浙江大学
ZHEJIANG UNIVERSITY

第十二章 事务处理

陶煜波
计算机科学与技术学院

空间数据库回顾

- 地理空间数据库概念，关系代数和SQL　第1-4周
- 几何对象模型与查询　第4-5周
- 空间网络模型与查询　第11-12周
- 空间数据库设计
  - 需求分析 (几何，属性，行为)
  - 概念设计 (扩展的E/R图)　第6-7周
  - 逻辑设计 (将扩展E/R图转换为关系，关系优化)
  - 物理设计 (空间数据库，存储方式，建立索引)　第8-10周
  - 实施　(建表，导入数据，SQL编程，试运行)
  - 运行维护 (转储和恢复，安全性和完整性控制)
    (性能监督、分析改进、重组与重构)　第10,12-14周
- OLAP / NoSQL

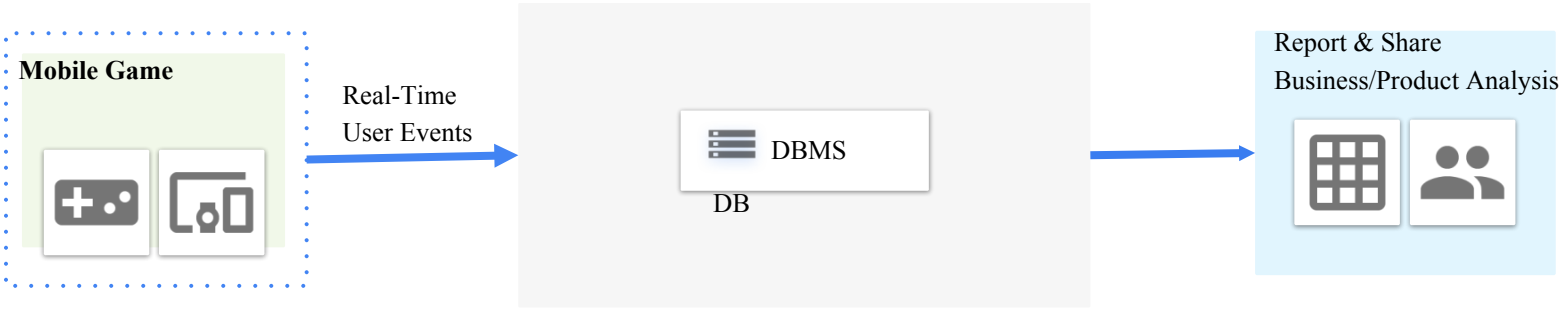
第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions (ACID)
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.1 Transactions

- Game App



Q1: 1000 users/sec?
Q2: Offline?
Q3: Support v1, v1' versions?

App designer

Q7: How to model/evolve game data?
Q8: How to scale to millions of users?
Q9: When machines crash, restore game state gracefully?

Systems designer

Q4: Which user cohorts?
Q5: Next features to build?
Experiments to run?
Q6: Predict ads demand?

Product/Biz designer

- ATM DB
  - Visa does > 60,000 TXNs/sec with users & merchants
- Online shopping
- Signing up for classes

12.1 Transactions

- Basic Definition
  - A transaction ("TXN") is a sequence of one or more *operations* (reads or writes) which reflects *a single real-world transition*
  - Read / Write / Update / Delete / Insert
- In the real world, a TXN either happened completely or not at all
- In "ad-hoc" SQL, each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction

auto-commit

```
START TRANSACTION
      UPDATE Bank SET amount = amount – 100 WHERE name = 'Bob'
      UPDATE Bank SET amount = amount + 100  WHERE name = 'Joe'
COMMIT
ROLLBACK (= ABORT)
```

Motivation for Transactions

- Grouping user actions (reads & writes) into transactions helps with two goals
  - Recovery & Durability: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.
  - Concurrency: Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation for Transactions - R&D

- Recovery & Durability of user data is essential for reliable DBMS usage
  - The DBMS may experience crashes (e.g. power outages, etc.)
  - Individual TXNs may be aborted (e.g. by the user)
- Idea
  - Make sure that TXNs are either durably stored in full, or not at all
  - Keep log to be able to “roll-back” TXNs

Motivation for Transactions - R&D

- What goes wrong?

- Now we'd be fine!

```
Client 1:
        START TRANSACTION
        INSERT INTO SmallProduct(name, price)
                SELECT pname, price
                FROM Product
                WHERE price <= 0.99

        DELETE Product
                WHERE price <=0.99
        COMMIT OR ROLLBACK
```

Motivation for Transactions - C

- Concurrent execution of user programs is essential for good DBMS performance
  - Disk accesses may be frequent and slow – optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
  - Users should still be able to execute TXNs as if in isolation and such consistency is maintained
- Idea
  - Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming …

Motivation for Transactions - C

- Two managers attempt to discount products *concurrently* – What could go wrong?

```
Client 1: UPDATE Product
              SET Price = Price – 1.99
              WHERE pname = 'Gizmo'

Client 2: UPDATE Product
              SET Price = Price * 0.5
              WHERE pname = 'Gizmo'
```

```
Client 1: START TRANSACTION
              UPDATE Product
              SET Price = Price – 1.99
              WHERE pname = 'Gizmo'
            COMMIT

Client 2: START TRANSACTION
              UPDATE Product
              SET Price = Price * 0.5
              WHERE pname = 'Gizmo'
            COMMIT
```

第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions (ACID)
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.2 Properties of Transactions

- 事务的四个特性 (ACID)
  - 原子性 (Atomicity)
    - State shows either all the effects of txn, or none of them
  - 一致性 (Consistency)
    - Txn moves from a state where integrity holds, to another where integrity holds
  - 隔离性 (Isolation)
    - Effect of txns is the same as txns running one after another (ie looks like batch mode)
  - 持久性 (Durability)
    - Once a txn has committed, its effects remain in the database

ACID: Atomicity

- TXN's activities are atomic: all or nothing
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*

- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

ACID: Consistency

- The tables must always satisfy user-specified *integrity constraints*
  - Examples
    - Account number is unique
    - Stock amount can't be negative
    - Sum of debits and of credits is 0
- How consistency is achieved?
  - Programmer writes a TXN to go from one consistent state to a consistent state
  - *System* makes sure that the TXN is atomic
- Can defer checking the validity of constraints until the end of a transaction

ACID: Isolation

- A transaction executes concurrently with other transactions

- Isolation: the effect is as if each transaction executes in isolation of the others
  - E.g. Should not be able to observe changes from other transactions during the run

ACID: Durability

- The effect of a TXN must continue to exist ("persist") after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…
- Means: Write data to disk

Change on the horizon?
Non-Volatile Ram (NVRam).
Byte addressable.

Challenges for ACID properties

- In spite of Power failures (not media failures)
- Users may abort the program: need to “rollback changes”
  - Need to *log* what happened
- Many users execute concurrently
  - Can be solved via locking

And all this with… Performance!!

ACID实现

- 事务的这四个特性一般简称为事务的ACID特性。保证事务的ACID特性是事务处理的重要任务其中
  - 事务的原子性和持久性由DBMS系统的恢复机制来保证
  - 事务的隔离性是由DBMS系统的并发控制机制实现的
  - 事务的一致性是由事务管理机制的综合机制包括并发控制机制和恢复机制共同保证的。当然前提是用户在定义事务的时候，事务逻辑是准确的

ACID实现

- 假设关系R(A)包含元组{(5), (6)}，两个事务

T1：Update R set A = A + 1

T2：Update R set A = A * 2

假设这两个事务同时满足隔离性和原子性，下列哪个不可能是关系R的最终状态？

A. (10, 12)　B. (11, 13)　C. (11, 12)　D. (12, 14)

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**
- Some “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs…

ACID is an extremely important & successful paradigm, but still debated!

第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions (ACID)
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.3 Concurrency

- Concurrency: Isolation & Consistency

- DBMS is responsible for concurrency so that…
  - **Isolation** is maintained: Users must be able to execute each TXN as if they were the only user
  - **Consistency** is maintained: TXNs must leave the DB in a consistent state
- 假设用户C1依次执行事务T1和T2，用户C2同时依次执行事务T3和T4，存在多少种等价的事务串行调度？
- A schedule is a sequence of interleaved actions from all transactions

Example – Consider two TXNs

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = 'A'

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = 'B'
    COMMIT
```

Each action in the TXNs *reads a value from global memory* and then *writes one back to it*

