCK(A)	⋮	
COMMIT		
	W(A)	
	UNLOCK(A)	
	COMMIT	

Example

T_1	T_2	Lock manager
BEGIN		
X-LOCK(A)		GRANTED T_1:A
R(A)		
W(A)		
	BEGIN	
	X-LOCK(A)	DENIED!
R(A)		
UNLOCK(A)		RELEASED T_1:A
COMMIT		
	W(A)	GRANTED T_2:A
	UNLOCK(A)	RELEASED T_2:A
	COMMIT	

time

Two-Phase Locking

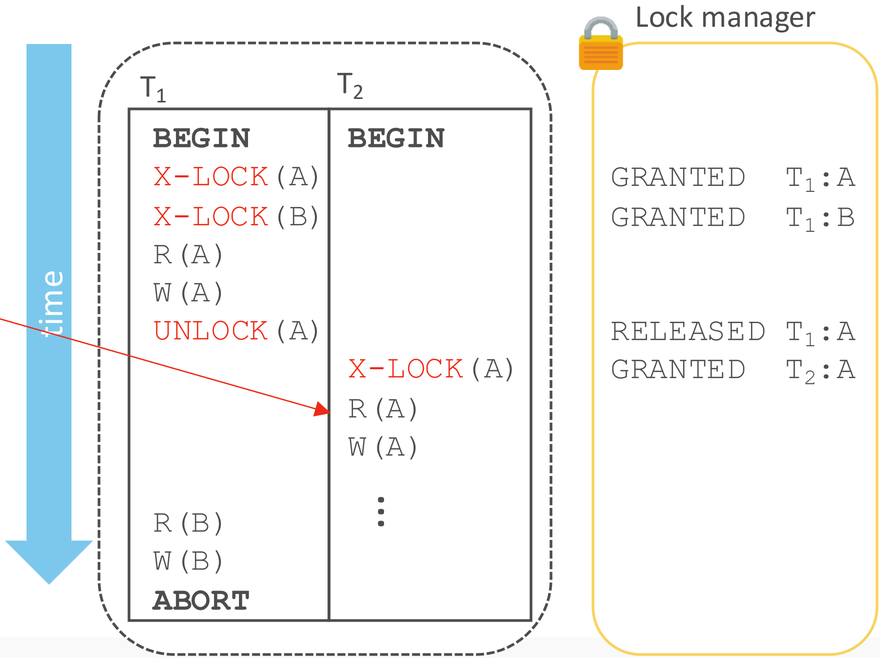
Dresden Database Research Group

2PL on its own is...

- ...sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic)
- but, it is subject to **cascading aborts**

Example

- This is a permissible schedule in 2PL, but we have to abort T_2 too
- This is all wasted work!



Strict Two-Phase Locking

Dresden Database Research Group

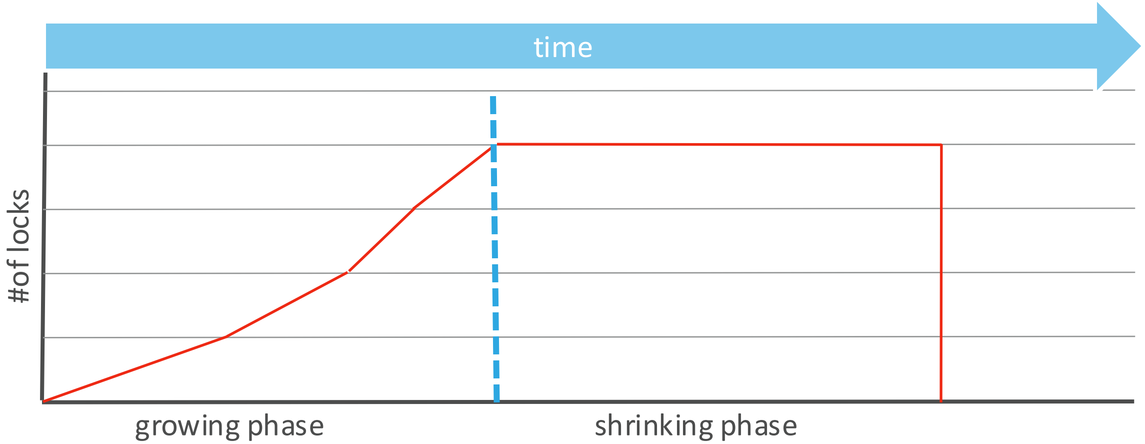
A transaction releases it write locks only after it has ended (committed or aborted)

A schedule is strict if…

- …a value written by a transaction is not read or overwritten by other transaction until that transaction finishes

Advantages

- Does **not** incur **cascading aborts**
- Aborted transactions can be undone by just restoring original values of modified tuples



