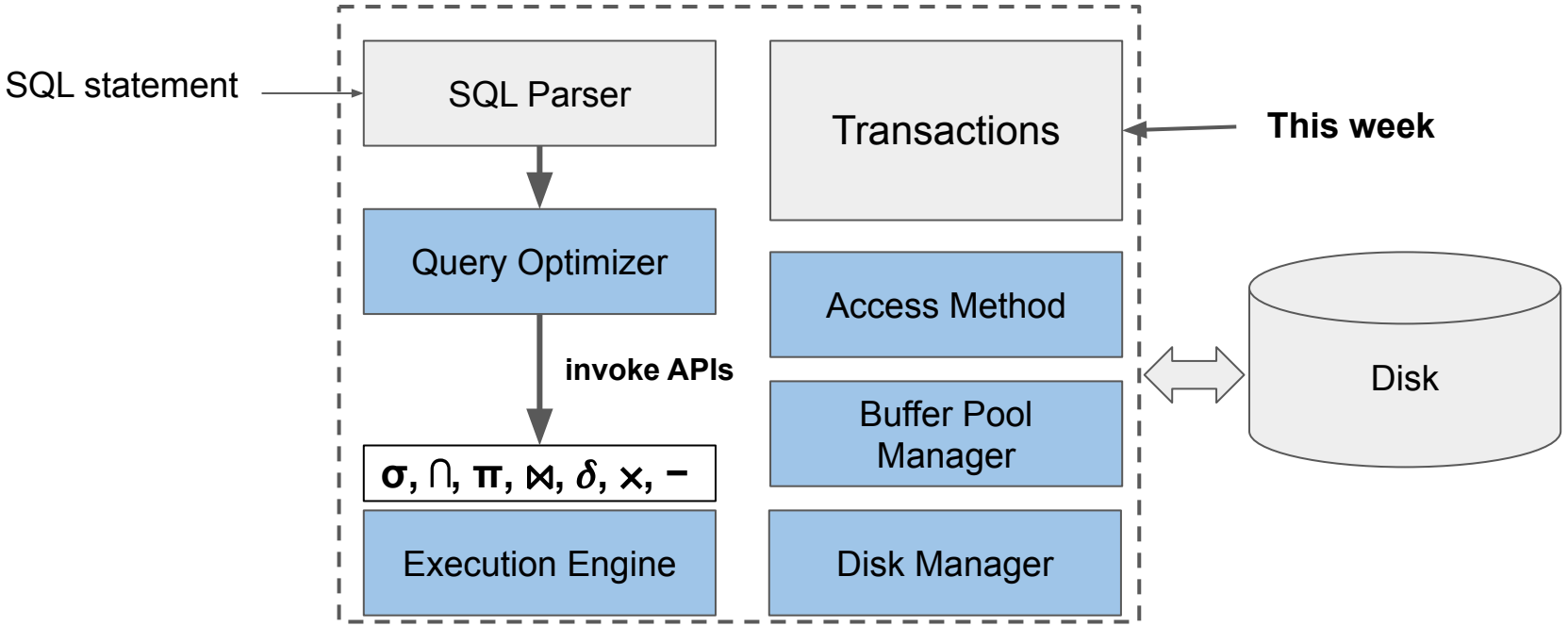
Databases and Big Data

Transactions 2

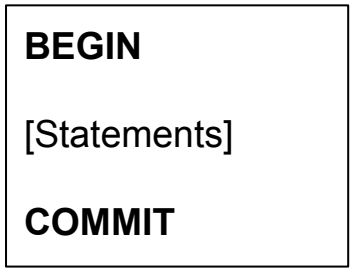
Recap

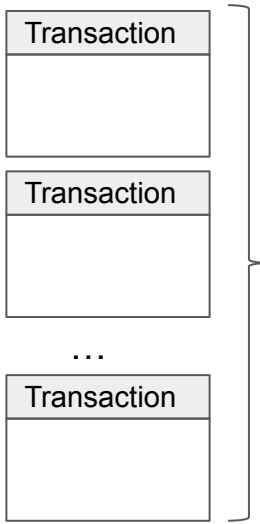


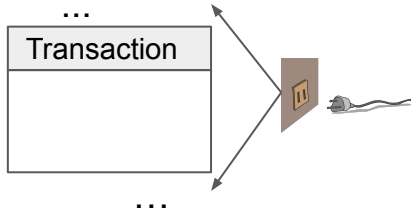
How to support **write** operations under **failures** and **concurrency**

Recap

Transaction = A sequence of operations, executed together as **<u>one indivisible unit</u>**







All or nothing (Atomicity)

Executed as if alone (Isolation)

Recap - How To Rob A Bank

XYZ	100

```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

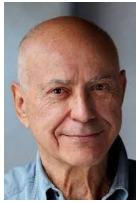
XYZ	90

XYZ	100

```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

XYZ	90

XYZ	100



```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

XYZ	90

Transaction

- Atomicity
  - Ensure correctness under failures
  - *More or less solved*
  - Using Write Ahead Logging (WAL)
- Today: Isolation
  - Ensure correctness under concurrency
  - *Still active area of research*
  - Achieved by **concurrency control** mechanism

[2018]

Consolidating Concurrency Control and Consensus for Commits under Conflicts

Shuai Mu*, Lamont Nelson*, Wyatt Lloyd†, and Jinyang Li*
*New York University, †University of Southern California

Abstract

Conventional fault-tolerant distributed transactions layer a traditional concurrency control protocol on top of the Paxos consensus protocol. This approach provides scalability, availability, and strong consistency. When used for wide-area storage, however, this approach incurs cross-data-center coordination twice, in serial: once for concurrency control, and then once for consensus. In this paper, we make the key observation that the coordination required for concurrency control and consensus is highly

sensus provides availability despite server failure. Coordinating transactions across the replicated shards with concurrency control provides strong consistency despite conflicting data accesses by different transactions. This approach is commonly used in practice. For example, Spanner [9] implements two-phase-locking (2PL) and two phase commit (2PC) in which both the data and locks are replicated by Paxos. As another example, Percolator [35] implements a variant of opportunistic concurrency control (OCC) with 2PC on top of BigTable [7] which relies on primary-backup replication and Paxos.

[2016]

Carousel: Low-Latency Transaction Processing for Globally-Distributed Data

Xinan Yan
University of Waterloo
xinan.yan@uwaterloo.ca

Linguan Yang
University of Waterloo
l69yang@uwaterloo.ca

Hongbo Zhang
University of Waterloo
hongbo.zhang@uwaterloo.ca

Xiayue Charles Lin
University of Waterloo
xy3lin@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Kenneth Salem
University of Waterloo
kmsalem@uwaterloo.ca

Tim Brecht
University of Waterloo
brecht@uwaterloo.ca

ABSTRACT

The trend towards global applications and services has created an increasing demand for transaction processing on globally-distributed data. Many database systems, such as Spanner and CockroachDB, support distributed transactions but require a large number of wide-area network roundtrips to commit each transaction and ensure the transaction's state is durably replicated across multiple datacenters. This can significantly increase transaction completion time, resulting in developers replacing database-level transactions with

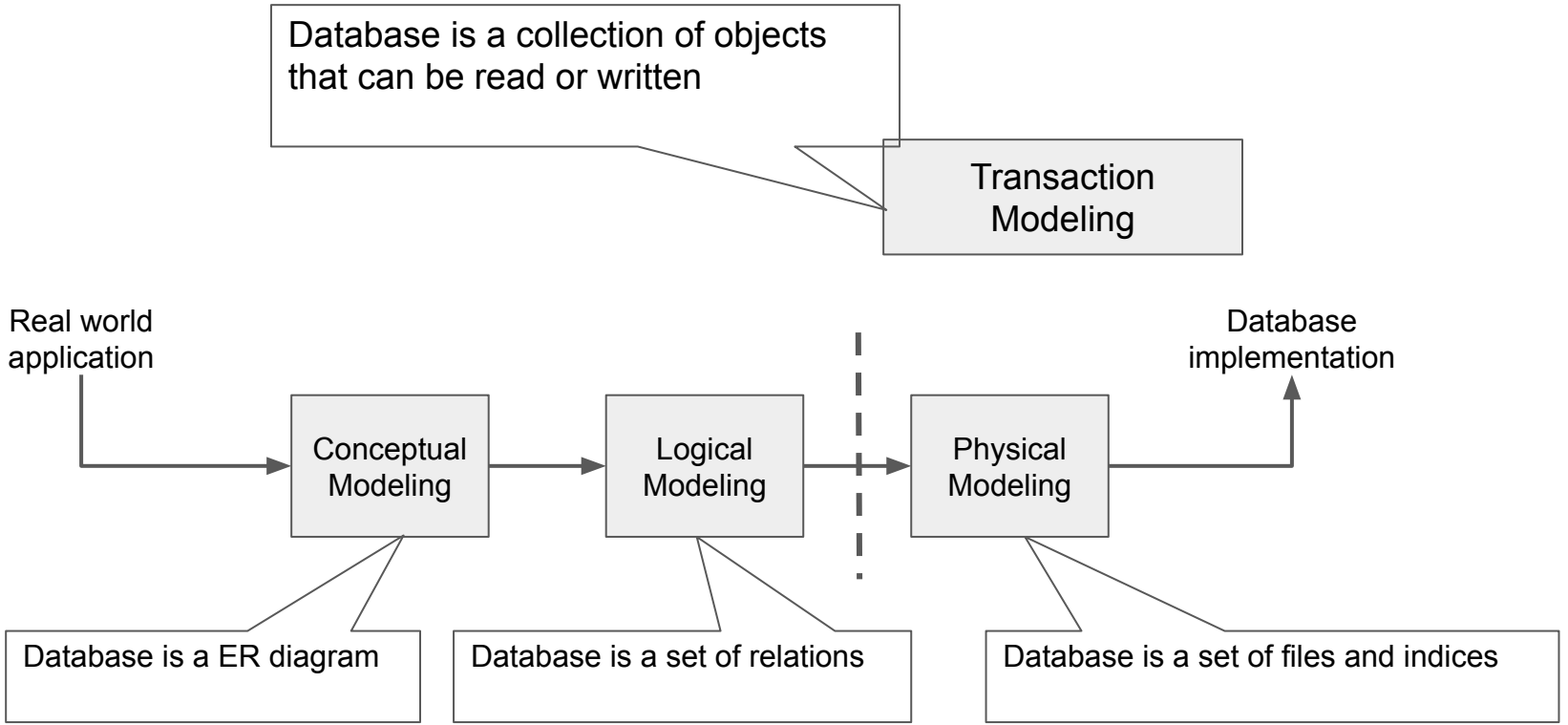
KEYWORDS

globally-distributed data; distributed transactions

ACM Reference Format:

Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3196912

Transaction Modeling

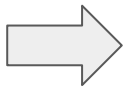


Transaction Modeling

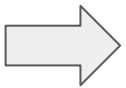
- Database is a set of objects
  - R(A): read object A
  - W(B): write to object B
- Transaction is a sequence of read and write

