How Petflix Streams a Cat Video

A Top-to-Bottom Tour of a Distributed System

From global routing to CPU cores

The Setup

Meet Fatima

- Graduate student in Karachi
- Friday night, 10pm
- Taps play on "Mittens Goes to the Vet"
- 22-minute cat video

Petflix at Scale

200M
subscribers

15M
concurrent streams

The Question

What happens between Fatima's tap and the first frame on her screen?

We'll trace her request through 8 layers — from global DNS to CPU cores

The 8-Layer Stack

Fatima's request

#	Layer	Description
1	DNS Routing	Route users to nearest region
2	API Gateway	Authenticate & route requests
3	Microservices	Parallel fan-out to services
4	Data & Caching	Cache hits & persistent storage
5	CDN (Petflix Box)	Video served from user's ISP
6	Pods & Replication	Leader election & consensus
7	VMs & Containers	Isolation & orchestration
8	Cores & Threads	Thread scheduling & parallelism

Global

Local

Layer 1

Global DNS Routing

Where on the internet does play.petflix.com live?

DNS: Routing Fatima to Singapore

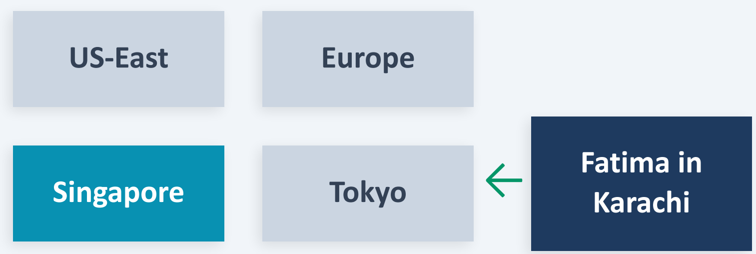
"What is the IP address of play.petflix.com?"

Fatima's Phone → ISP DNS Resolver → Root Nameserver → .com Nameserver → Petflix Auth DNS

Petflix DNS Intelligence

- Sees resolver IP (Karachi)
- Checks data center health & load
- Returns nearest healthy region IP
- **Karachi → Singapore (closest)**

Regional Routing



DNS Failover & Load Balancing

If Singapore Goes Down

- Health checks detect failure in seconds
- DNS stops returning Singapore's IP
- Next lookup routes to Hong Kong or Tokyo

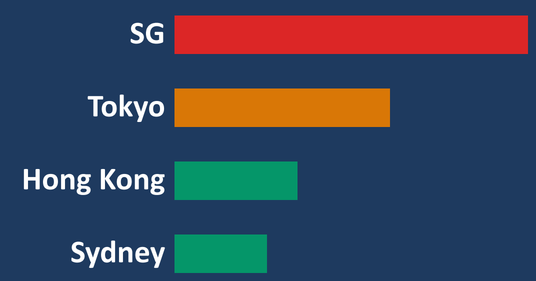
SG ✗ → Hong Kong

The TTL Problem

- DNS answers are cached by resolvers
- Cached IP may point to a dead region
- **Solution: keep TTLs short (30s)**
- Trade-off: more DNS traffic for faster failover

Geographic Bottleneck: Singapore

Singapore absorbs traffic from all of South and Southeast Asia. Smart DNS sometimes routes users to a slightly farther region to avoid overloading a hot data center.



Layer 2

API Gateway

PetGate: the front door to all of Petflix's backend

PetGate: Routing & Service Discovery

Fatima's Request → Load Balancer → PetGate API Gateway → Service Registry

Service Registry

Every instance registers on startup

Periodic heartbeats

1. Authenticate • 2. Route to Service • 3. Pick Healthy Instance

PetGate routes to healthy playback service instances

Playback #1	Playback #2	Playback #3	Playback #4
✓ Healthy	✓ Healthy	✗ Slow (5s)	✓ Healthy

Circuit Breaker: traffic to slow instance #3 is cut off

Circuit Breaker Pattern

CLOSED

Traffic flows normally.
PetGate monitors error rates.

→ *>50% errors*

OPEN

Traffic stopped.
Fallback response returned.
Cooldown period.

→ *cooldown expires*

HALF-OPEN

Single test request sent.
Success → close circuit.
Failure → stay open.