```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
    COMMIT
```

T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

Note:

1. DB does not care if T1 → T2 or T2 → T1 (which TXN executes first)
2. If developer does, what can they do? (Put T1 and T2 inside 1 TXN)

Goal for scheduling transactions:

- Interleave transactions to boost performance
- Data stays in a good state after commits and/or aborts (ACID)

Example – Consider two TXNs

Serial schedule T_1, T_2:

Starting Balance

A	B
$50	$200

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

Time

Interleaved schedule A:

Same result!

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

We call the particular order of interleaving a **schedule**

Example – Consider two TXNs

Serial schedule T_1, T_2:

Starting Balance

A	B
$50	$200

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

Time

Interleaved schedule B:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$112

Time

Different result than serial T_1, T_2!

Example – Consider two TXNs

Serial schedule T_2, T_1:

Starting Balance

A	B
$50	$200

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$153	$112

Time

Interleaved schedule B:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$159	$112

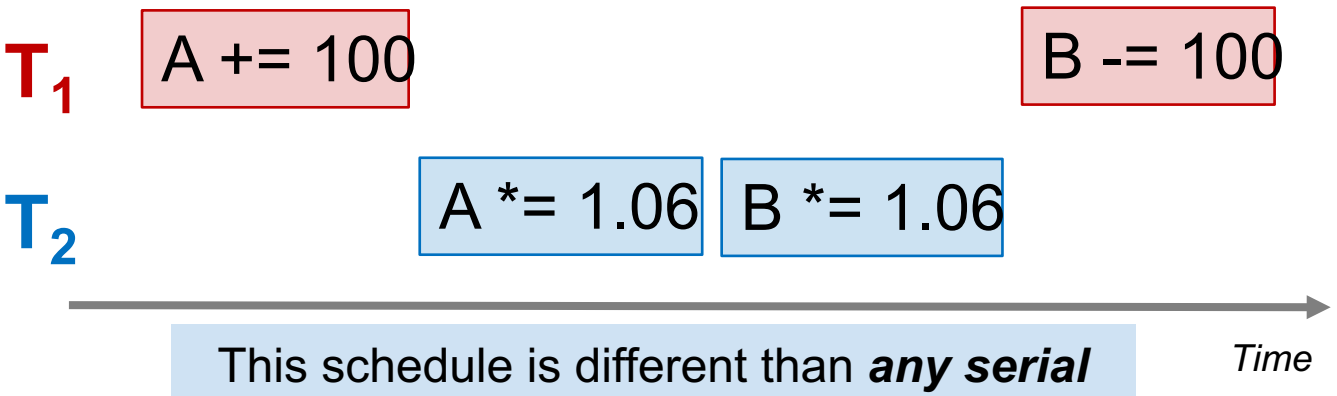
Time

Different result than serial T_2, T_1!

Interleaving & Isolation

- The DBMS has freedom to interleave TXNs
- However, it must pick an interleaving or schedule such that isolation and consistency are maintained
- ⇒ Must be as if the TXNs had executed serially!

Interleaved schedule B:



This schedule is different than ***any serial order!*** We say that it is **not serializable**

DBMS must pick a schedule which maintains isolation & consistency

"With great power comes great responsibility"

Scheduling Definitions

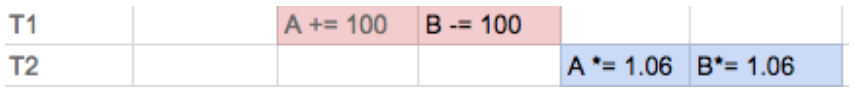
- A **serial schedule** (串行调度) is one that does not interleave the actions of different transactions

- A and B are **equivalent schedules** if, **for any database state**, the effect on DB of executing A **is identical to** the effect of executing B

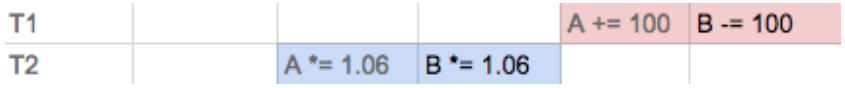
- A **serializable schedule** (可串行化调度) is a schedule that is equivalent to ***some*** serial execution of the transactions
  - The word “some” makes this definition powerful & tricky

Scheduling Definitions

Serial Schedules

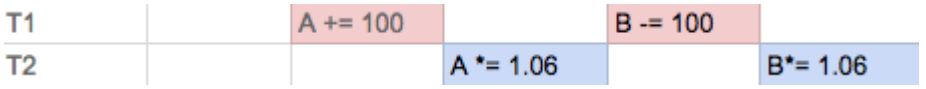


S1

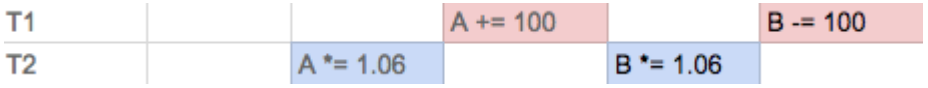


S2

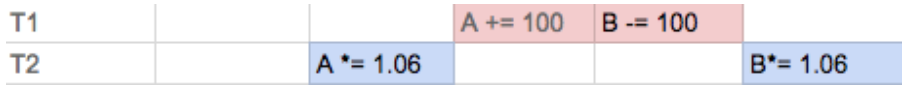
Interleaved Schedules



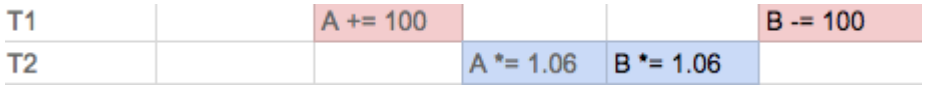
S3



S4



S5



S6

Serial Schedules	S1, S2
Serializable Schedules	S3, S4 (And S1, S2)
Equivalent Schedules	<S1, S3>, <S2, S4>
Non-serializable (Bad) Schedules	S5, S6

Conflicts and Anomalies

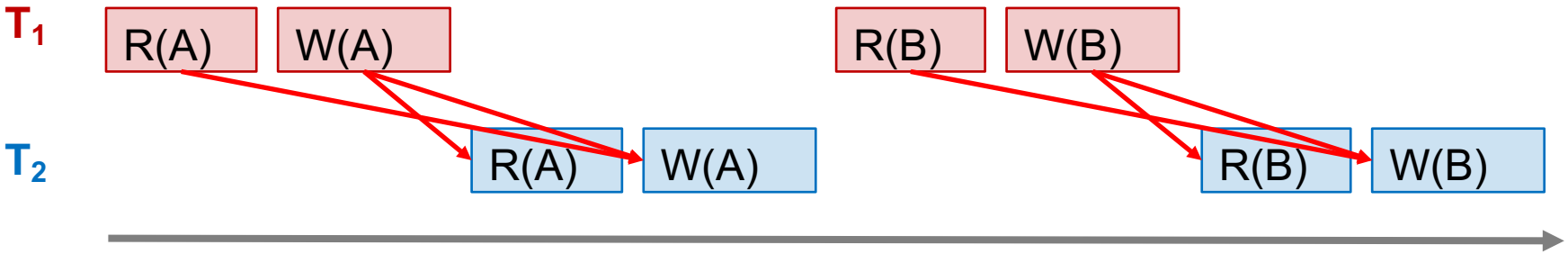
- Two actions **conflict** (冲突) if they are part of different TXNs, involve the same variable, and at least one of them is a write
  - Swapping will change program behavior
  - Two actions by same transaction T_i: $R_i(X)$, $W_i(Y)$
  - Two writes by T_i, T_j to same element: $W_i(X)$, $W_j(X)$
  - Read/write by T_i, T_j to same element: $W_i(X)$, $R_j(X)$/ $R_i(X)$, $W_j(X)$/
- Thus, there are three types of conflict
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

Why no "RR Conflict"?

Note: **conflicts** happen often in many real world transactions. (E.g., two people trying to book an airline ticket)

Conflicts and Anomalies

- Two actions **conflict** (冲突) if they are part of different TXNs, involve the same variable, and at least one of them is a write



All “conflicts”!

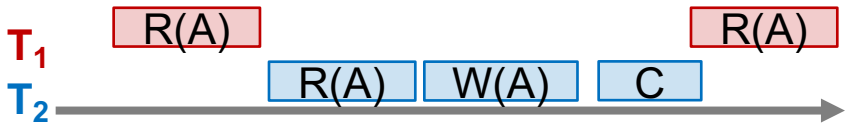
- Conflicts are in both “good” and “bad” schedules (they are a property of transactions)

Goal: Avoid Anomalies while interleaving transactions with conflicts!

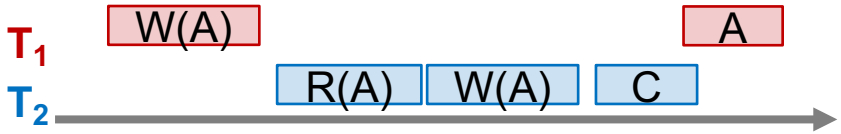
- Do not create “bad” schedules where isolation and/or consistency is broken (i.e., Anomalies)

Classic Anomalies with Interleaved Execution

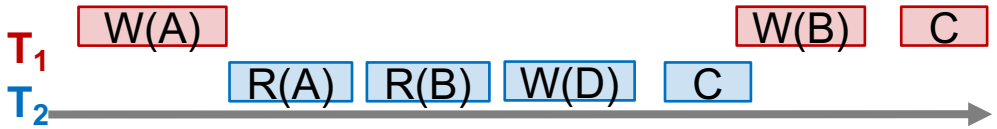