```
update Account
set balance = balance - 10
where accountNo="A"

update Account
set balance = balance + 10
where accountNo="B"
```



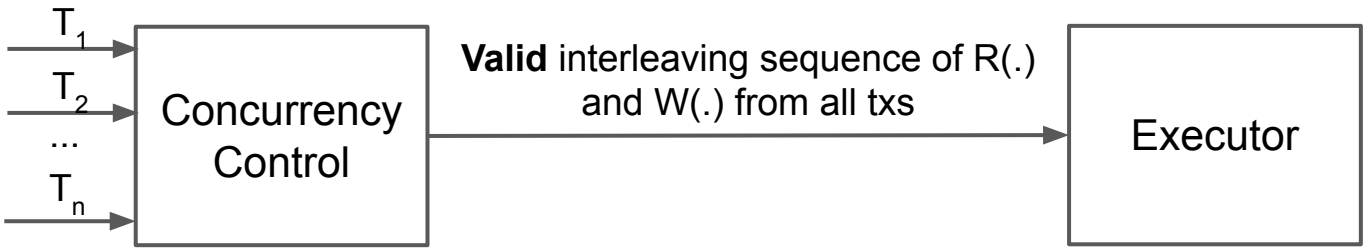
BEGIN
A = A - 10
B = B + 10
END



BEGIN
R(A)
W(A)
R(B)
W(B)
END

Transaction

- Concurrency Control



Serializability

- Standard for correctness
- Example: 2 account A and B, each has $1000
  - T_1: move $100 from A to B
  - T_2: credit 6% interest to both account

T_1

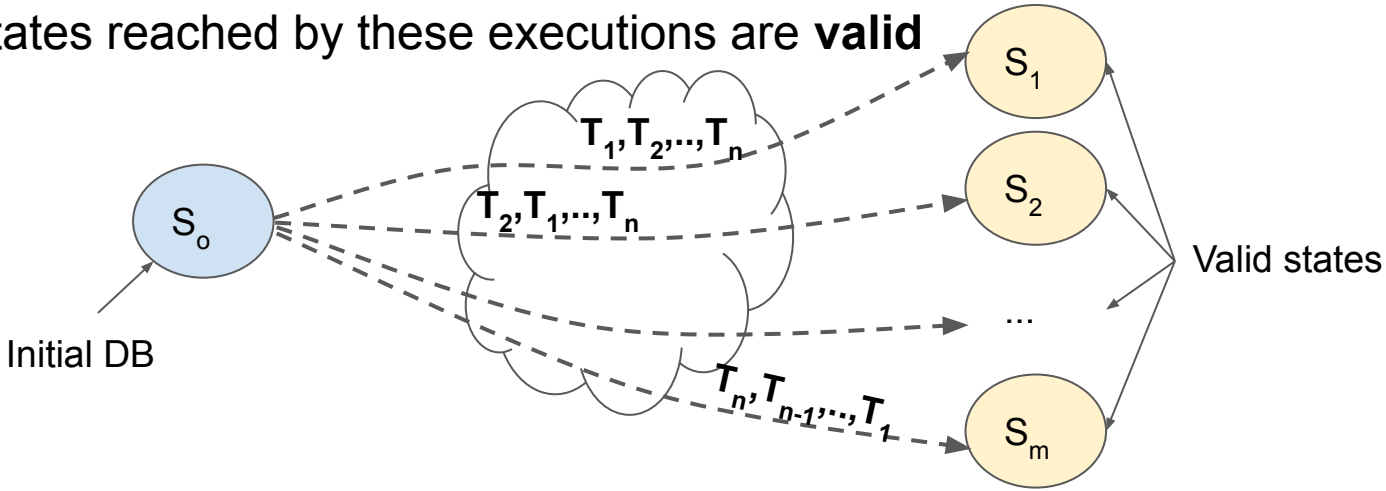
```
BEGIN
  A = A - 100
  B = B + 100
END
```

T_2

```
BEGIN
  A = A*1.06
  B = B*1.06
END
```

Serializability

- Starting from initial state S_0
  - Execute $T_1, T_2, ..., T_n$ serially in random order (**n!** choices)
    - No interleaving yet!
  - All final states reached by these executions are **valid**



Serializability

A=1000	B=1000

TIME

T_1

```
BEGIN
  A = A - 100
  B = B + 100
END
```

T_2

```
BEGIN
  A = A*1.06
  B = B*1.06
END
```

A=954	B=1166

T_2

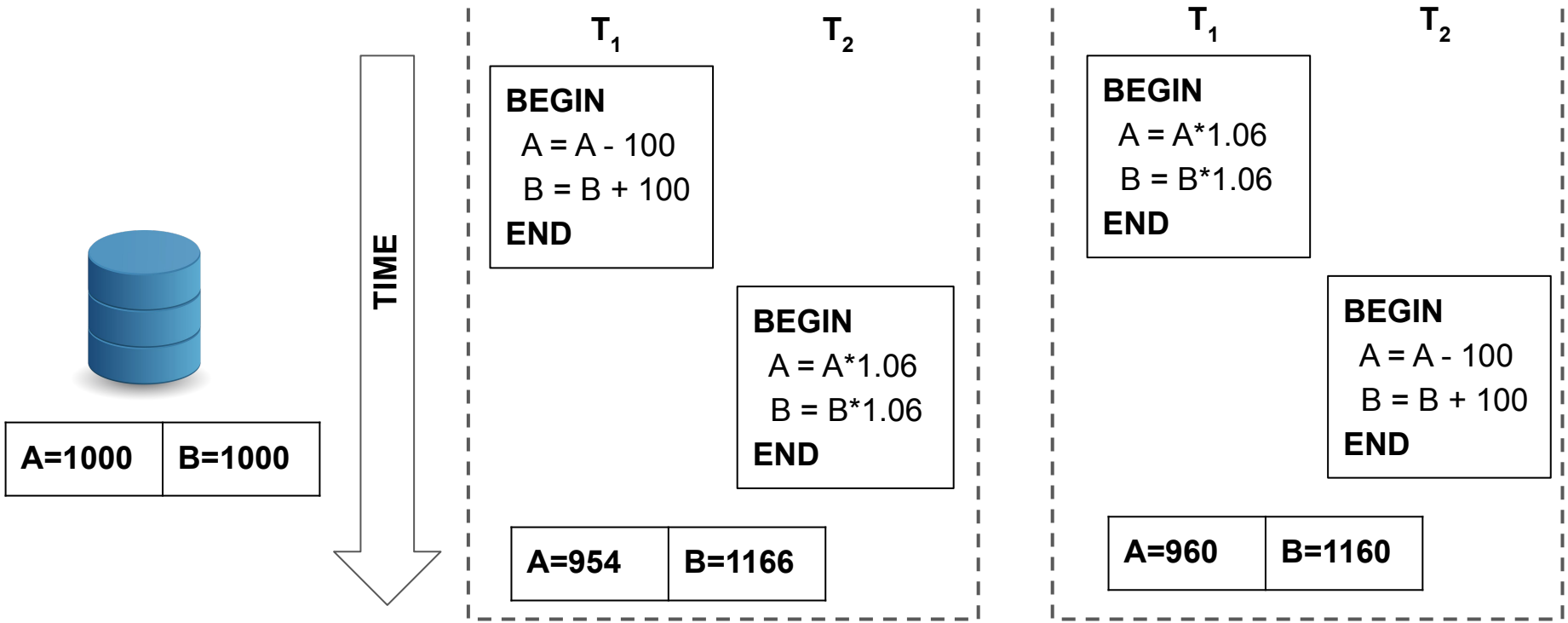
```
BEGIN
  A = A*1.06
  B = B*1.06
END
```

T_1