✓ success → CLOSED ✗ failure → stays OPEN

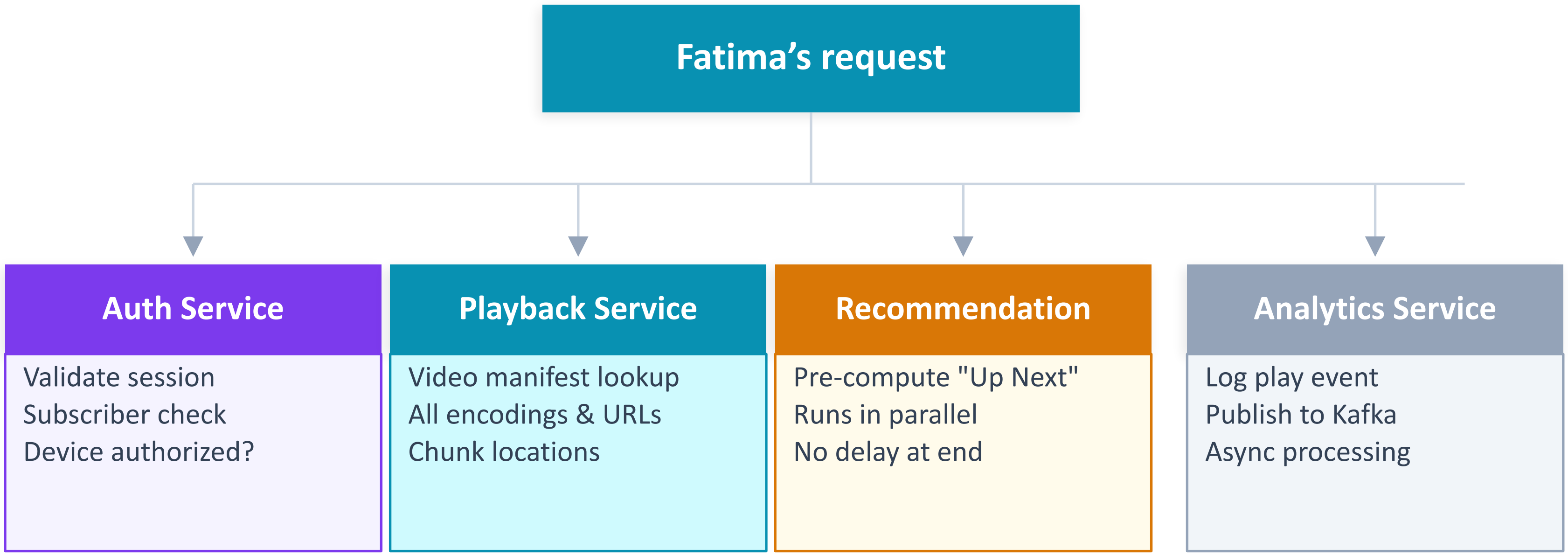
Bulkhead Principle: isolate failures so they don't spread. A slow playback instance shouldn't take down recommendations.

Layer 3

Microservices Fan-Out

One tap, four parallel service calls

Parallel Fan-Out



Key insight: total wait = slowest call, not the sum of all calls. All 4 fire simultaneously via gRPC.

Thundering Herd & Load Shedding

"Paws of Fury" Season 2 drops at 9pm

10 million users hit play within 30 seconds.
Each request fans out to 4 services = 40M internal calls.

Prioritized Load Shedding

PLAY REQUESTS	High Priority
RECOMMENDATIONS	Medium
ANALYTICS EVENTS	Low — can queue

Retry Storm Problem

Service overloaded → returns errors → all clients retry → doubled load on a drowning service

Mitigation

Exponential backoff
Each retry waits longer than the last

Jitter
Randomize wait times so retries don't sync

Layer 4

Data & Caching

400 million cache operations per second

Two-Tier Data Architecture

PetCache (Memcached)

200 clusters • 20,000 instances • trillions of items • sub-millisecond reads

✓ Cache Hit (fast path)

"Mittens" is popular — manifest returned in <1ms

✗ Cache Miss (fallthrough)

Obscure title → query falls to persistent store

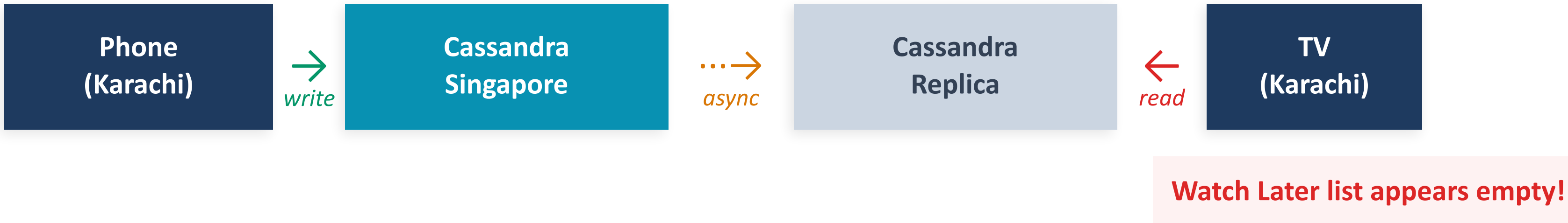
Apache Cassandra

Many clusters • petabytes of data • multi-region replication • eventual consistency

Consistent hashing: adding/removing a server only reshuffles 1/N of keys, not all of them

The Consistency Problem

Fatima's Watch Later list



The Consistency Spectrum

Eventual

Write returns before replication.
Other replicas catch up... eventually.

Analytics, view counts

Session-Level

Fatima always sees her own writes.
Others may see stale data briefly.

Watch Later list, profiles

Strong (ACID)

Write synchronously replicates everywhere before returning success.

Billing, payments

← Faster, less safe

Slower, more safe →

Layer 5

CDN — The Petflix Box

8,000+ boxes installed inside ISPs worldwide

Petflix Box: Video Served From Your ISP

Fatima's ISP (Karachi)

Petflix Box

High-capacity storage
Pre-loaded overnight
Popular content cached

→ 2 hops!