“Unrepeatable read”:



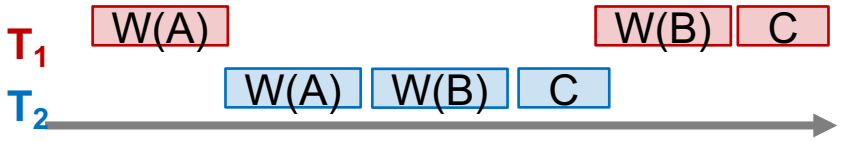
“Dirty read” / Reading uncommitted data:



“Inconsistent read” / Reading partial commits:



Partially-lost update:



第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

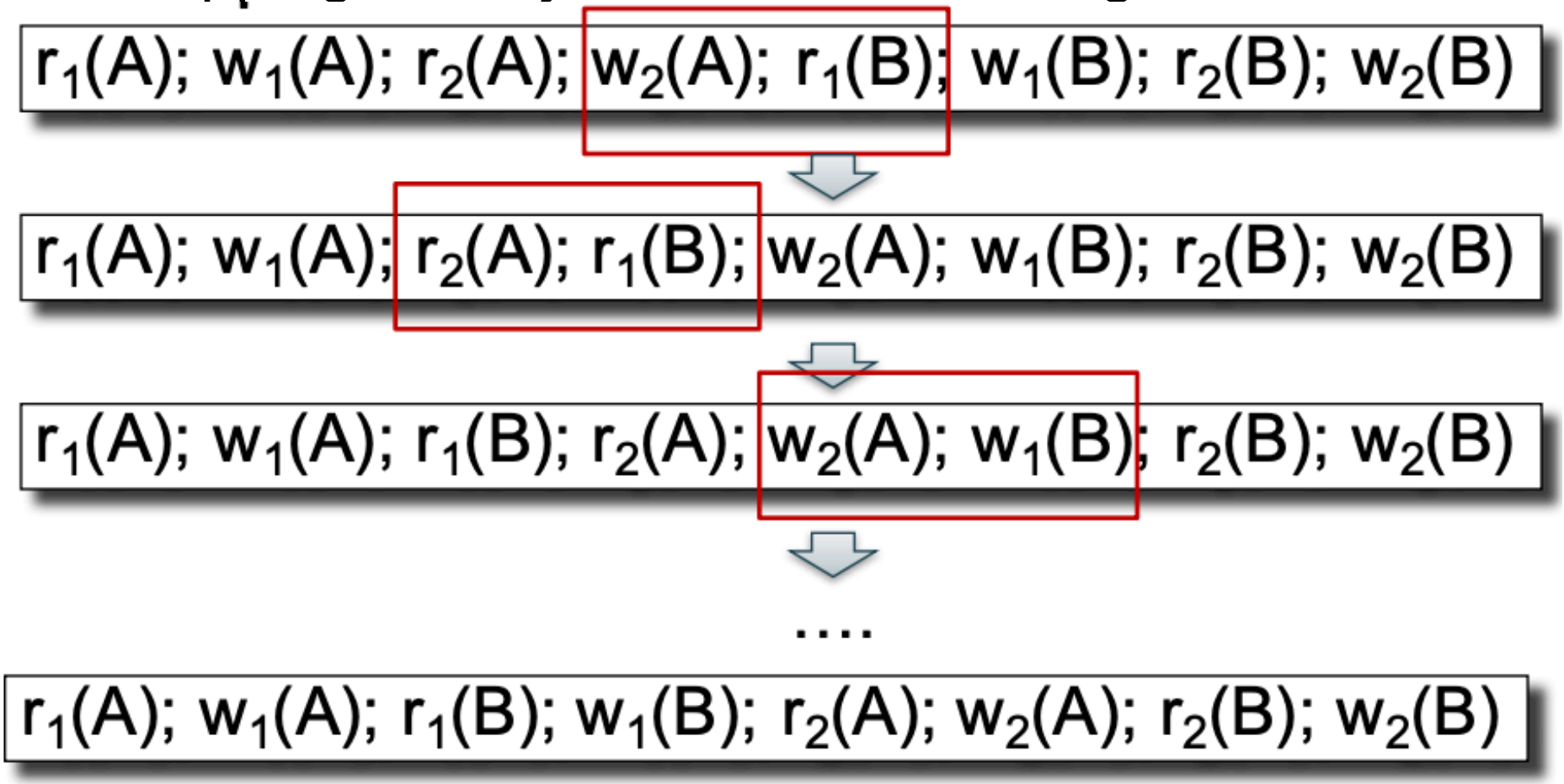
12.4 Conflict Serializability

- Two schedules are **conflict equivalent (冲突等价)** if
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is conflict serializable (冲突可串行化) if S is conflict equivalent to some serial schedule

Conflict serializability ⇒ Serializable
So if we have conflict serializable, we have consistency & isolation!

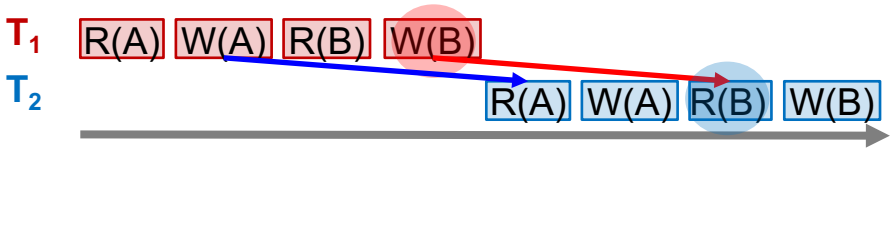
12.4 Conflict Serializability

- A schedule is **conflict serializable** if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

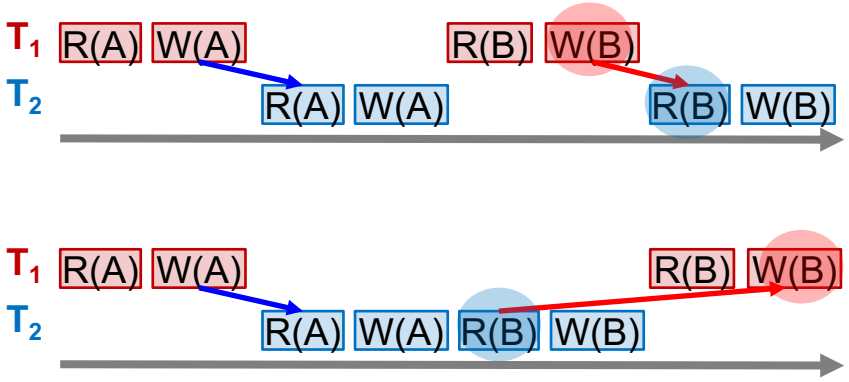


Example “Good” vs. “Bad” Sched.

Serial Schedule:



Interleaved Schedules:



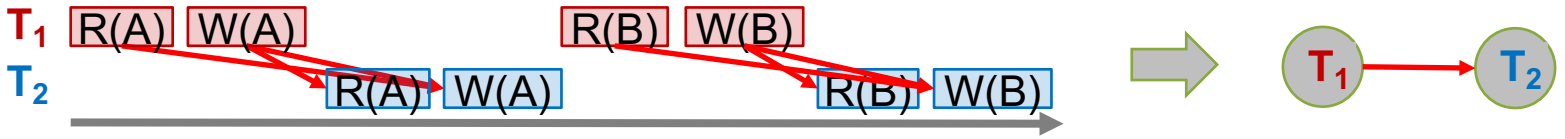
Note that in the “bad” schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

Conflict serializability provides us with an operative notion of “good” vs. “bad” schedules! “Bad” schedules create data Anomalies

The Conflict Graph

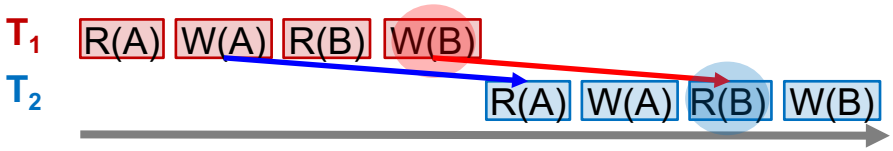
- Let's now consider looking at conflicts at **the TXN level**

- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i precede and conflict with any actions in T_j**

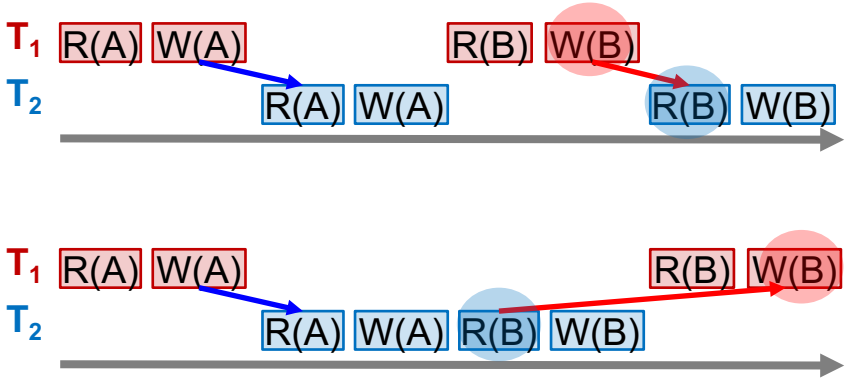


What can we say about "good" vs. "bad" conflict graphs?

Serial Schedule:

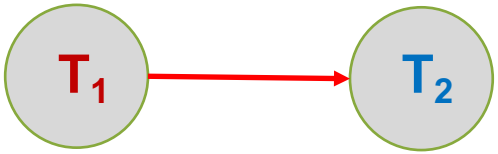


Interleaved Schedules:

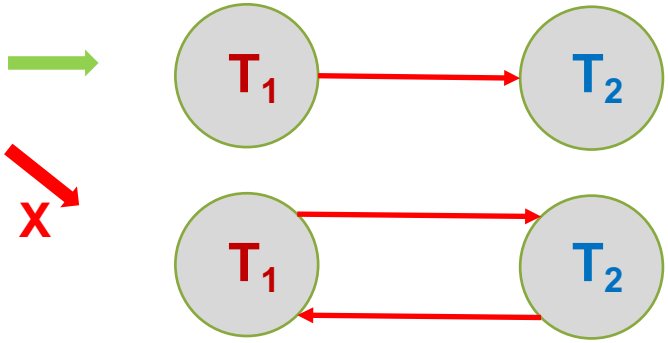


A bit complicated…

Serial Schedule:



Interleaved Schedules:

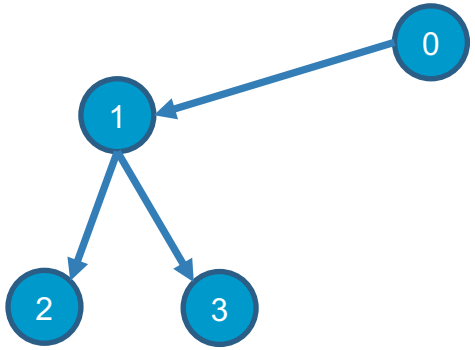


Simple!

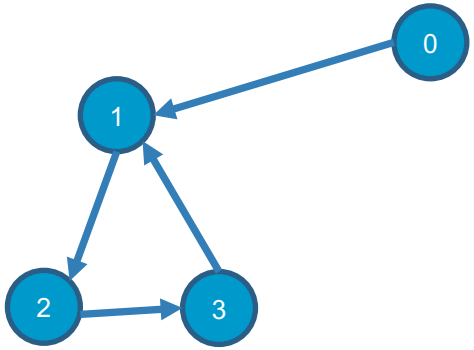
Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

DAGs & Topological Ordering

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A directed **acyclic** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering if and only if there are no directed cycles)



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)



There is none!

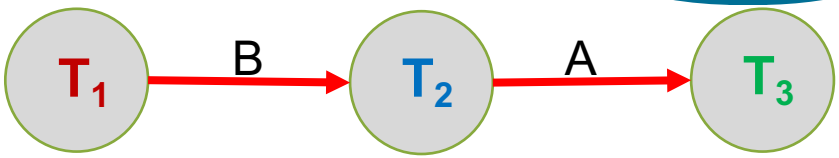
Connection to Conflict Serializability

- In the conflict graph, a topological ordering of nodes corresponds to **a serial ordering of TXNs**
- Thus an **acyclic** conflict graph → conflict serializable!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

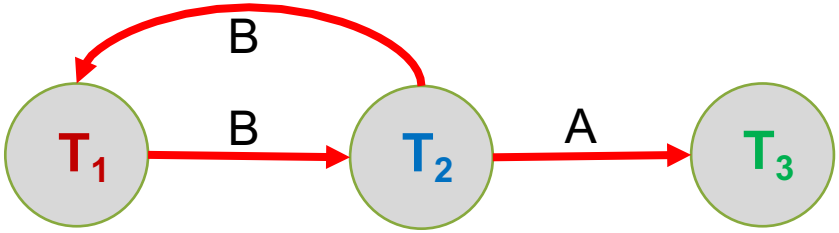
Example with 2 Transactions

- Example 1
  - $R_2(A), R_1(B), W_2(A), R_3(A), W_1(B), W_3(A), R_2(B), W_2(B)$



Conflict Serializable

- Example 2
  - $R_2(A), R_1(B), W_2(A), R_2(B), R_3(A), W_1(B), W_3(A), W_2(B)$



NOT Conflict Serializable

Example with 2-3 Transactions

- Example 3
  - $R_1(A), R_2(B), R_1(B), W_2(B), W_1(A), W_1(B), R_2(A), W_2(A)$
- Example 4
  - $R_1(A), R_2(B), R_3(A), R_2(A), R_3(C), R_1(B), R_3(B), R_1(C), R_2(C)$
- Example 5
  - $W_1(A), W_2(A), W_1(B), W_3(B), W_1(C), W_3(C), W_2(C)$

Example with 5 Transactions

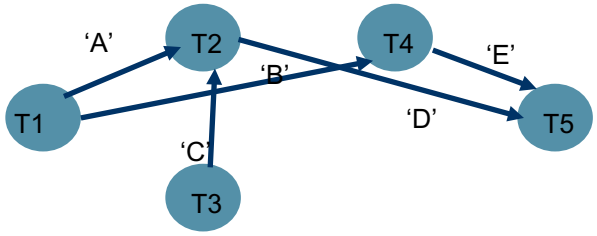
Good or Bad schedule?
Conflict serializable?

Schedule S1

	w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)
T1	w1(A)		w1(B)							
T2		r2(A)			r2(C)		w2(D)			
T3				w3(C)						
T4						r4(B)		w4(E)		
T5									r5(D)	w5(E)

Step1
Find conflicts
(RW, WW, WR)

Step2
Build Conflict graph
Acyclic? Topo Sort



Acyclic
⇒ Conflict serializable!
⇒ Serializable

Step3
Example serial schedules
Conflict Equiv to S1

Serial Sched (SS1)

T3	T1	T4	T2	T5				
w3(C)	w1(A)	w1(B)	r4(B)	w4(E)	r2(A)	r2(A)	w2(D)	r5(D)	w5(E)

Serial Sched (SS2)

T1	T3	T2	T4	T5				
w1(A)	w1(B)	w3(C)	r2(A)	r2(A)	w2(D)	r4(B)	w4(E)	r5(D)	w5(E)

Conflict Serializability Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation & consistency** are maintained
  - We formalized a notion of **serializability** that captured such a “good” interleaving schedule
- We defined **conflict serializability**

- Scheduler = the module that schedules the transaction’s actions, ensuring serializability