```
BEGIN
  A = A - 100
  B = B + 100
END
```

A=960	B=1160

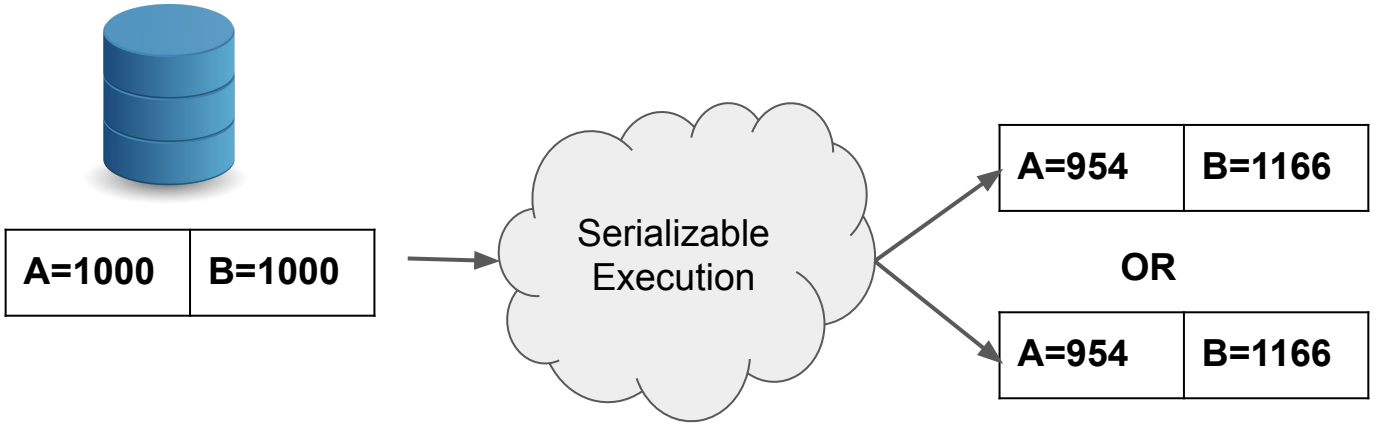
Serializability



Serializability

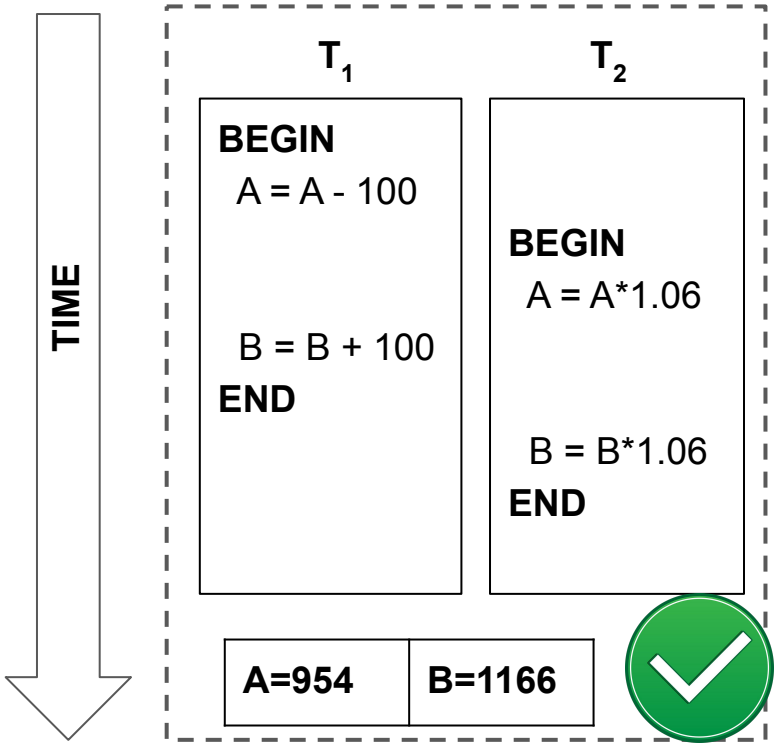
- Serializable execution
  - Any interleaving of read/write that **reaches a valid state**

A serializable schedule is the one that always leaves the database in consistent state

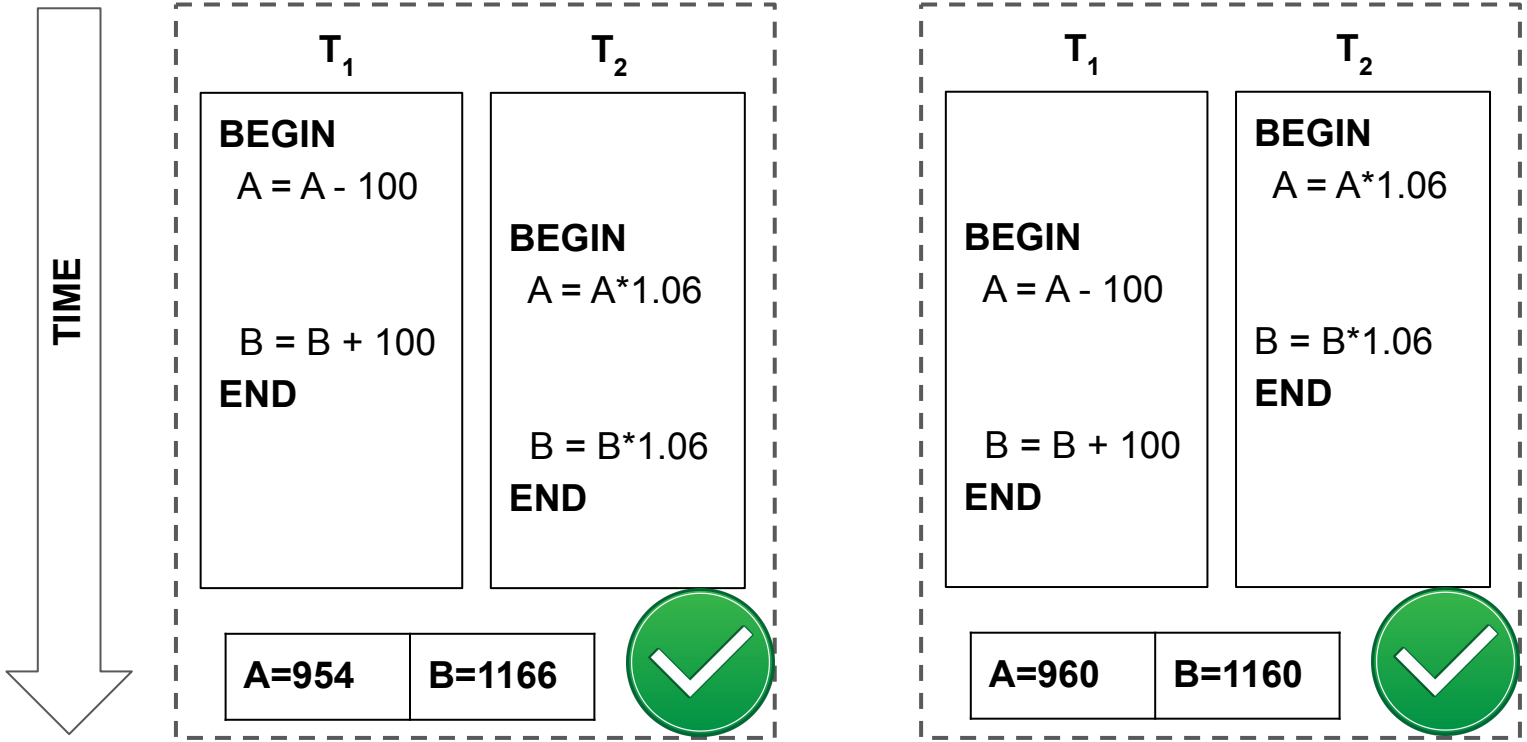


Serializability

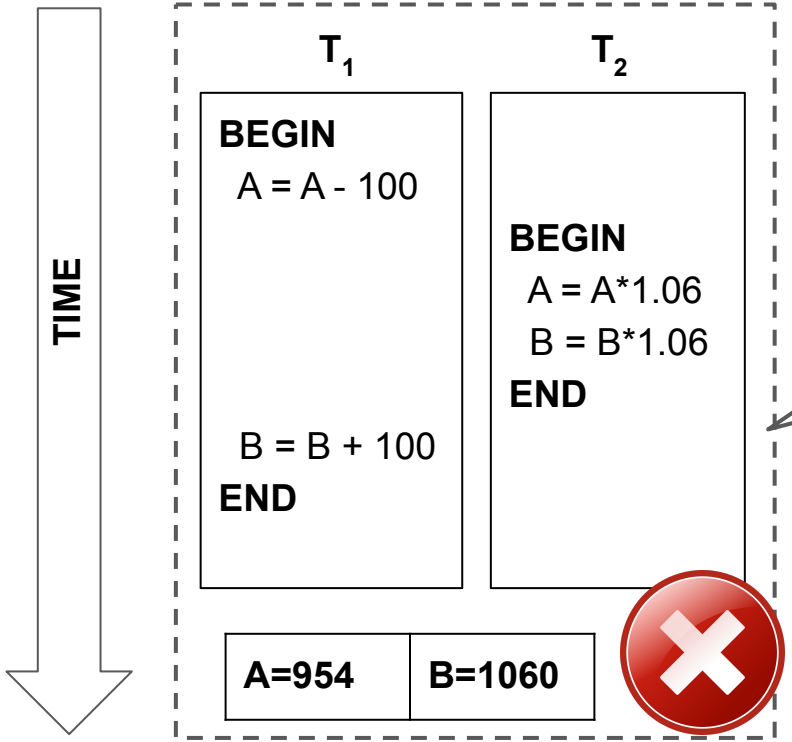
- Interleaving:
  - vs. serial execution
  - To maximize concurrency (like threads)
    - Some operations are slow
    - Some waits for input, etc.



Serializability

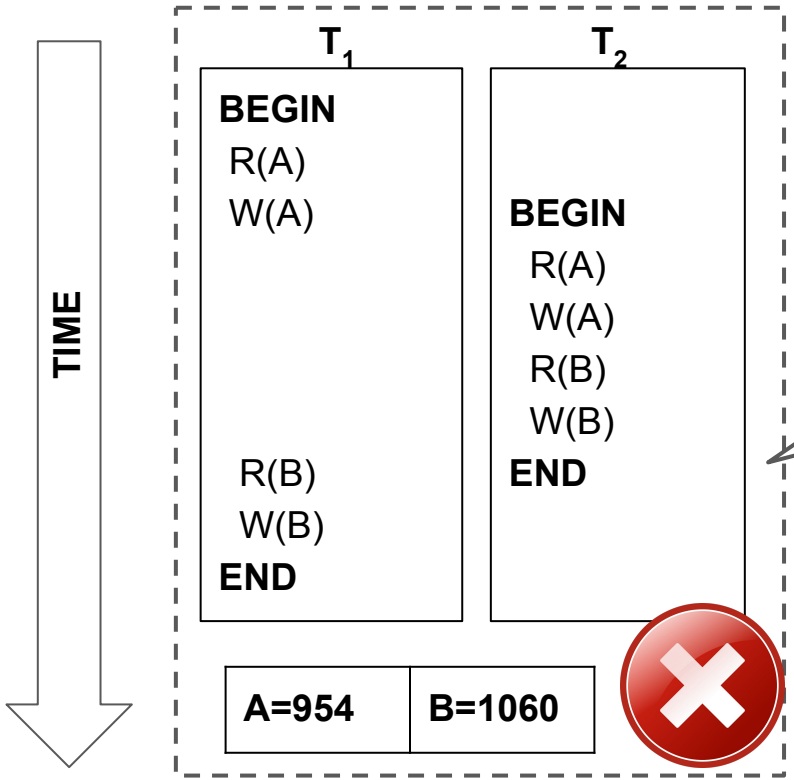


Serializability



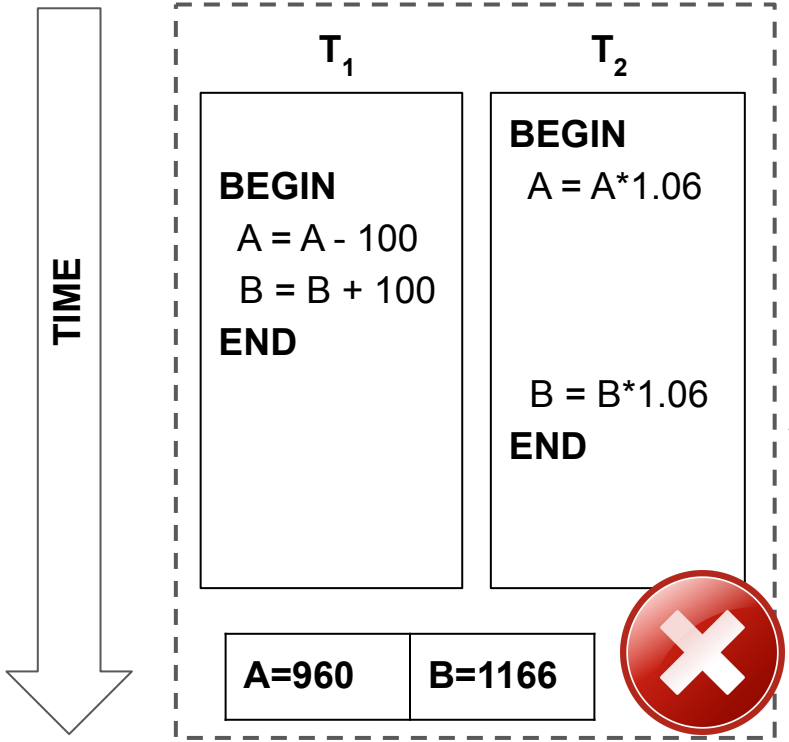
The bank is missing $160

Serializability



The bank is missing $160

Serializability



$6 out of thin air

Serializability

- Given an interleaving sequence
  - Can DBMS check if it is serializable without executing?
    - VERY HARD!!
    - Depending on specific values, “bad” sequence may still be serializable
- In practice:
  - Check if the sequence is **conflict serializable**

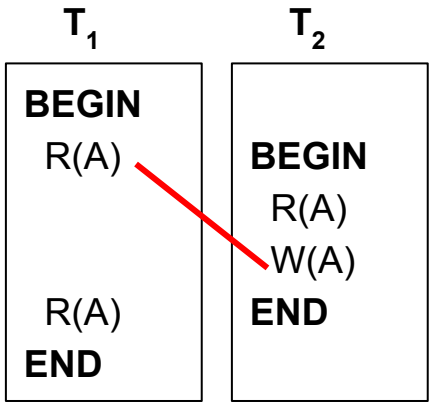