Fatima's Phone

origin fetch

Singapore Data Center

On Cache Miss

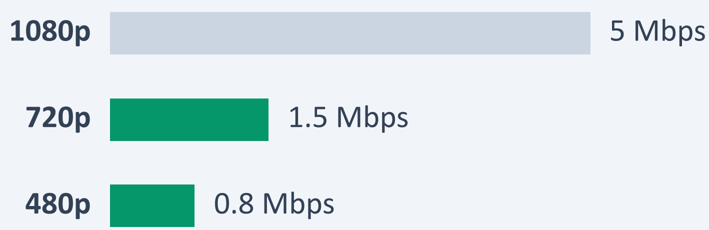
Niche title not pre-loaded?
Fetch from higher tier.
First chunks: +200ms latency.
Then cached for next viewers.

Win-win: Petflix ships boxes for free. ISP saves bandwidth. Users get low-latency HD video, even with unreliable international connectivity.

Per-Title Encoding & Adaptive Bitrate

"Mittens Goes to the Vet"

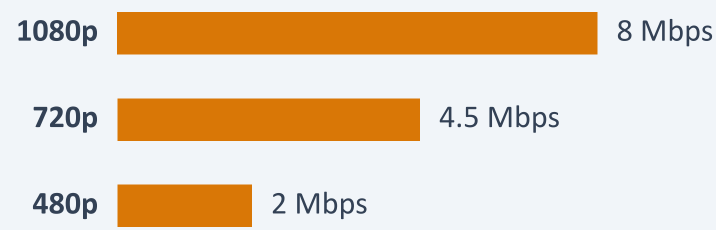
Low complexity: static shot, cat in carrier



720p looks identical to 1080p → encoded lower

"Dog Agility Championship"

High complexity: fast motion, fine detail



Needs high bitrates to look good (VMAF)

Adaptive Bitrate Streaming

Player monitors bandwidth and switches at chunk boundaries — seamlessly

Layer 6

Pods, Leader Election & Replication

How 5 machines agree on one truth

Leader-Follower Replication

Why Replicate?

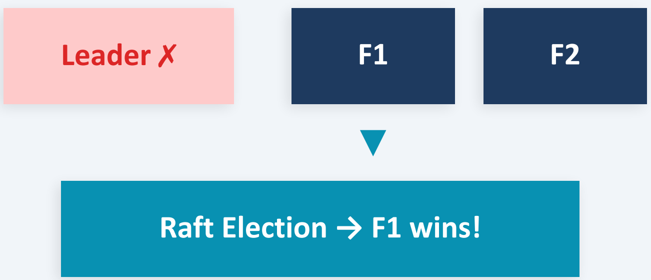
- Single machine = single point of failure
- 5 replicas: any one can die safely
- Trade-off: write latency for durability

The Consensus Problem

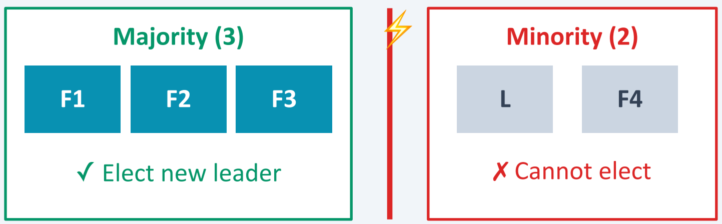
- At most one leader at any time
- Committed writes durable across majority
- Raft protocol (simpler than Paxos)

Leader Election & Network Partitions

Scenario: Leader Dies



Scenario: Network Partition



Raft Consensus Rules

Strict majority required

A candidate needs ≥3 of 5 votes. Only one majority exists in any group of 5.

Randomized timers

Followers start elections at random intervals to avoid split votes.

Old leader steps down

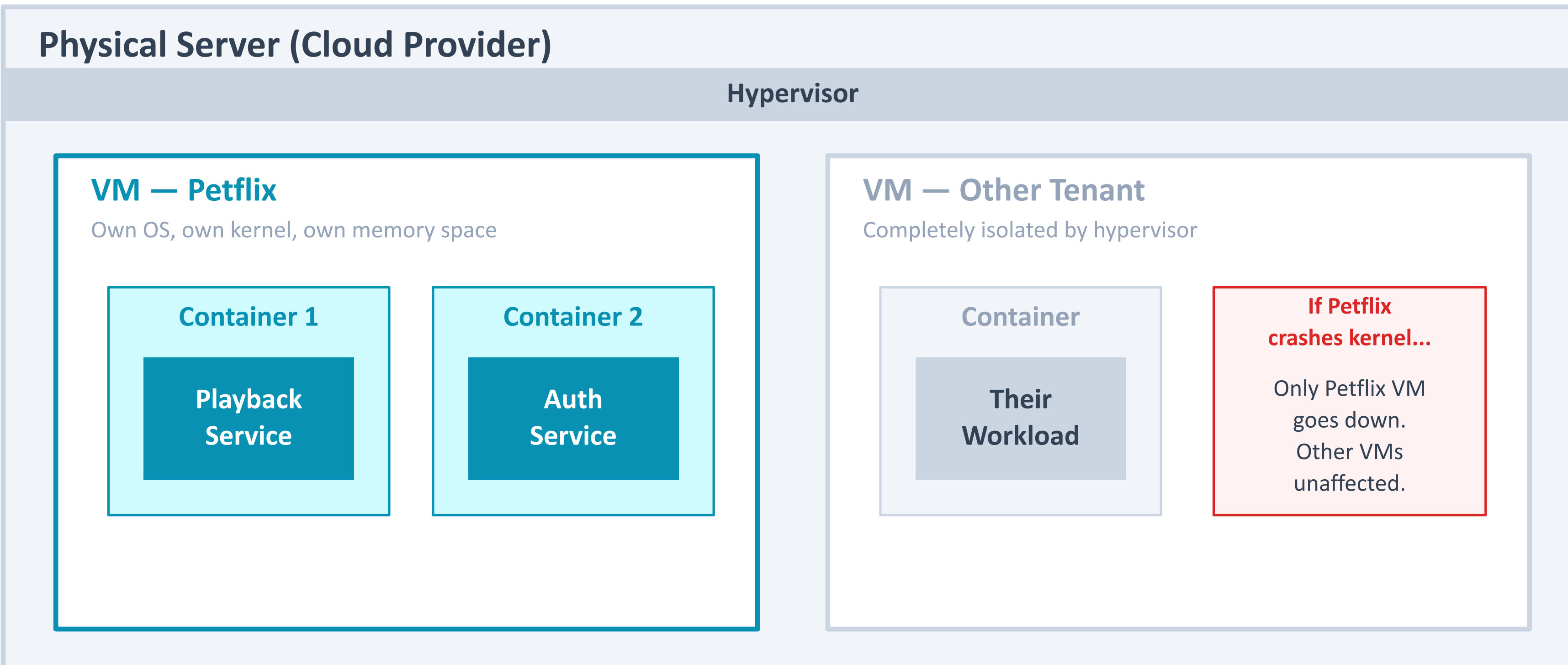
When isolated leader can't reach majority, it rejoins as follower after healing.