  - Concurrency Control Manager

Conflict Serializability Summary

- Major differences between database vendors
  - Locking Scheduler
    - Aka “pessimistic concurrency control”
    - Each element has a unique lock
    - Each transaction must first acquire the lock before reading/writing that element
    - If the lock is taken by another transaction, then wait
    - The transaction must release the lock(s)
    - SQLite, SQL Server, DB2
  - Multiversion Concurrency Control (MVCC)
    - Aka “optimistic concurrency control”
    - Postgres, Oracle https://www.postgresql.org/docs/current/mvcc.html

By using lock scheduler ensures conflict-serializability

第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.5 Two-Phase Locking

- Algorithm: *strict two-phase locking* - as a way to deal with concurrency
  - Guarantees conflict serializability
  - Also (*conceptually*) straightforward to implement, and transparent to the user

Locks

TXNs obtain

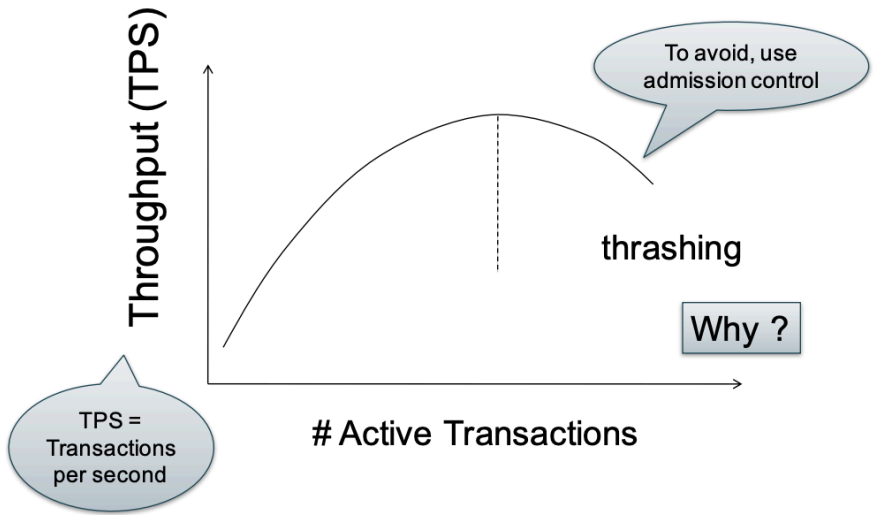
- An X (exclusive) lock on object before writing
  - If a TXN holds, no other TXN can get a lock (S or X) on that object
- An S (shared) lock on object before reading
  - If a TXN holds, no other TXN can get an X lock on that object
- All locks held by a TXN are released when TXN completes

Note: Terminology here- “exclusive”, “shared” - meant to be intuitive - no tricks!

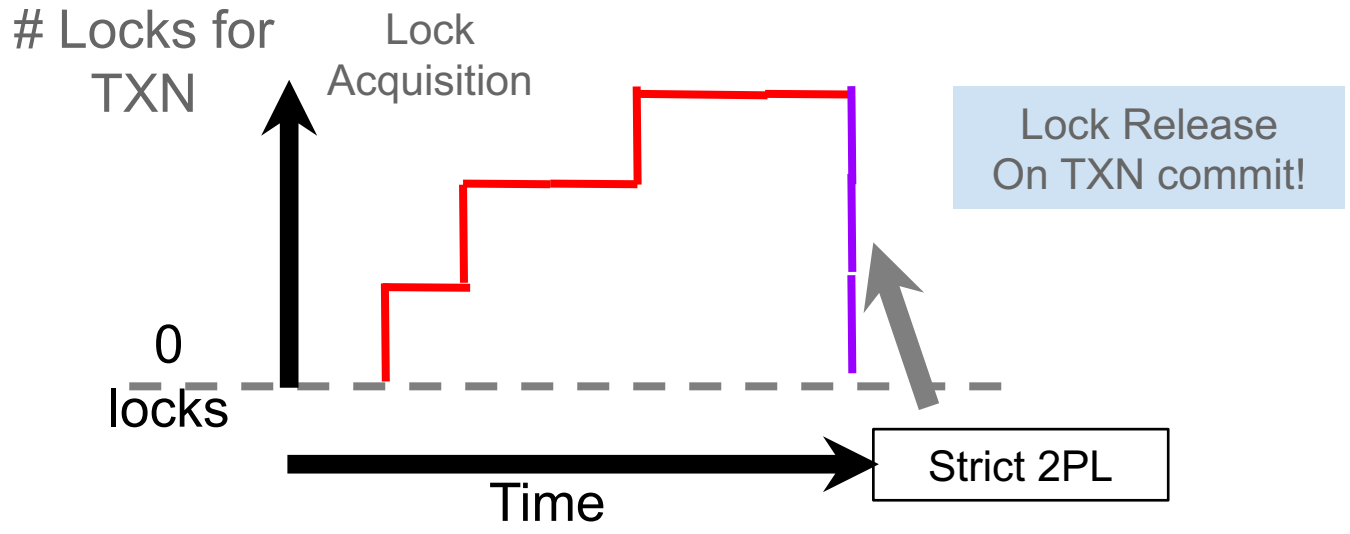
T_1 \ T_2	X	S	
X	N	N	Y
S	N	Y	Y
	Y	Y	Y

Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g., SQL Server
- Coarse grain locking (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g., SQLite
- Solution
  - Lock escalation changes granularity as needed



2-Phase Locking (2PL)



2PL: A transaction can not request additional locks once it releases any locks. Thus, there is a “growing phase” followed by a “shrinking phase”

Strict 2PL: Release locks only at COMMIT (COMMIT Record flushed) or ABORT

Strict 2PL

- If a schedule follows strict 2PL, it is **conflict serializable** …
  - …and thus serializable
  - …and we get isolation & consistency
  - …and we get recoverable

- Popular implementation
  - Simple!
  - Produces subset of *all* conflict serializable schedules

Example: Recoverable

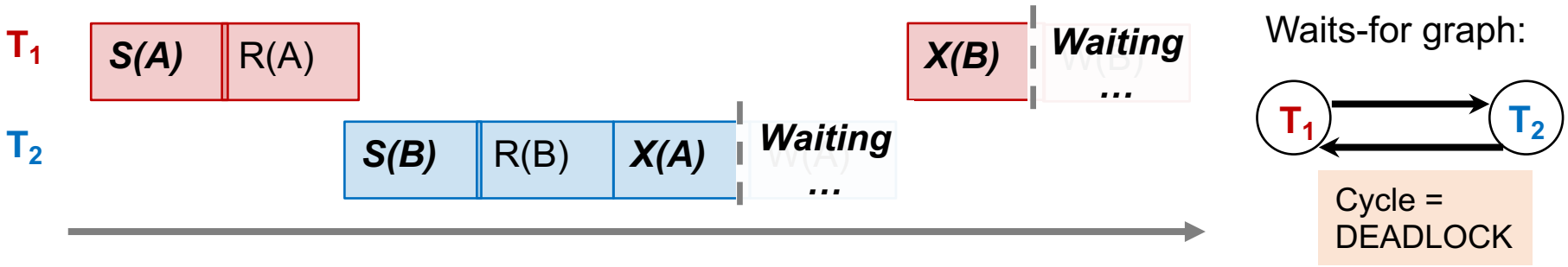
- 2PL: Conflict serializable (L = Lock, U = Unlock)

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;

Rollback Non-recoverable Schedule

Example: Deadlock Detection

- Deadlock
  - First, T1 requests a shared lock on A to read from it
  - Next, T_2 requests a shared lock on B to read from it
  - T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**
  - Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**



Waits-For graph: Track which Transactions are waiting
IMPORTANT: WAITS-FOR graph different than CONFLICT graph we learnt earlier !

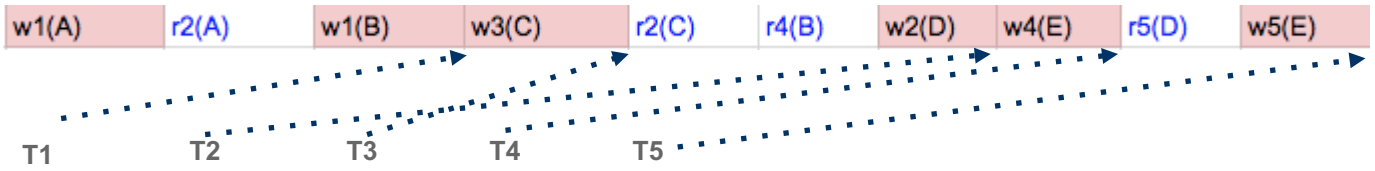
Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other
- Two ways of dealing with deadlocks
  - Deadlock prevention
  - Deadlock detection
    - Create the waits-for graph
      - Nodes are transactions
      - There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*
    - Periodically check for (and break) cycles in the waits-for graphs

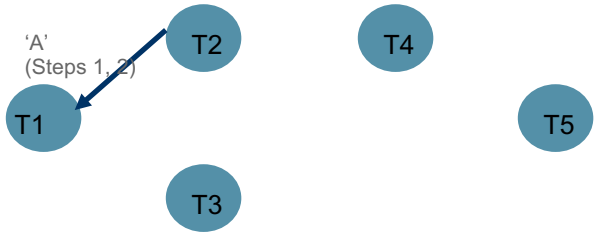
Example with 5 Transactions (2PL)

Schedule S1

Execute with 2PL



	T1	T2	T3	T4	T5
Step 0	X (A) w1(A)				
Step 1		Req S(A)			
Step 2	X (B) w1(B) Unl B, A				
Step 3		Get S(A) r2(A)			
Step 4			X (C) w3(C) Unl C		
Step 5		S(C) r2(C)			
Step 6				S(B) r4(B)	
Step 7		X(D) w2(D) Unl A, C, D			
Step 8				X(E) w4(E) Unl B,E	
Step 9					S (D) r5(D)
Step 10					X (E) w5(E) Unl D, E



Waits- For Graph

2PL Summary

- Locking allows only conflict serializable schedules
  - If the schedule completes … (it may deadlock!)
- Major differences between vendors
  - Lock on the entire database
    - SQLite
      - There is only one exclusive lock, thus, never deadlocks
  - Lock on individual records
    - SQL Server, DB2, etc
      - Checks periodically for deadlocks and aborts on TXN