ConflictSerializable(X) → Serializable(X)

The other direction is not true

Conflict

- Two operators **conflict** iff
  - Belong to two transactions
  - On the same object
  - One of them is write

Read-Write (R-W)
Write-Read (W-R)
Write-Write (W-W)

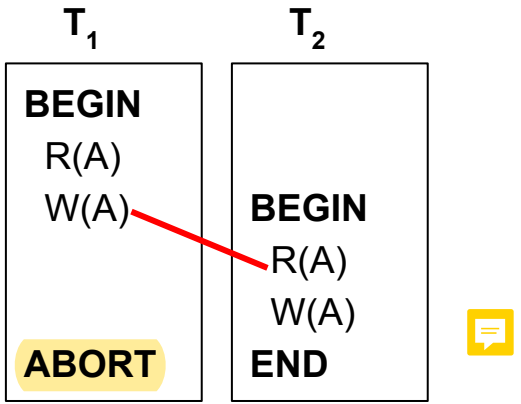


R-W conflict
(Unrepeatable Read)

Conflict

- Two operators **conflict** iff
  - Belong to two transactions
  - On the same object
  - One of them is write

Read-Write (R-W)
Write-Read (W-R)
Write-Write (W-W)

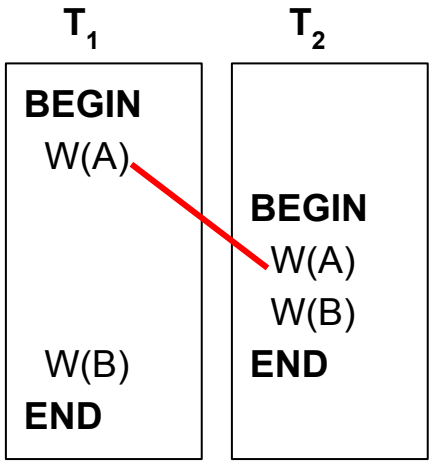


W-R conflict
(Uncommitted Read)

Conflict

- Two operators **conflict** iff
  - Belong to two transactions
  - On the same object
  - One of them is write

Read-Write (R-W)
Write-Read (W-R)
Write-Write (W-W)



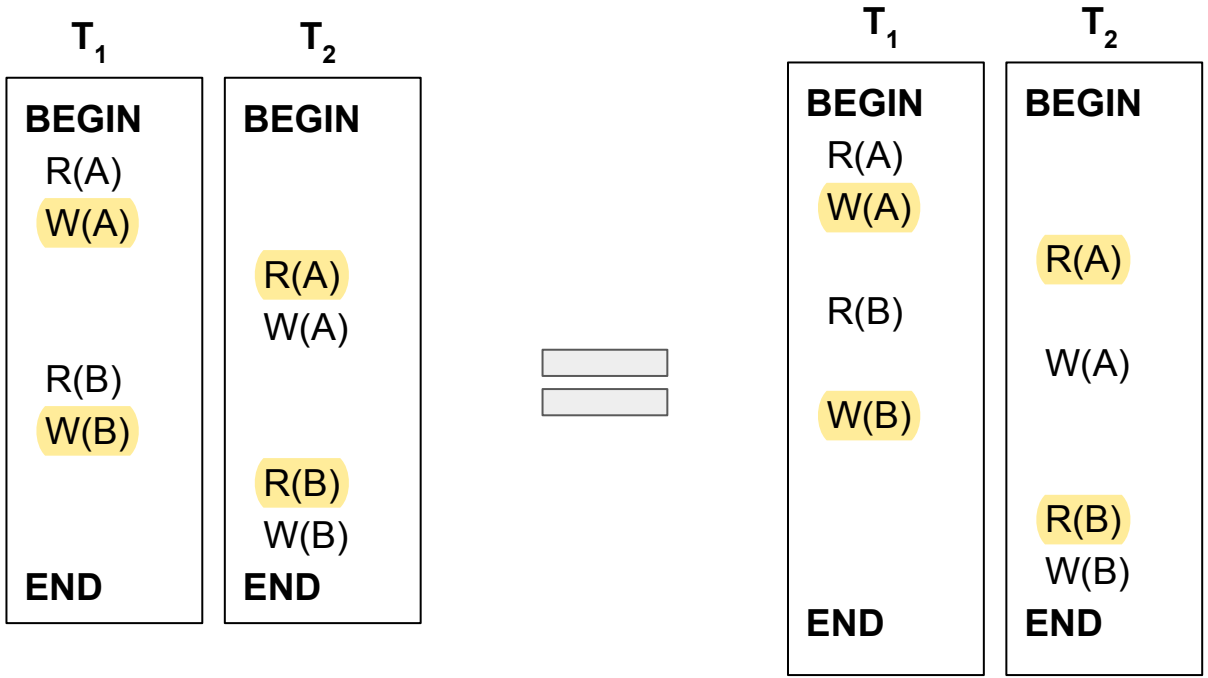
W-W conflict
(Lost Update)

Conflict Equivalence

- Two sequence X_1, X_2 are **conflict equivalent**:
  - From the same transaction
  - Any pair of conflict is ordered the same way.

Conflict Equivalence

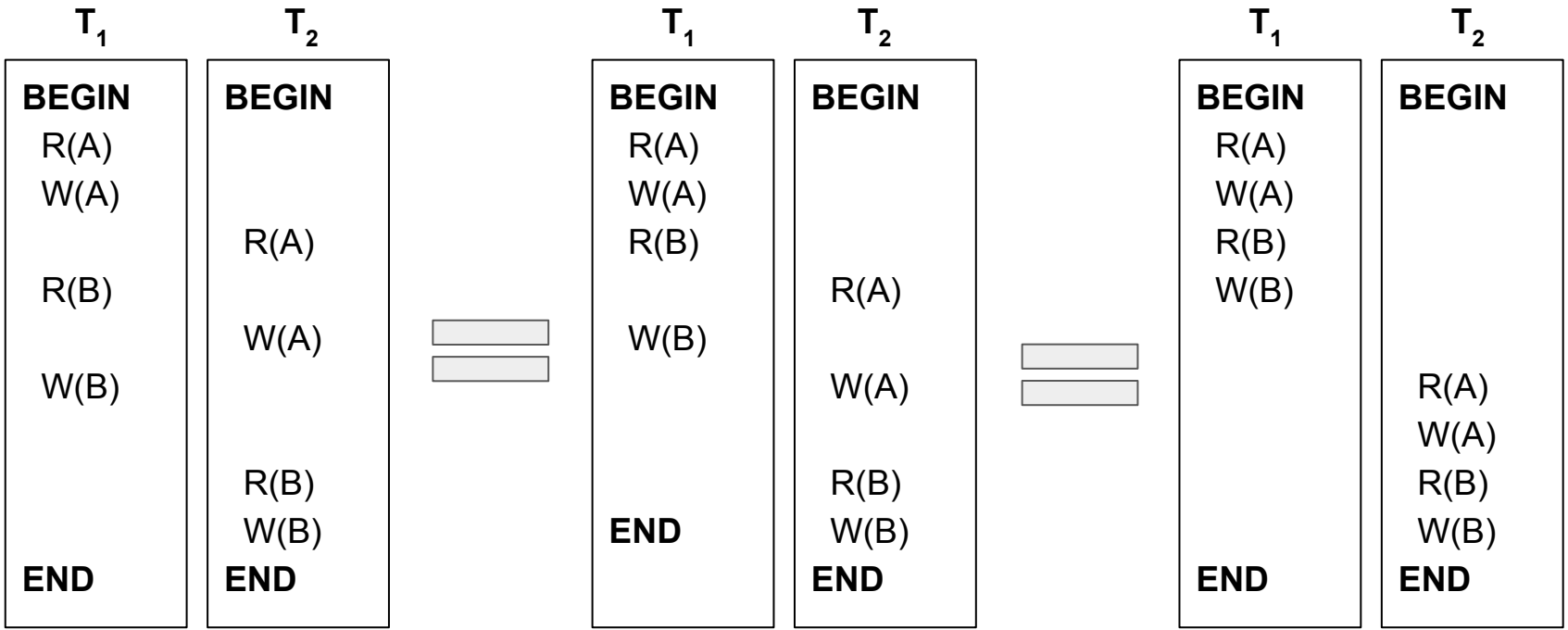
Whatever conflict that they have will be the same



Conflict Equivalence

push up so that we can get some serial execution =>