Layer 7

VMs, Containers & the Orchestrator

Russian nesting dolls of isolation

The Isolation Stack



VMs = strong isolation (own kernel) | Containers = lightweight isolation (shared kernel) | Both = defense in depth

Kubernetes: The Orchestrator

Schedule

Decide which containers run on which VMs

Scale

Spin up/down replicas based on load

Heal

Restart crashed containers automatically

Deploy

Rolling updates, one at a time, zero downtime

"Paws of Fury" Traffic Spike

Normal Load

Pod 1 | Pod 2 | Pod 3

Spike Detected → Auto-Scale

Pod 1 | Pod 2 | Pod 3 | New 1 | New 2 | New 3 | New 4

Noisy Neighbor Problem

Shared physical hardware means another tenant's I/O burst can slow you down.

Mitigation: resource reservation, or dedicated hardware for critical services.

Horizontal vs. Vertical scaling

Layer 8

Cores, Threads & Video Decoding

Where concurrency meets real hardware

Concurrency vs. Parallelism

Petflix Box: 16-Core CPU

Core 1	Core 2	Core 3	Core 4
Core 5	Core 6	Core 7	Core 8
Core 9	Core 10	Core 11	Core 12
Core 13	Core 14	Core 15	Core 16

Concurrency

1,000 threads managed simultaneously. The system juggles many tasks, switching between them rapidly (context switching).

Parallelism

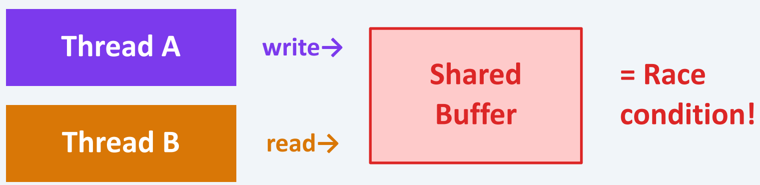
Only 16 things happen at the exact same instant — one per core. Multi-core CPUs make true parallelism real.

Thread Queue: 1,000 streams

16 running • 984 waiting

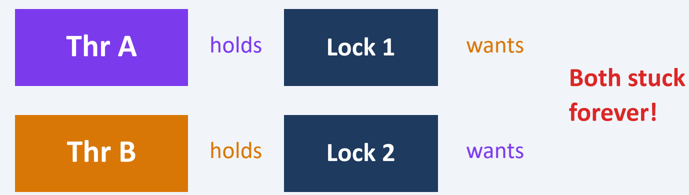
Race Conditions, Locks & Real-Time Decoding

Shared Memory: The Danger

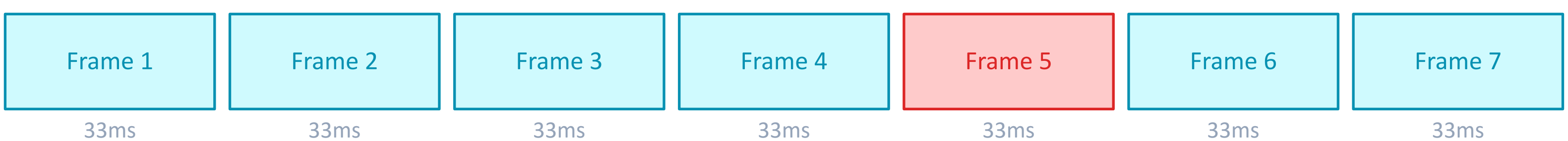


Solution: Locks (mutual exclusion)

Deadlock



Real-Time Decoding on Fatima's Phone



30 fps = 33ms per frame. Miss the deadline = visible stutter.

Hardware decoders handle the heavy lifting.