Deadlocks

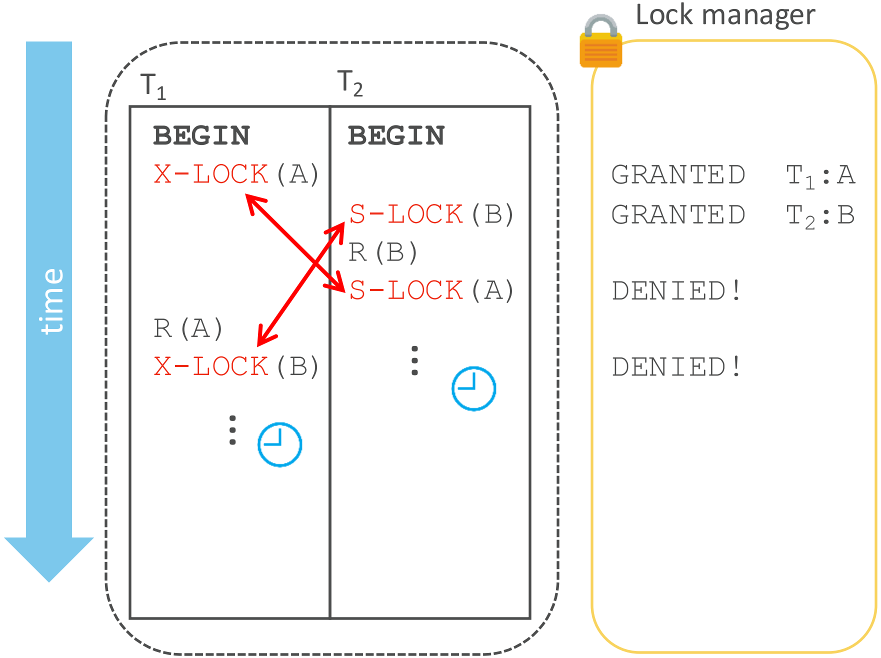
Deadlock

Deadlock

- Cycle of transactions waiting for locks to be released by each other

Two ways of dealing with deadlocks

- Deadlock **detection**
- Deadlock **prevention**



Deadlock Detection

The DBMS creates a waits-for graph

- Nodes are transactions
- Edge from T_i to T_j if T_i is waiting for T_j to release a lock

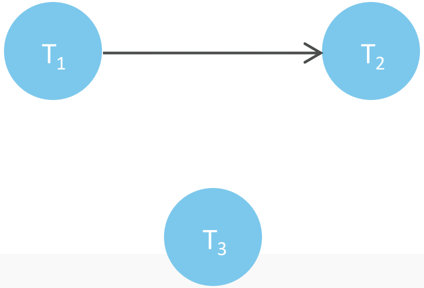
The system periodically check for cycles in waits-for graph

Schedule

time ↓

T_1	T_2	T_3
BEGIN	**BEGIN**	**BEGIN**
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-for graph



Deadlock Detection

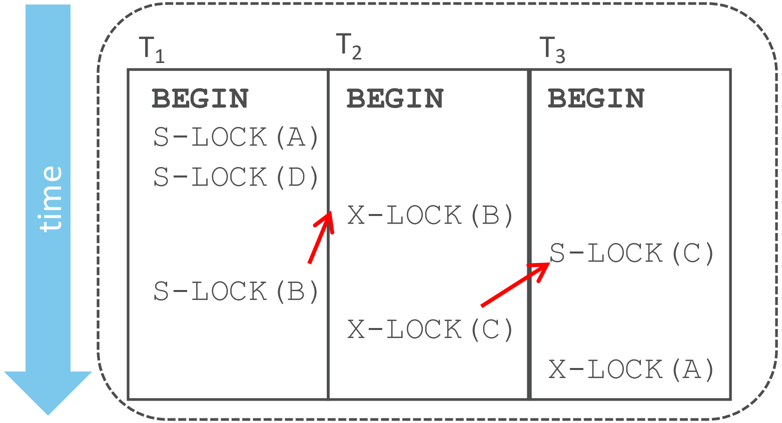
Dresden Database Research Group

The DBMS creates a waits-for graph

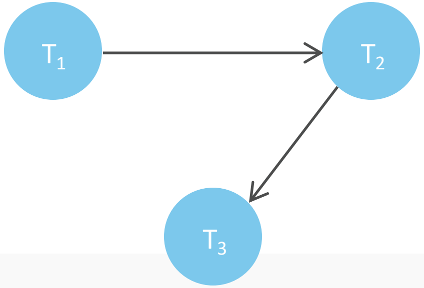
- Nodes are transactions
- Edge from T_i to T_j if T_i is waiting for T_j to release a lock