Serialisable Execution

Serial execution



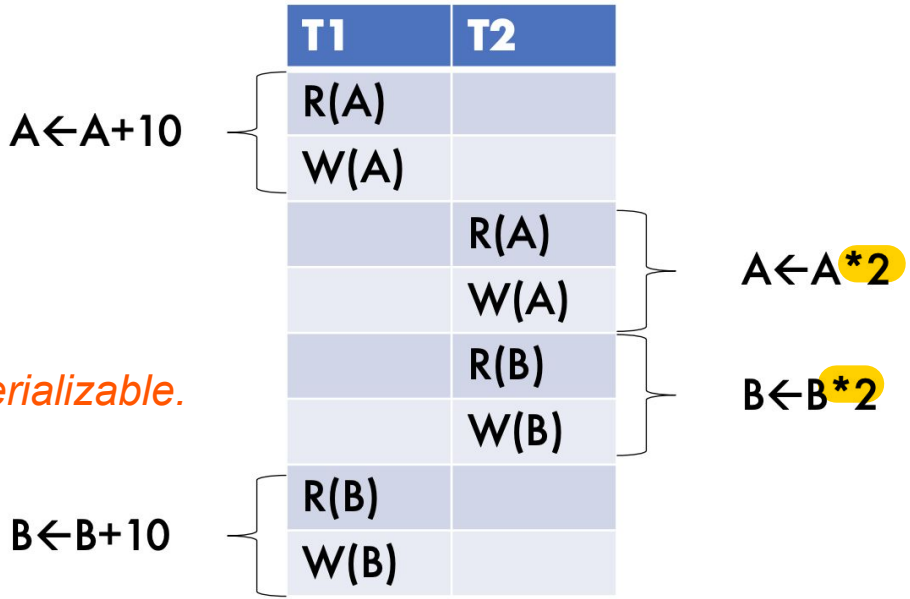
Conflict Serializable

- A sequence X is conflict serializable
  - If it is conflict equivalent to a serial execution
- Intuition:
  - If X can be **transformed** to a serial execution
    - By swapping order of non-conflicting operations

ConflictSerializable(X) → Serializable(X)

Conflict Serializable

- Not conflict serializable
  - Cannot swap R(B) with W(B)
- Not serializable either

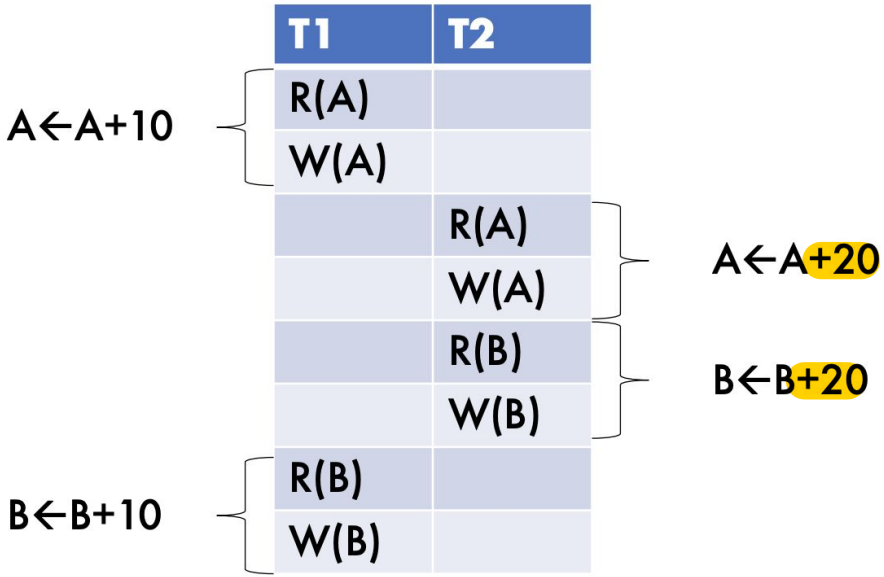


Cannot push R(B) up to make it serializable.
(A+10) x 2 != (2xA) +10

Conflict Serializable

- Not conflict serializable
  - Cannot swap R(B) with W(B)
- BUT serializable

Gives you the correct results
(A+10) + 20

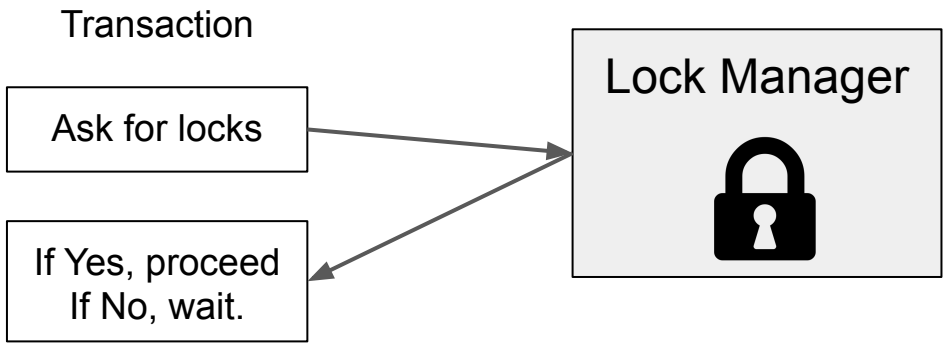


Serializability in Practice

- Two main approaches:
  - Pessimist: prevent non-serializability from happening
    - Locking
  - Optimist: just do it (then fix it later)
    - Optimistic Concurrency Control (OCC): not covered here
- Conflicting goal
  - We also want a lot of concurrency!

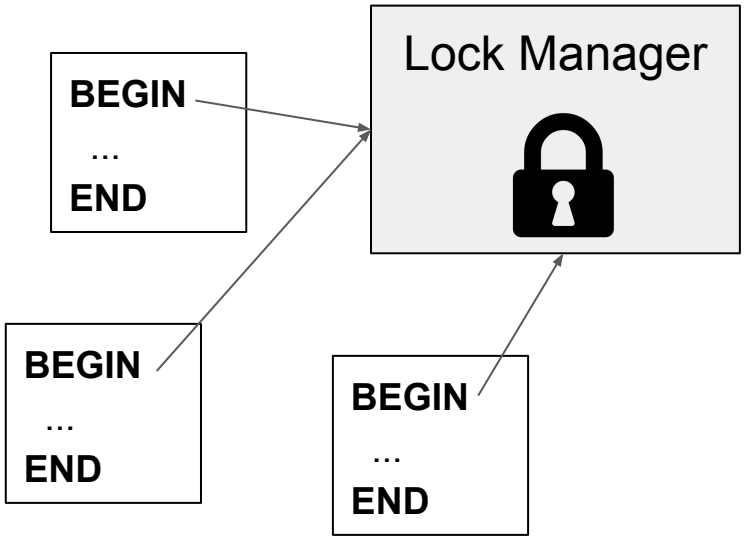
Locking

- You MUST already know what locks are!