The Full Stack in One Picture

Layer	Concurrency Problem
DNS Routing	Global load balancing
API Gateway	10K concurrent connections
Microservices	Billions of internal calls/min
Cache & DB	400M cache ops/sec
CDN Box	1000s streams/box
Pods	Write serialization
VMs	Bin-packing containers
Cores	Race conditions, deadlines

Concurrency is not one problem.

It's a different problem at every layer.

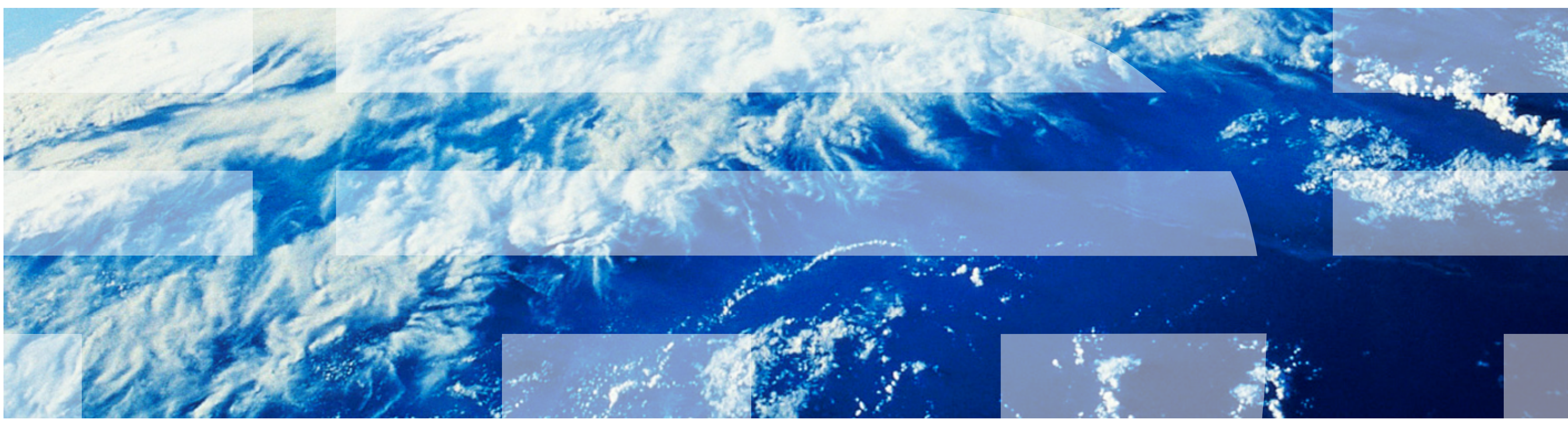
DNS | Gateway | Services | Cache | CDN | Pods | VMs | Cores

Our users don’t, and should not have to know any of this.

Borrowed from Shiva Shivakumar

Computer Systems for Data Science
Topic 3

Transactions and OLTP Databases



A note on OLTP vs. OLAP

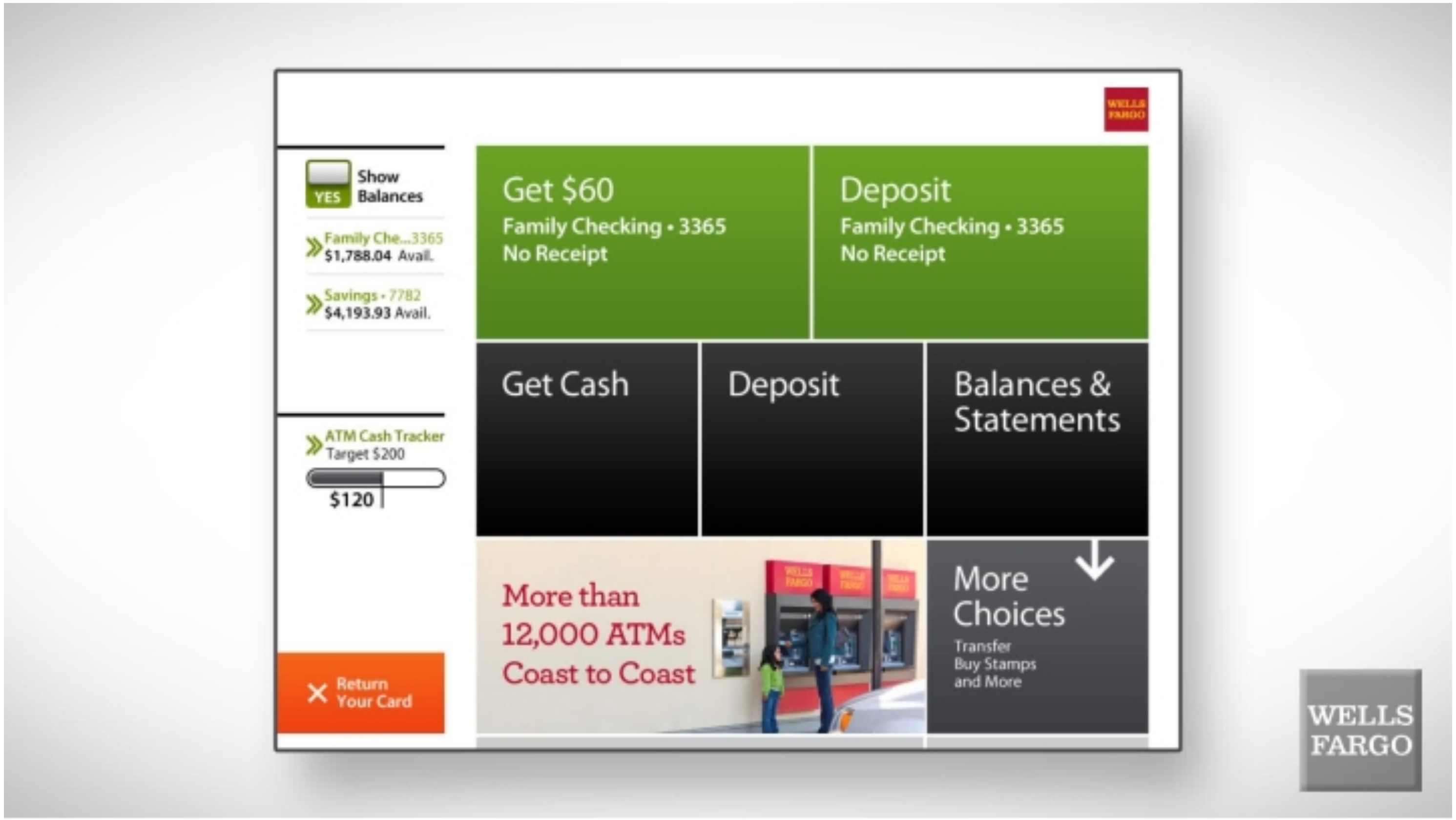
- There are broadly two types of databases in the world: OLTP and OLAP
- OLTP
  - Online Transaction Processing database
  - Supports reading and writing/updating data in real time
  - Provides **transactions** abstraction (usually, but not always!)
  - Usually smaller queries
  - Sometimes relational, sometimes not
  - Use cases
    - Financial transactions
    - Keep up-to-date progress in a game
    - Up-to-date user settings in a social media site
    - Google docs
  - Examples: MySQL, Postgres, Redis, Cassandra, DynamoDB, Aurora/ RDS, …