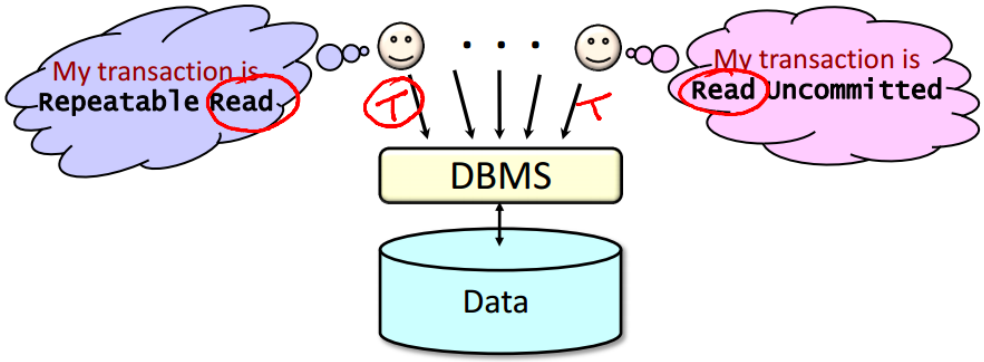
第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.6 Isolation Levels

- Serializability
  - Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions
- Trade of Overhead vs. Reduction in concurrency
  - Read Uncommitted
  - Read Committed
  - Repeatable Read
  - Serializable

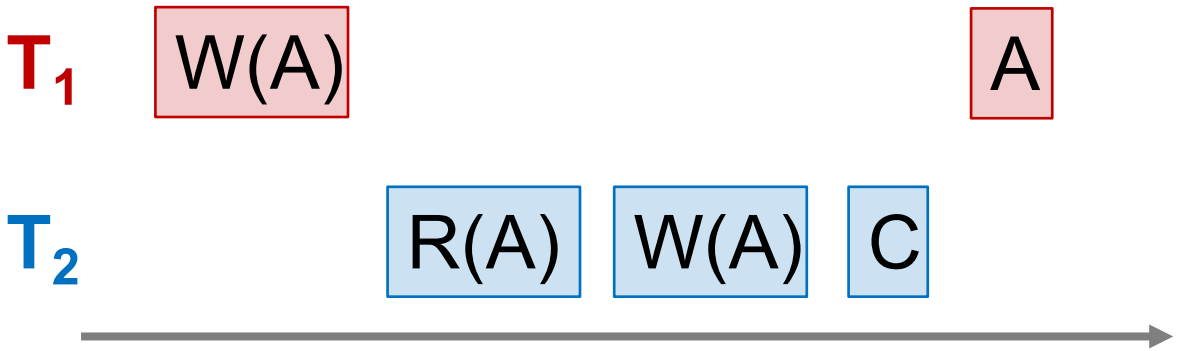


- Per transaction
  - “In the eye of the beholder”

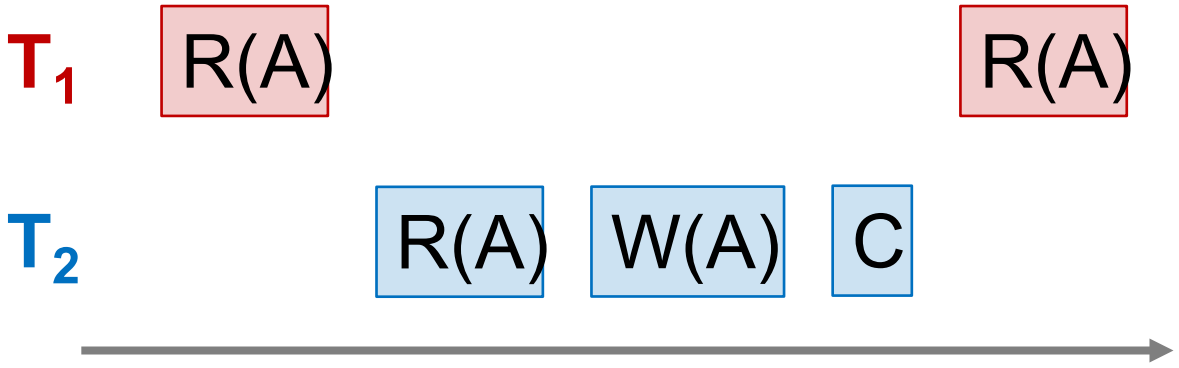
PostgreSQL Isolation Levels: http://www.postgresql.org/docs/current/static/mvcc.html

Inconsistency

- Dirty reads
  - "dirty" data item: written by an uncommitted transaction



- Unrepeatable reads
  - When reading same element twice, may get two different values



Inconsistency

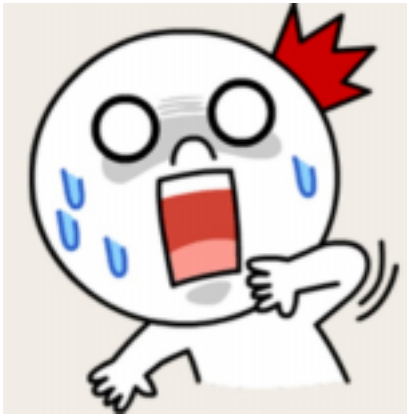
- Phantom rows (幻行)
  - Update (static) → Insert / Delete (dynamic)

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

  - Suppose there are two blue products, A1 and A2
  - $R_1(A1), R_1(A2), W_2(A3), R_1(A1), R_1(A2), R_1(A3)$
  - Conflict serializable

Inconsistency

- A “phantom” is a tuple that is invisible during part of a transaction execution but visible during the entire execution
- In our example (tuple lock)
  - T1: read list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears
- Dealing with phantoms
  - Lock the entire table
  - Lock the index entry for ‘blue’
    - If index is available
  - Or user predicate locks
    - A lock on an arbitrary predicate



Isolation Levels

- Read Uncommitted
  - A transaction may perform dirty and inconsistent reads
  - T1 [Serializable]

  Update Student Set GPA = GPA * 1.1 Where ….
  - T2 [Read Uncommitted]

  Set Transaction Isolation Level Read Uncommitted

  Select Avg(GPA) From Student
  - Partial GPA has updated and others not [Not Serializable]

Isolation Levels

- 关系R(A)包含元组{(1), (2)}，执行以下事务：
- T1：update R set A = 2 *A;
- T2：select avg(A) from R;
- 如果T2以read uncommitted执行，可能的结果是？A

A. 1.5, 2, 2.5, 3　　B. 1.5, 2, 3

C. 1.5, 2.5, 3　　D. 1.5, 3

Isolation Levels

- Read Committed
  - A transaction may not perform dirty reads
  - A transaction may perform inconsistent reads
  - Still does not guarantee global serializability
  - T1 [Serializable]

  A: Update Student Set GPA = GPA * 1.1 Where ….
  - T2 [Read Committed]

  Set Transaction Isolation Level Read Committed

  B: Select Avg(GPA) From Student

  C: Select Avg(GPA) From Student
  - SQL Statement Order: B A C [Not Serializable]

Isolation Levels

- 关系R(A)和S(B)都包含元组{(1), (2)}，执行以下事务：
- T1：update R set A = 2 *A;
  update S set B = 2 * B;
- T2：select avg(A) from R;
  select avg(B) from S
- 如果T2以read committed执行，可能的结果?
- (1.5, 1.5), (3, 3), (1.5, 3)

Isolation Levels

- Repeatable Read
  - A transaction may not perform dirty reads
  - An item read multiple times cannot change value (inconsistent reads)
  - Still does not guarantee global serializability
  - T1 [Serializable]

A: Update Student Set GPA = GPA * 1.1

B: Update Student Set SizeHS = 1500 Where SID = 123;

  - T2 [Repeatable Read]

Set Transaction Isolation Level Repeatable Read

C: Select Avg(GPA) From Student

D: Select Avg(SizeHS) From Student

  - SQL Statement Order: C A B D [Not Serializable]可以?

Isolation Levels

- Repeatable Read
  - A transaction may not perform dirty reads
  - An item read multiple times cannot change value (inconsistent reads)
  - Still does not guarantee global serializability
  - But a relation can change: "phantom" tuples

A: Insert Into Student [100 new tuples]

Set Transaction Isolation Level Repeatable Read

B: Select Avg(GPA) From Student

C: Select Max(GPA) From Student

- SQL Statement Order: B A C [Not Serializable]

Isolation Levels

- 关系R(A)包含元组{(1), (2)}，执行以下事务：
- T1：update R set A = 2 *A;
  insert into R values(6);
- T2：select avg(A) from R;
  select avg(A) from R;
- 如果T2以Repeatable Read执行，可能的结果？ A

A. 1.5, 4　　B. 1.5, 2, 4　C. 1.5, 3, 4　D. 1.5, 2, 3, 4

Isolation Levels

- Read Only transaction
  - Helps system optimize performance
  - Independent of isolation level