The system periodically check for cycles in waits-for graph

Schedule



Waits-for graph



Deadlock Detection

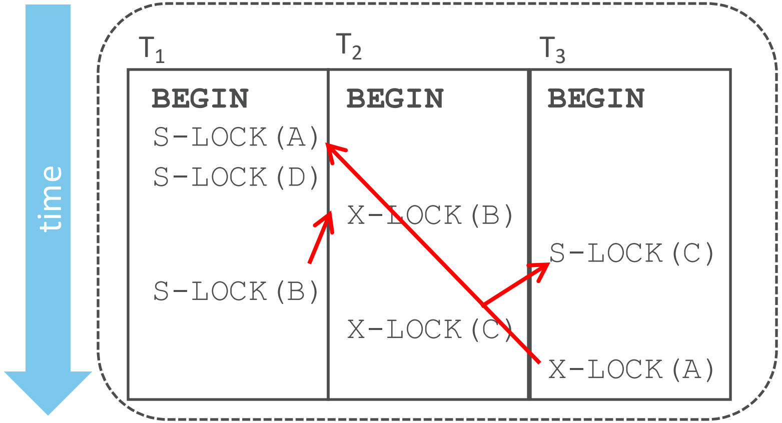
Dresden Database Research Group

The DBMS creates a waits-for graph

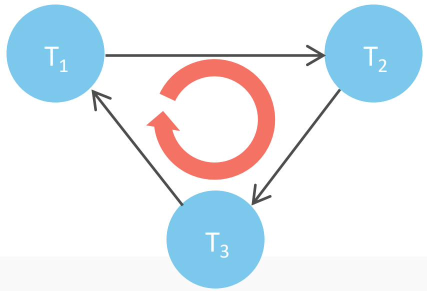
- Nodes are transactions
- Edge from T_i to T_j if T_i is waiting for T_j to release a lock

The system periodically check for cycles in waits-for graph

Schedule



Waits-for graph



Deadlock Handling

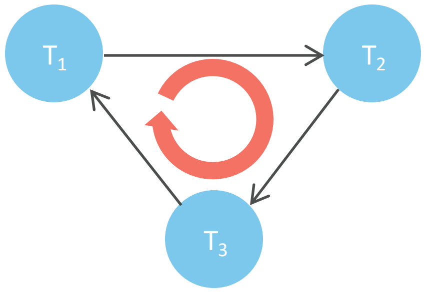
Dresden Database Research Group

What do we do if a deadlock occurs?

- Select a “victim” and rollback it back to break the deadlock

Which one do we choose?

- Decide by considering
  - age (lowest timestamp)
  - progress (least/most operations issued)
  - # of items already locked
  - # of transaction that we have to rollback with it
- We also should consider the # of times a transaction has been restarted in the past



Deadlock Prevention

Dresden Database Research Group

When a transaction tries to acquire a lock that is held by another transaction, kill one of them to prevent a deadlock

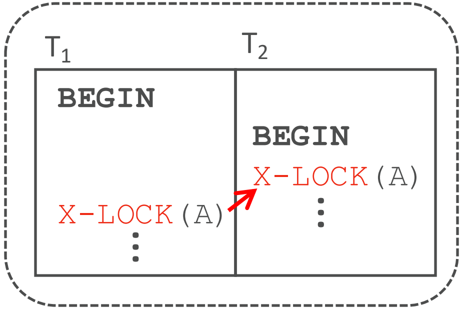
- No waits-for graph or detection algorithm

Assign priorities based on timestamps

- Older → higher priority (e.g., $T_1 > T_2$)
- Two different prevention policies
  - Wait-Die: If T_1 has higher priority, T_1 waits for T_2; otherwise T_1 aborts ("old wait for young")
  - Wound-Wait: If T_1 has higher priority, T_2 aborts; otherwise T_1 waits ("young wait for old")

Deadlock Prevention

Examples

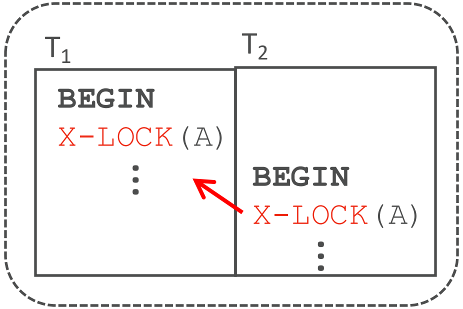


Wait-Die

- T_1 waits

Wound-Wait

- T_2 aborted



Wait-Die

- T_2 aborted

Wound-Wait

- T_2 waits

Summary

Dresden Database Research Group

Transaction

- Sequence of operations
- Serial vs. interleaving execution

ACID

- Atomicity
- Consistency
- Isolation
- Durability

Isolation

- Anomalies
- Locking
- Deadlocks