A note on OLTP vs. OLAP

- OLAP
  - Online Analytical Processing Database
  - Read-only, no updates, data is re-written in batches offline (e.g., once a day)
  - Usually larger queries (scans, aggregates)
  - Sometimes relational, sometimes not
  - Use cases
    - Interactive data analysis
    - Monthly business report
    - Design a statistical model
  - Examples: BigQuery, Snowflake, Databricks, Redshift, …

- We will focus in the next couple of weeks on **OLTP**

OLTP motivating example: an ATM



Read Balance
Give money
Update Balance

vs

Read Balance
Update Balance
Give money

What if multiple applications/users are accessing the same table?

Visa does > 60,000 TXNs/sec with users & merchants

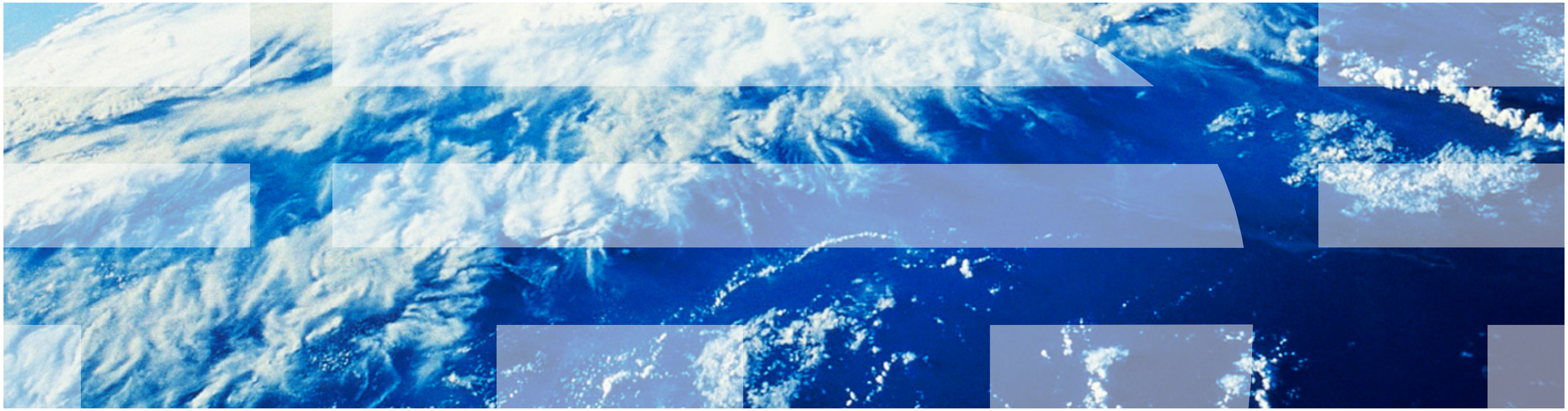
Want your 6$ Starbucks transaction to wait for a 10k$ bet in Las Vegas ?
(Transactions can (1) be quick or take a long time, (2) unrelated to you)

Transactions are not just used for finance

Transactions are at the core of

-- payment, stock market, banks, ticketing
-- Gmail, Google Docs (e.g., multiple people editing)
-- Gaming
-- Healthcare systems
…

Transactions



Example: monthly bank interest transaction

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’

Monthly Interest 10%

4:28 am Starts run on 10M bank accounts

Takes 24 hours to run

```
UPDATE Money
SET Balance = Balance * 1.1
```

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	??