```
Set Transaction Read Only;
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Max(GPA) From Student;
```

Isolation Levels

	Dirty Reads (读尚未committed的数据)	Nonrepeatable reads (一个事务中对数据的两次读取数值不相同)	Phantoms (事务中能对关系增加新的元组)
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

- Read Uncommitted: 读数据前不对数据加S锁
- Read Committed: 读之前加S锁，读完释放
- Repeatable Read: 读之前加S锁，事务结束释放
- Serializable: 严格按照两段封锁协议对数据加锁

Isolation Levels

- Read Uncommitted
  - “Long duration” WRITE locks (Strict 2PL)
  - No READ locks (Read-only transactions are never delayed)
- Read Committed
  - “Long duration” WRITE locks (Strict 2PL)
  - “Short duration” READ locks (Only acquire lock while reading (not 2PL))
- Repeatable Read
  - “Long duration” WRITE locks (Strict 2PL)
  - “Long duration” READ locks (Strict 2PL)
- Serializable
  - “Long duration” WRITE locks (Strict 2PL)
  - “Long duration” READ locks (Strict 2PL)
  - Predicate locking (to deal with phantoms)

Isolation Levels Summary

- Standard default: Serializable
- Weaker isolation levels
  - Increased concurrency + decreased overhead = increased performance
  - Weaker consistency guarantees
  - Some systems have default Repeatable Read
- Isolation level per transaction and “eye of the beholder”
  - Each transaction’s reads must conform to its isolation level

Isolation Levels Summary

- In commercial DBMSs
  - Default level is often NOT serializable
  - Default level differs between DBMSs
  - Some engines support subset of levels
  - Serializable may not be exactly ACID
    - Locking ensures isolation, not atomicity
  - Also, some DBMSs do NOT use locking and different isolation levels can lead to different problems
  - Bottom line: Read the doc for your DBMS!

PostgreSQL Isolation Levels: http://www.postgresql.org/docs/current/static/mvcc.html

第十二章 事务处理

- 12.1 Transactions
- 12.2 Properties of Transactions
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.7 Logging

Recall (on disks)

- Sequential reads FASTER than random reads
- Sequential writes FASTER than random writes

Big Idea: LOGs (or log files or ledger)

- Any value that changes… Same with audit_trigger?
  - Write compact *change summary* (e.g. diff)
  - Into DB/disk blocks (i.e., sequential writes)
- Intuition
  - Data pages: (a) Optimistically, keep updating data pages in RAM (b) Delay updating data pages on disk
  - LOGs: (c) **But** keep track of updates in LOGs and (d) control when you Flush LOGs to disk

Basic Idea: (Physical) Logging

- Idea
  - Log consists of an ordered list of Update Records
  - Log record contains UNDO information for every update!