- Transactions do not manage locks themselves
  - DBMS has a lock manager



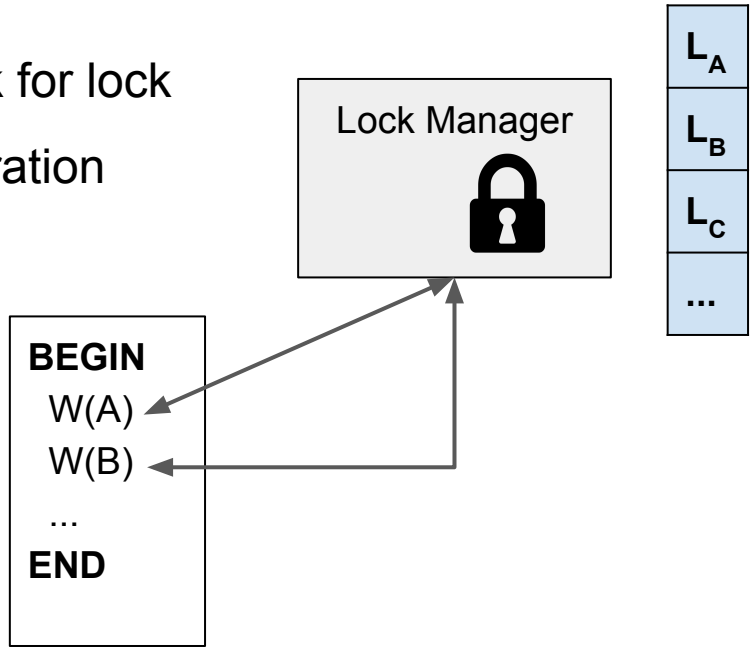
Locking

- Strawman:
  - Use one big, global lock **L**
  - All transactions request for L when starting (at BEGIN)
  - Release at the end (END)
- Serializable 
- Performance 



Locking

- Use more fine-grained locks
  - One lock per object: L_A, L_B, etc.
  - When read or write an object, ask for lock
  - Release when done with the operation
- **Not safe!**

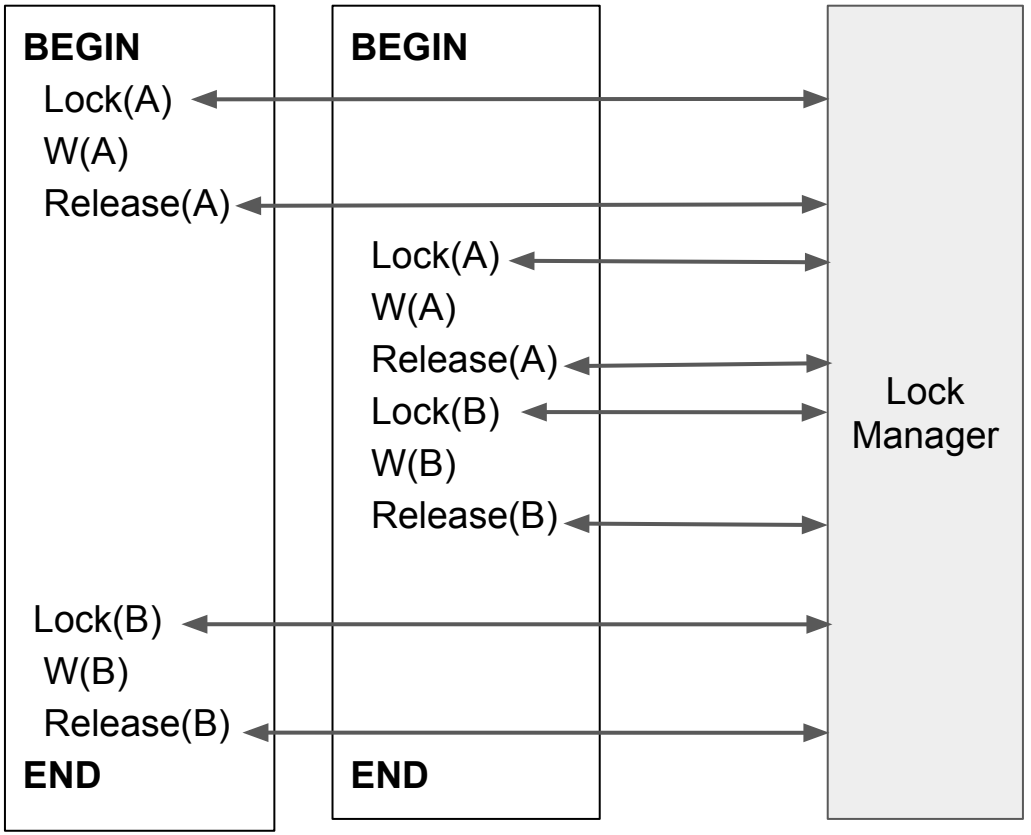


Locking

- **Not safe!**

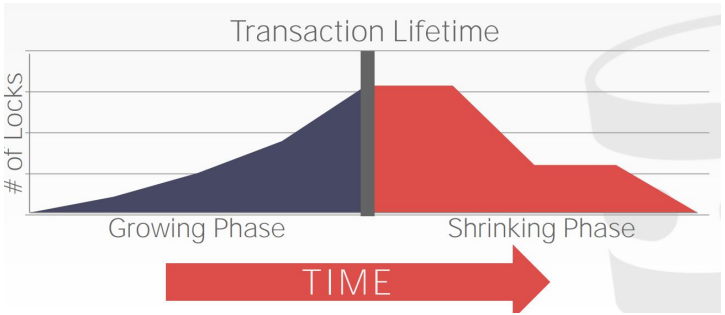
No lock violation here

BUT not serializable!



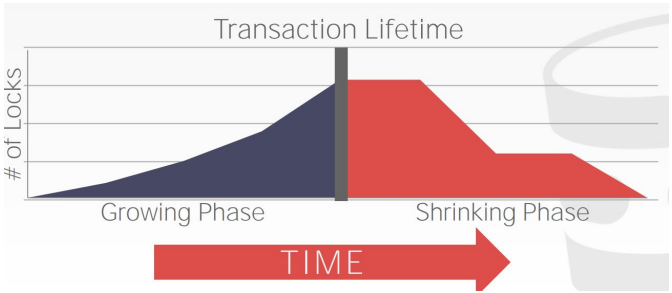
Locking

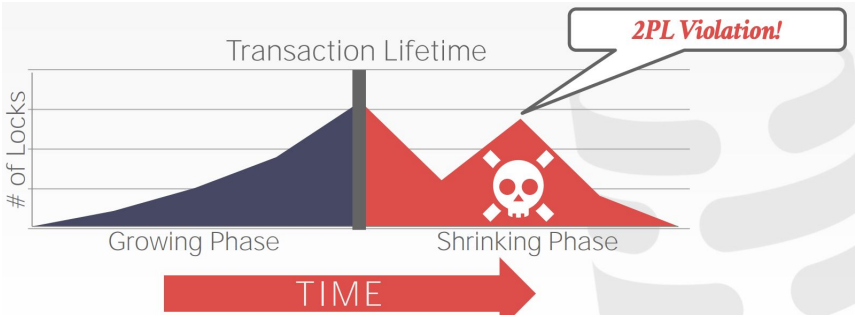
- Problem
  - Locks are released too quick
- Two Phase Locking (2PL):
  - Once release a lock, cannot acquire new one
  - Growing phase: lock acquired
  - Shrinking phase: lock release
    - May not be all at once
    - But cannot acquire new one



2PL

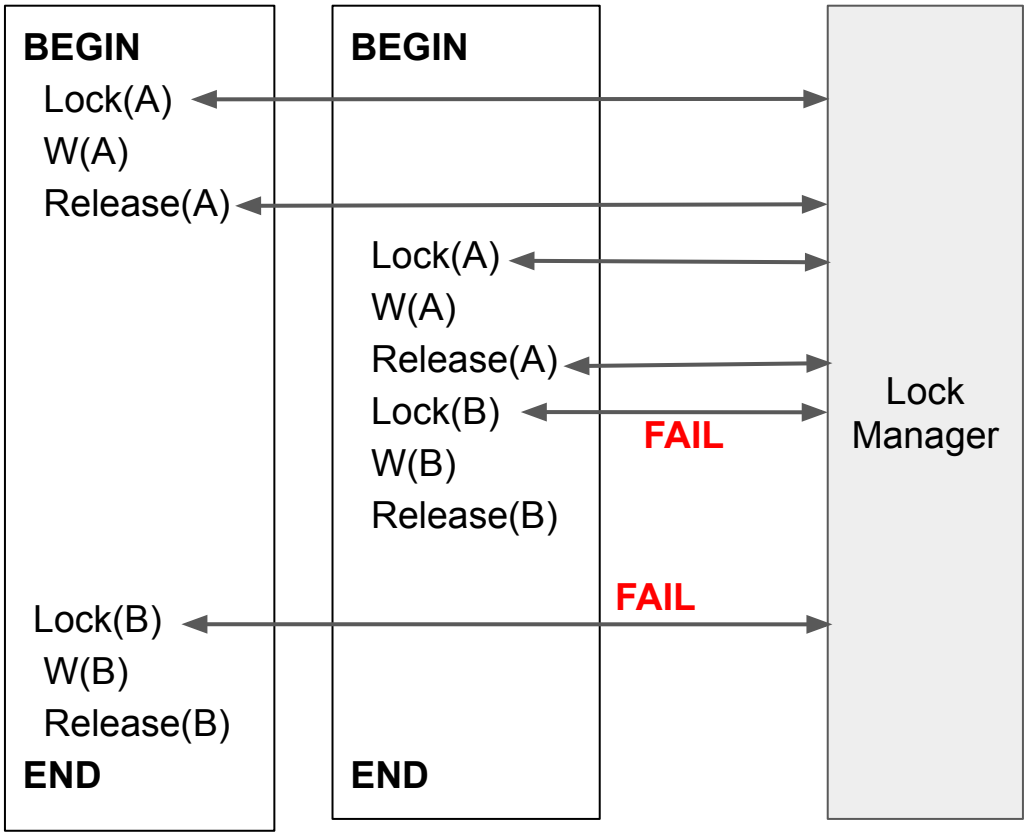
- Two Phase Locking (2PL):
  - Once release a lock, cannot acquire new one





2PL

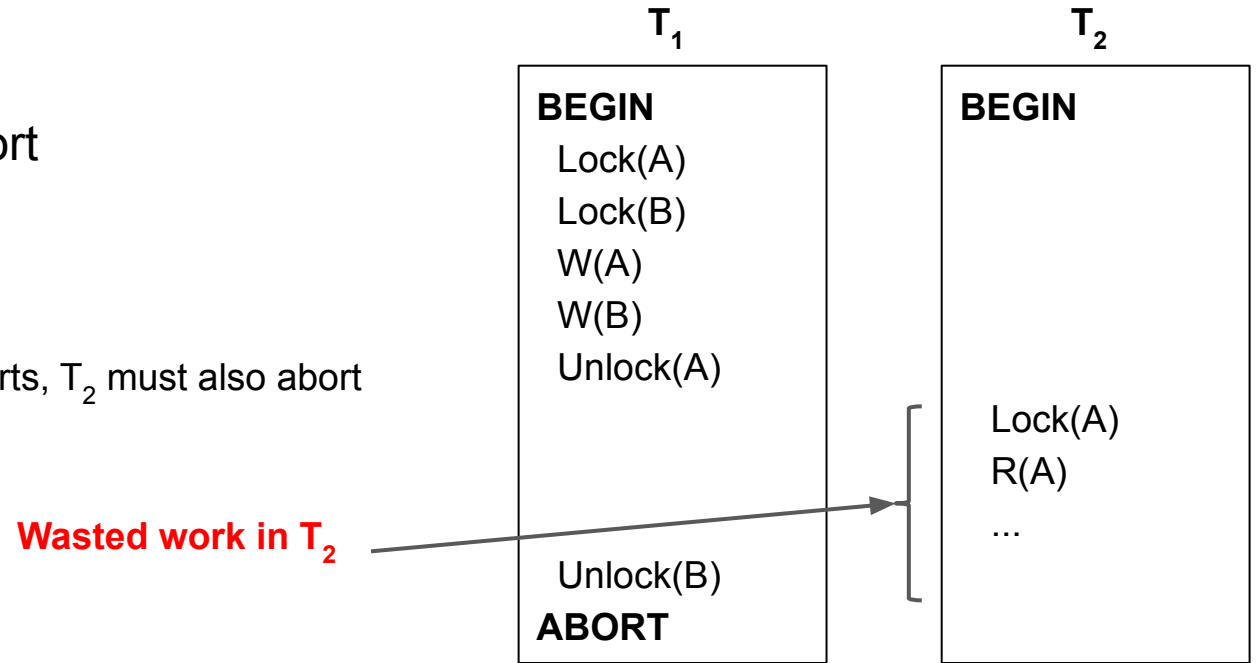
2PL violation



2PL