‘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

Transactions: Basic Definition

A transaction ("TXN") is a sequence of one or more *operations* (reads or writes) which reflects *a single real-world transition*.

TXN either happened completely or not at all

```
START TRANSACTION
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = ‘Gizmo’
COMMIT
```

Transactions in SQL

- In “ad-hoc” SQL, each statement = one transaction

- In a program, multiple statements can be grouped together as a transaction

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 100
        WHERE name = ‘Joe’
COMMIT
```

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **Recovery and Durability**: Keeping the DB data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

2. **Concurrency**: Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation -- Recovery & Durability

1. **Recovery and durability** of user data is essential for reliable database (and other data science systems)

- The database 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

Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99

DELETE Product
WHERE price <=0.99
```

Crash / abort!

What goes wrong?

Protection against crashes / aborts

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
```

Now we'd be fine! We'll see how / why this lecture

Motivation -- Concurrent execution

2. **Concurrent** execution of user programs is essential for good database 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 that consistency is maintained

Idea: Have the database handle running several user TXNs concurrently, in order to keep throughput high

Multiple users: single statements

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

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

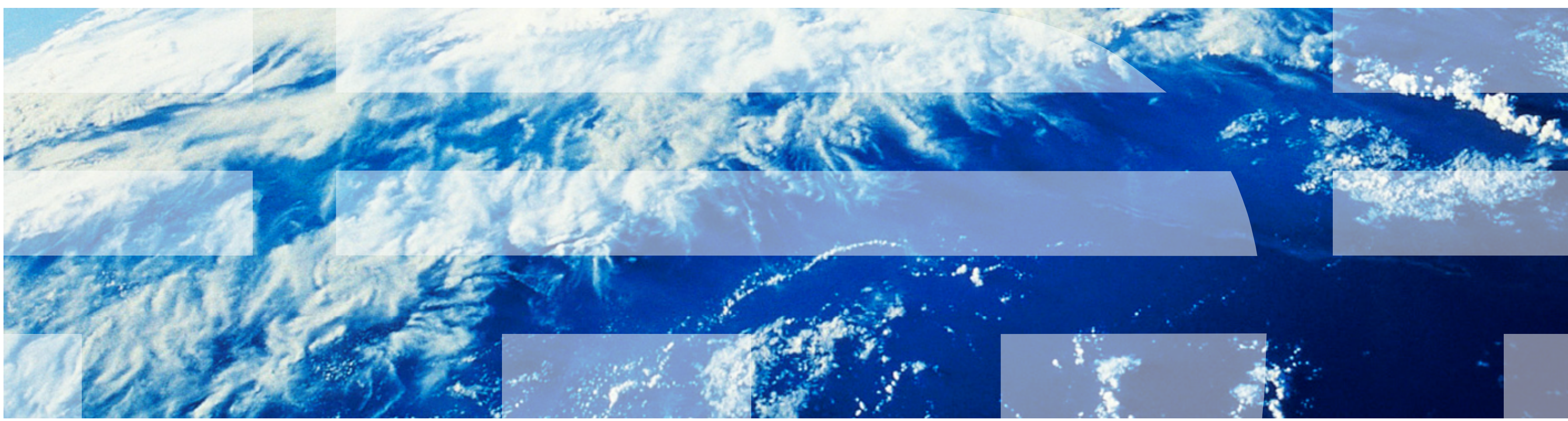
Two managers attempt to discount products *concurrently*-
What could go wrong?

Multiple users: single statements