  <TransactionID, location, old data, new data>

- What DB does?
  - Owns the log “service” for all applications/transactions
  - Transparent to application or transaction
  - Sequential writes to log, can **Flush** — force writes to disk

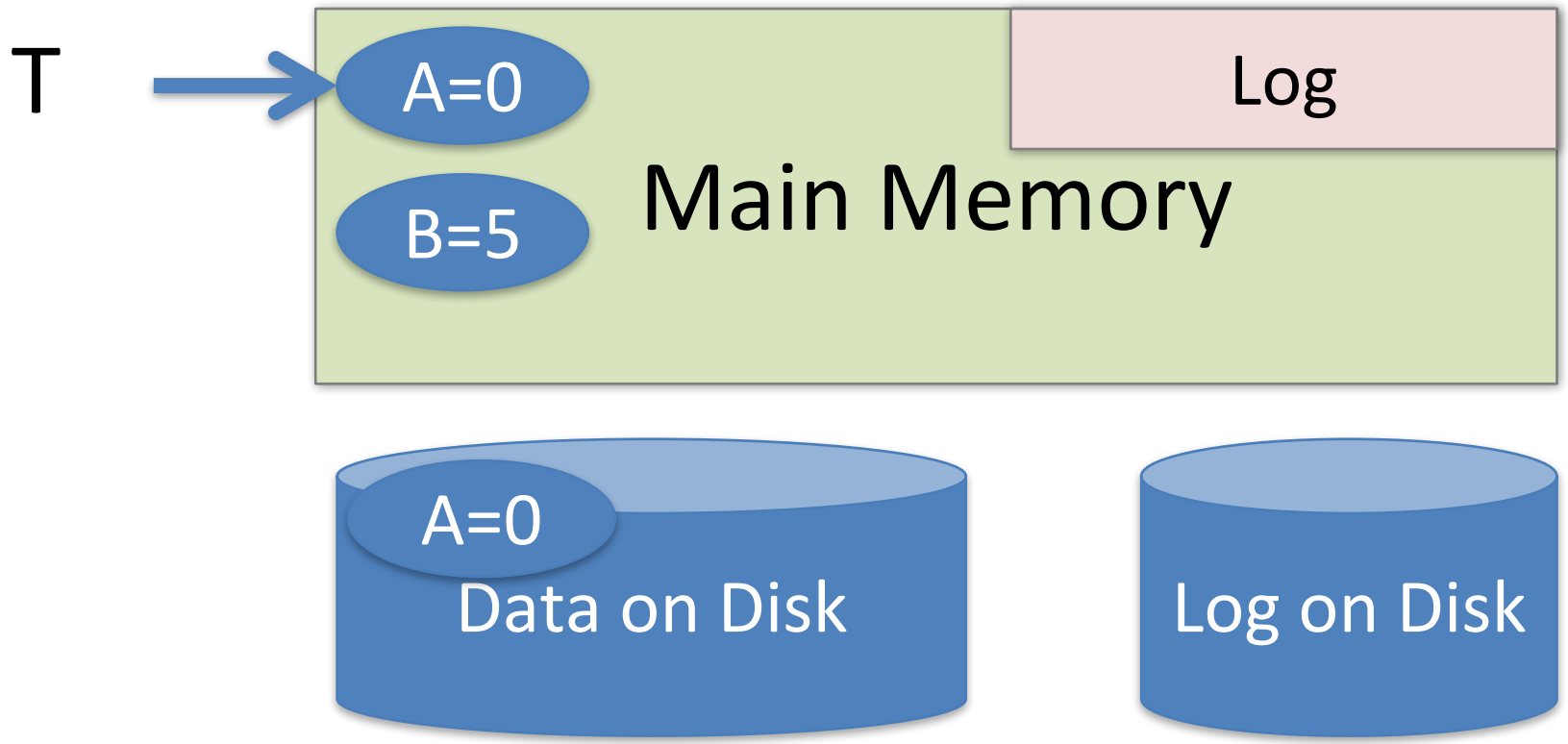
This is sufficient to UNDO any transaction!

A Picture of Logging

- T: R(A), W(A)

Log is a file (like any data table)
1. A set of Pages updated in RAM
2. Flushed as DB blocks on disk (sequential I/O)

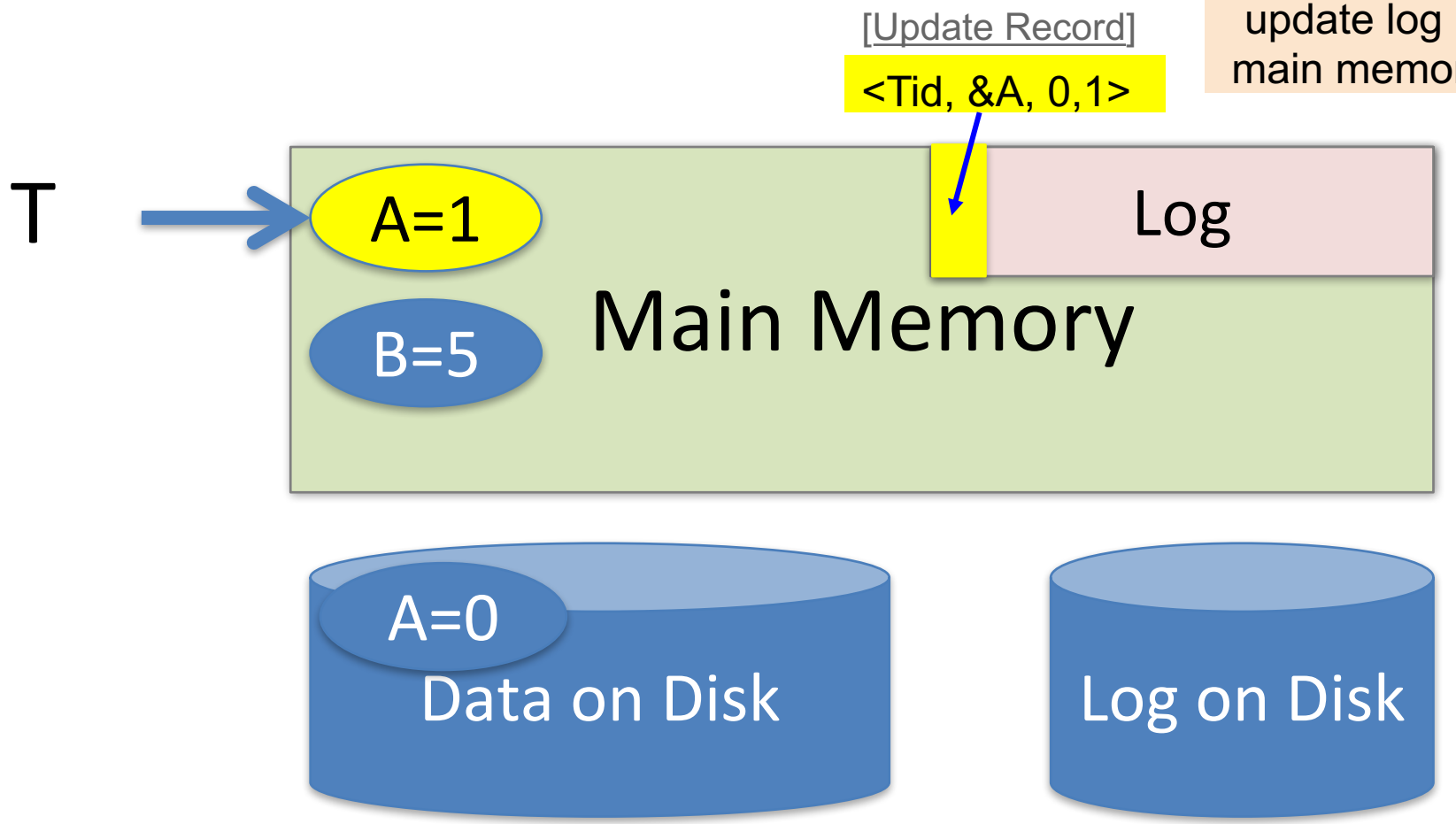
"Flushing to disk" = writing to disk from main memory



A Picture of Logging

- T: R(A=0), W(A=1)

Operation recorded in update log in main memory!



What do We Need Logging for Atomicity?

- Couldn't we just write TXN to disk only once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect - atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!

What is the Correct Way to Write This All to Disk?

- Write-Ahead Logging (WAL) protocol
- Remember: Key idea is to ensure durability while maintaining our ability to “undo”!

Write-ahead Logging (WAL) Commit Protocol

- T: R(A), W(A)

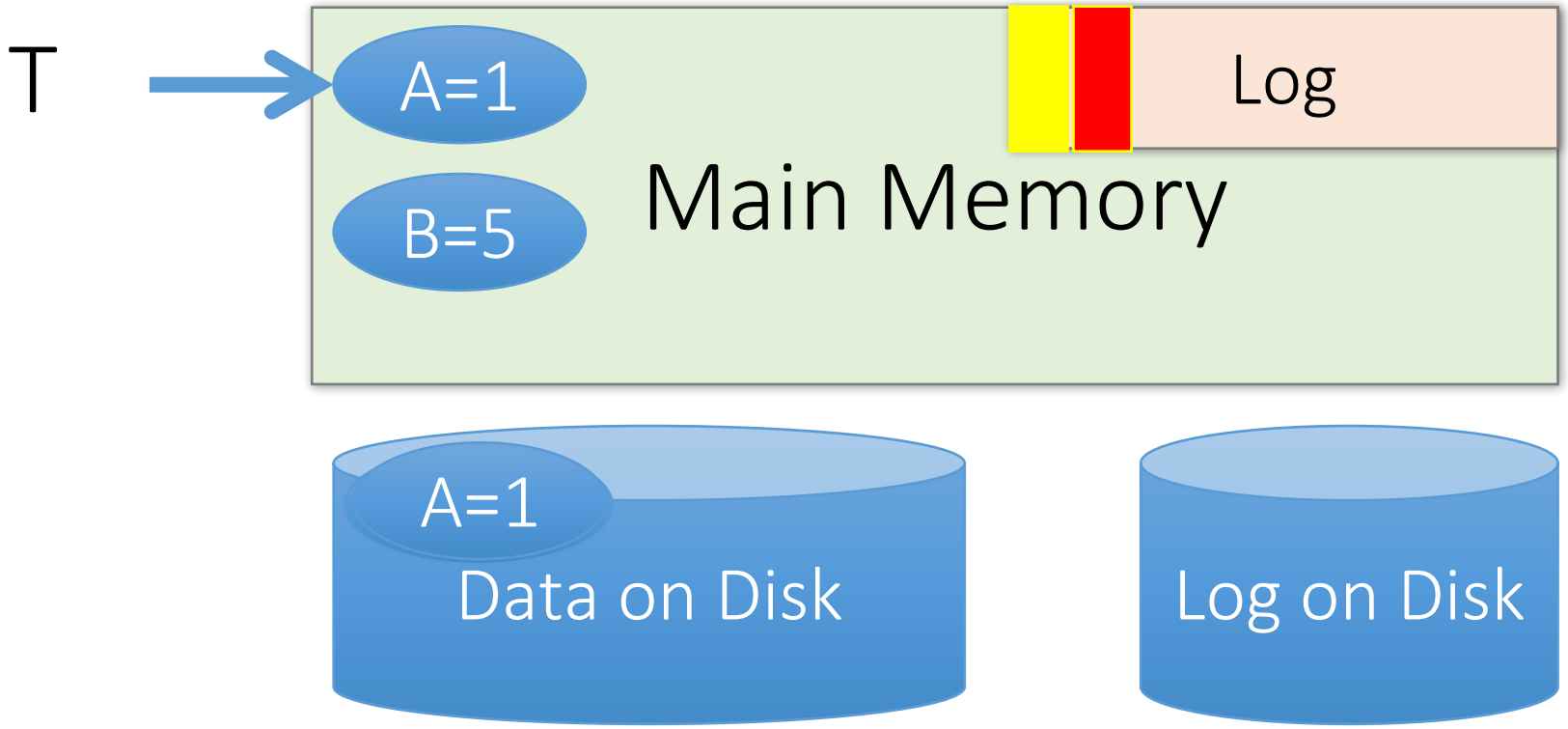
let's try committing after we've written log to disk but before we've written data to disk… this is WAL!

OK, Commit!

If we crash now, is T durable?

Use the log!

A: 0→1



Write-Ahead Logging (WAL)

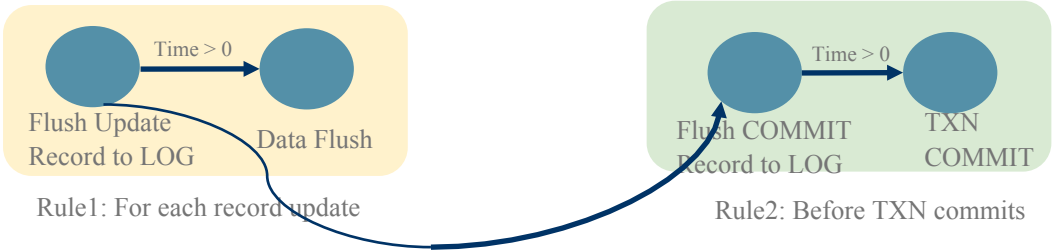
Algorithm: WAL

For each record update, write Update Record into LOG

Follow two Flush rules for LOG

- Rule1: Flush Update Record into LOG before corresponding data page goes to storage → **Durability**
- Rule2: Before TXN commits, → **Atomicity**
  - Flush all Update Records to LOG
  - Flush COMMIT Record to LOG

Transaction is committed *once COMMIT record is on stable storage*



Incorrect Commit Protocol #1

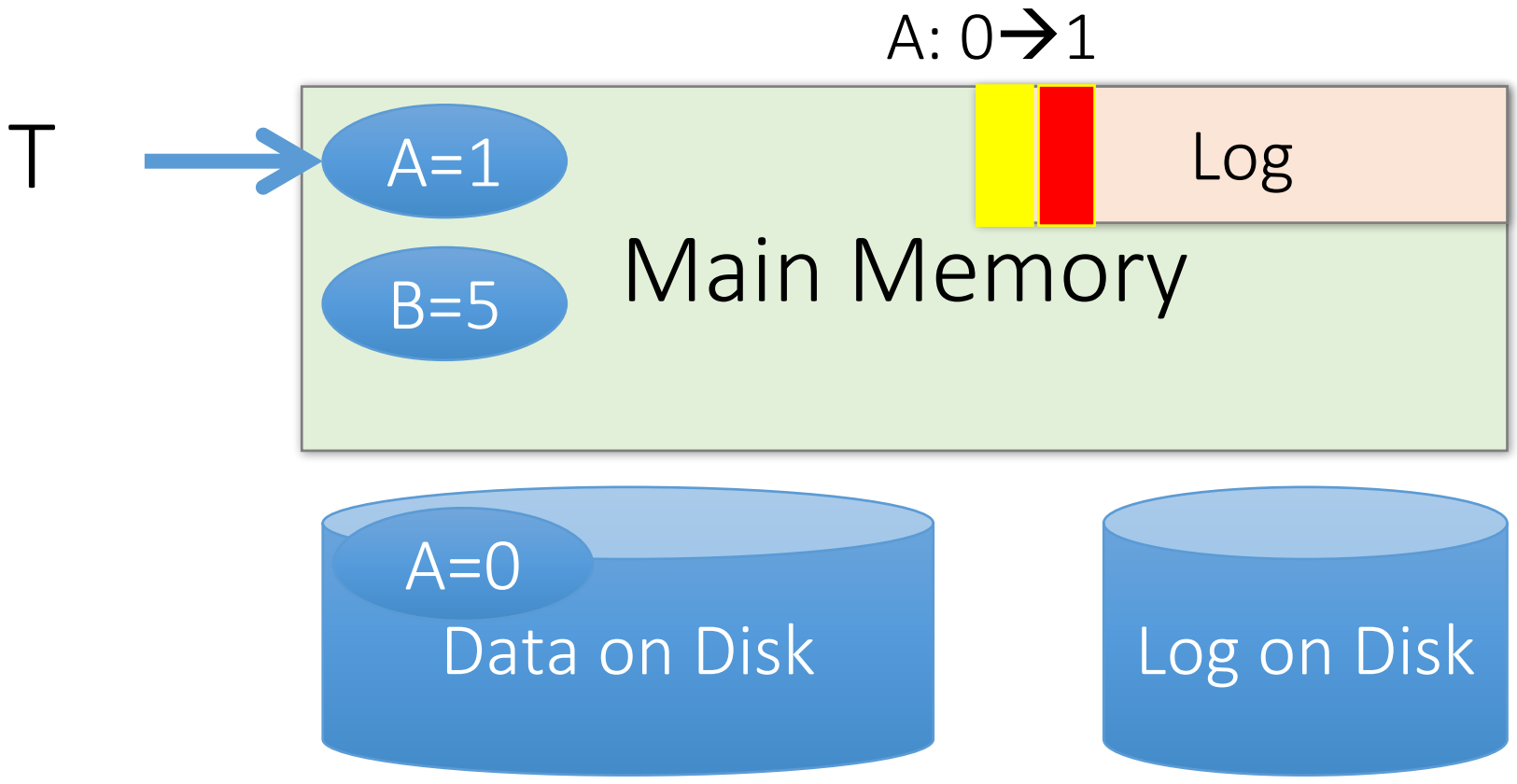
- T: R(A), W(A)

Let's try committing before we've written either data or log to disk…

OK, Commit!

If we crash now, is T durable?

Lost T's update!



Incorrect Commit Protocol #2

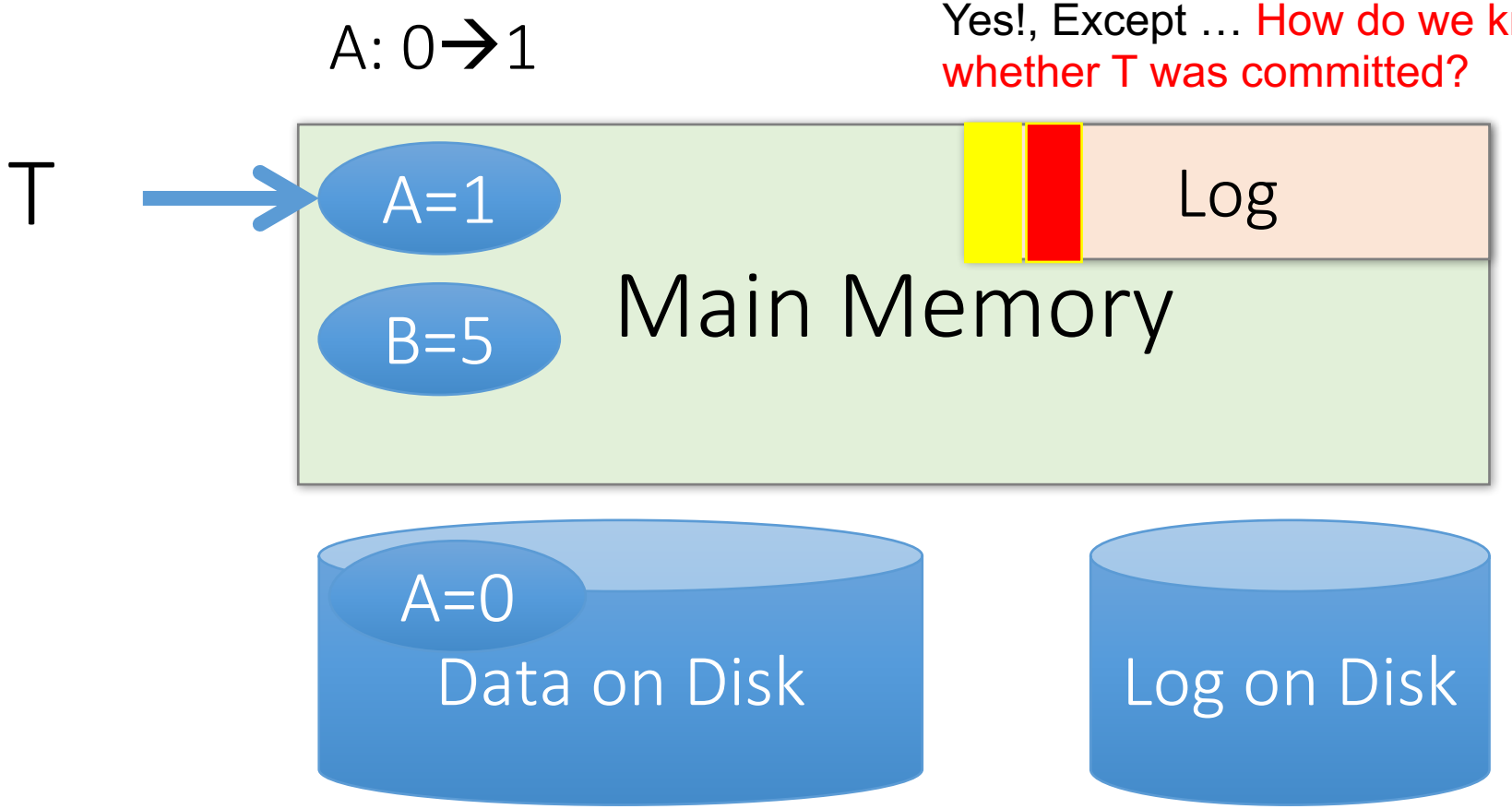
- T: R(A), W(A)

Let's try committing after we've written data but before we've written log to disk ...

OK, Commit!

If we crash now, is T durable?

Yes!, Except … How do we know whether T was committed?



Example

- Monthly bank interest transaction [Full Run]

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

Update Records

Commit Record

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

‘T-Monthly-423’

Monthly Interest 10%

4:28 am Starts run on 10M bank accounts

Takes 24 hours to run