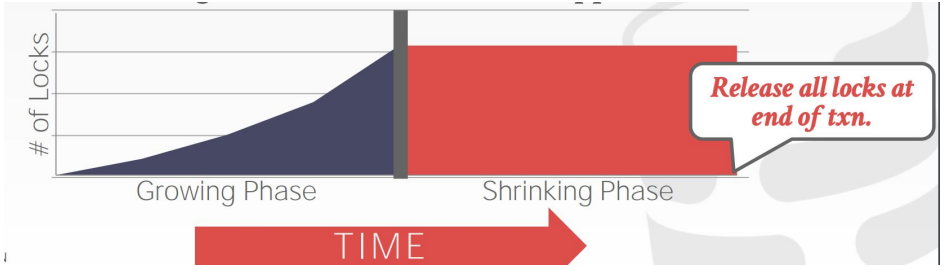
- Two Phase Locking (2PL):
  - User must indicate when the Shrinking phase start
  - Cascading Abort

When T_1 aborts, T_2 must also abort



Strict Two-Phase Locking

- 2PL:
  - Cascading Abort
  - Problem: release too early
- Strict 2PL (S2PL):
  - Only released at the end (COMMIT/ABORT)
  - Scarifying some performance



- ***lock*** *everything you need*
- ***unlock*** *everything at the end*
- *might need different locks for different threads so still got some* ***concurrency***

Why does two phase locking not guard against cascading aborts?
Suppose a transaction locks all its objects then proceeds to invoke them. If it unlocks each object as soon as its invocation is done then that object s state may be seen by other transactions before this one completes. If this one aborts we must abort all such transactions.

In what way does strict two phase locking guard against cascading aborts?
In strict 2PL the transaction does not unlock its objects until it completes with commit or abort. This ensures isolation.

S2PL

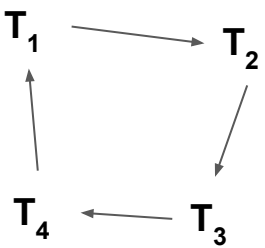
- Not done yet!

Deadlock

T_1	T_2
BEGIN	**BEGIN**
Lock(A)	
	Lock(B)