```
START TRANSACTION
        UPDATE Money
        SET Amt = Amt * 1.10
COMMIT
```

Example

- Monthly bank interest transaction [With Crash]

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@10:45 am)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...		
30108		-110
40008		110
50002		22

??

??

??

??

WAL log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

'T-Monthly-423'
Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run
Network outage at 10:29 am,
System access at 10:45 am

Did T-Monthly-423 complete?
Which tuples are bad?

Case1: T-Monthly-423 was crashed
Case2: T-Monthly-423 completed.
4002 deposited 20$ at 10:45 am

Example

- Monthly bank interest transaction [Recovery]

Money (@10:45 am)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...		
30108		-110
40008		110
50002		22

Money (after recovery)

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

WAL log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22

System recovery (after 10:45 am)

1. Rollback uncommitted transactions
   - Restore old values from WALlog (if any)
   - Notify developers about aborted txn
1. Redo Recent transactions (w/ new values)
2. Back in business; Redo (any pending) transactions

Example

- Monthly bank interest transaction [Performance]

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

<u>Cost to update all data</u>

10M bank accounts → 10M seeks? (worst case)

(@10 msec/seek, that's 100,000 secs)

<u>Cost to Append to log</u>

+ 1 seek to get 'end of log'
+ write 10M log entries sequentially (fast!)
(< 1 sec)

[Lazily update data on disk later, when convenient.]

<u>Speedup for commit</u>

100,000 secs vs 1 sec!!!

Logging Summary

- If DB says TX commits, TX effect remains after database crash
- DB can undo actions and help us with atomicity

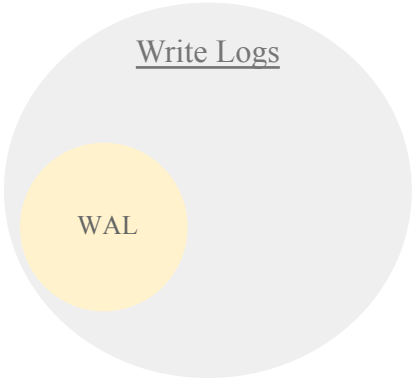
第十二章 事务处理

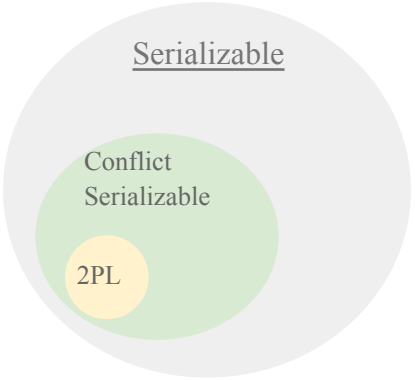
- 12.1 Transactions
- 12.2 Properties of Transactions
- 12.3 Concurrency
  - Interleaving & Scheduling
  - Conflict & Anomaly types
- 12.4 Conflict Serializability
- 12.5 Two-Phase Locking
- 12.6 Isolation Levels
- 12.7 Logging
- 12.8 Summary

参考教材：
数据库系统概念 4.3，14.1-14.8，15.1-15.4

12.8 Summary

- Why study Transactions?
  - Good programming model for parallel applications on shared data
    - Atomic
    - Consistent
    - Isolation
    - Durable





- Design choices?
  - Write update Logs (e.g., WAL logs)
  - Serial? Parallel, interleaved and serializable?