	Read(B)
Read(A)	
Write(A)	
Lock(B)	
	Lock(A)

S2PL

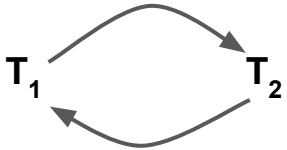
- Deadlock
- Luckily, it can be detected
  - Wait-for graph
- Wait-for graph
  - Nodes are transactions
  - Edge T_i, T_j if T_i waiting for a lock held by T_j

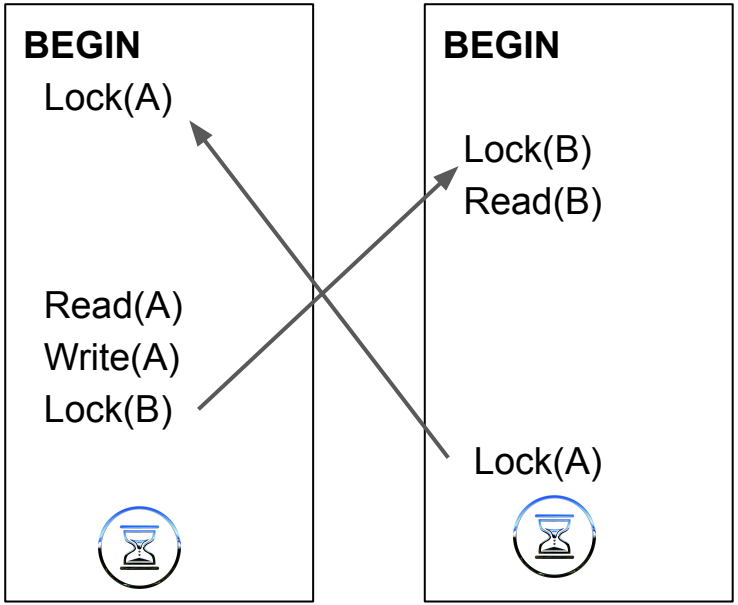


T_1 waits for T_2
T_2 waits for T_3
T_3 waits for T_4
T_4 waits for T_5

Deadlock

- Lock manager maintains wait-for graph
- A loop in the graph = deadlock
- Break deadlock:
  - Pick one transaction
  - Abort it



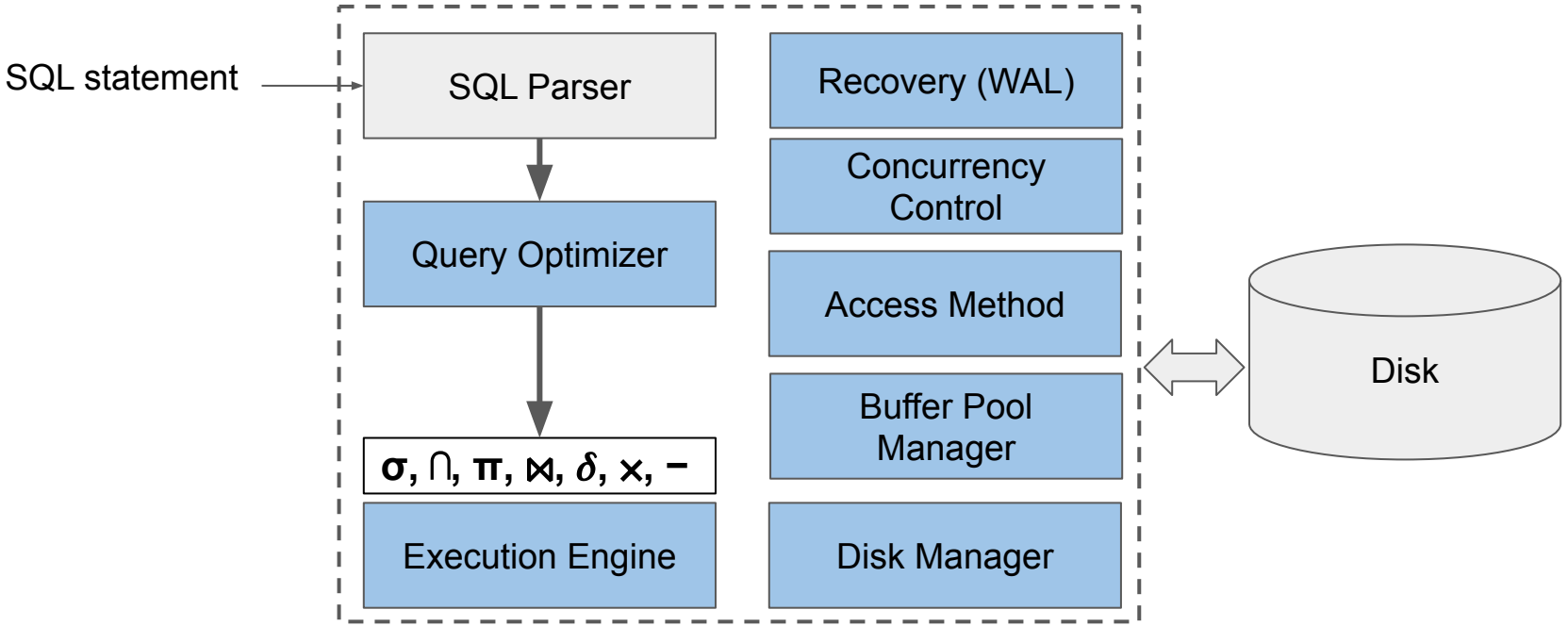


Summary

- Isolation is a hard problem
  - Want safety and performance at the same time
- Serializability is the standard
  - There’re many other, exotic ones
  - Means the interleaved execution has the same effect as a serial execution
- Locking vs. OCC
- Two-phase Locking (2PL), S2PL.

Summary

- Transaction is powerful abstraction
- But supporting it is difficult
  - ACID property
- We have now learned all major components inside a DBMS


SQL statement
SQL Parser
Query Optimizer
σ, ∩, π, ⋈, δ, ×, −
Execution Engine
Recovery (WAL)
Concurrency Control
Access Method
Buffer Pool Manager
Disk Manager
Disk