```
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
```

ACID
Atomicity, Consistency, Isolation, Durability



Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!
(in fact, by default, most databases don't provide it…)

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 (primary key constraint)
    - 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 another consistent state
  - *System* makes sure that the TXN is atomic
  - → Assuming system maintaining atomicity, this is often the user's responsibility

FINANCE

Europe

Irish bank glitch let customers pull out large sums of ‘free money’—sparking huge run on ATMs

By **Chloe Taylor**

August 16, 2023, 10:08 AM ET

A woman at a Bank of Ireland ATM in Clifden, April 2021. Thanks to a glitch, some Bank of Ireland customers recently found themselves able to withdraw hundreds of euros they didn’t own from their accounts, prompting long lines at ATMs.

ARTUR WIDAK—NURPHOTO/GETTY IMAGES

https://fortune.com/europe/2023/08/16/bank-of-ireland-free-money-glitch-lines-at-atms-police-garda-intervention-dublin/

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. Transaction A not be able to observe changes from another concurrent transaction B 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
  - And in data center settings: replicate data, backup, etc.

Challenges for ACID properties

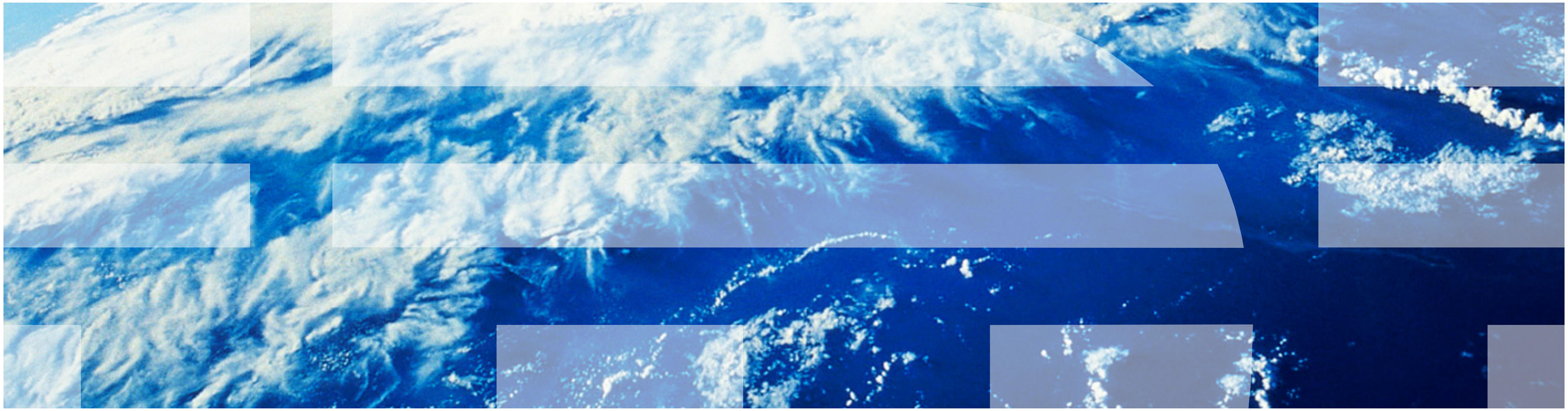
- In spite of power failures (i.e., in spite of loss of memory)
- Users may abort the program: need to "rollback changes"
  - Need to *log* what happened
- Many users executing concurrently

And all this with… Scalability and/or Performance!!

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**
- Many SQL databases do not provide ACID by default
  - Often provide read-committed transactions, a weaker form of isolation
- “NoSQL” DBs relax ACID even more
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBs…

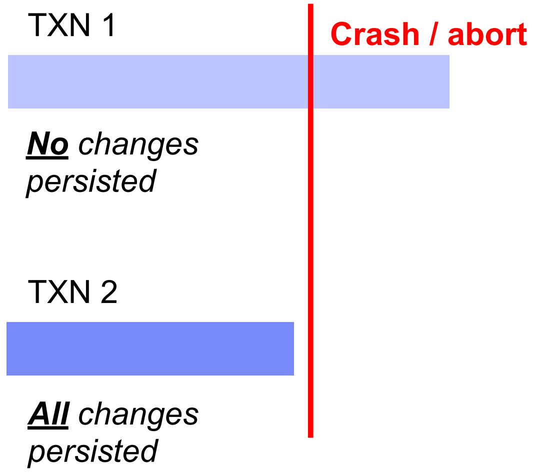
Atomicity and Durability via Logging



Goal: Ensuring Atomicity & Durability

- Atomicity:
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

- Durability:
  - If DB stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*



We'll focus on how to accomplish atomicity (via logging)

Basic Idea: (Physical) Logging

The tables themselves has no notion of history, so we introduce a notion of history using a **log**

Idea:

- Log consists of an ordered list of update records for ongoing transactions
- 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
  - Logically (and usually physically) separate from the actual data
- Transparent to application or transaction
- Sequential writes to log, can **flush** — force writes to disk

This is sufficient to UNDO any transaction!

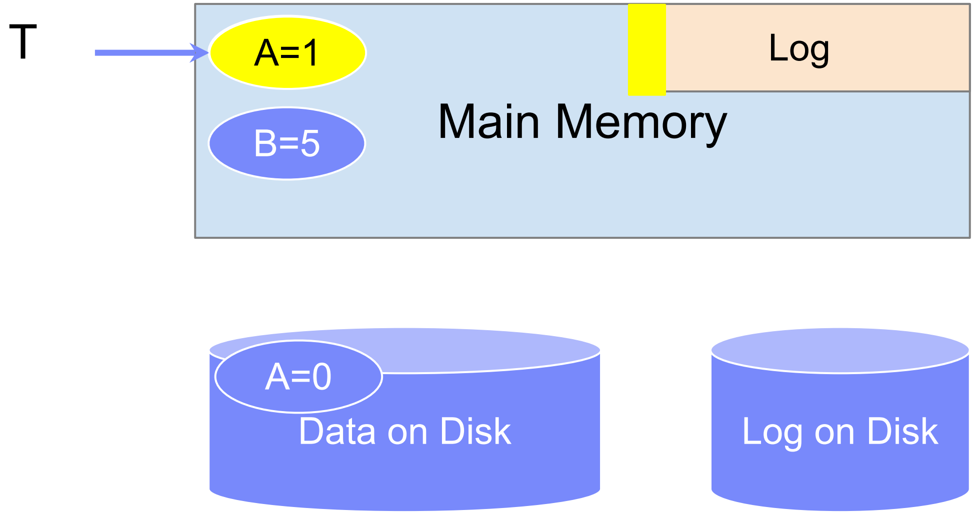
Using logging for atomicity

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

[T reads A=0, writes A=1]

[Update Record]

<Tid, &A, 0,1>



Operation recorded in update log in main memory!

Why do we need logging for atomicity?

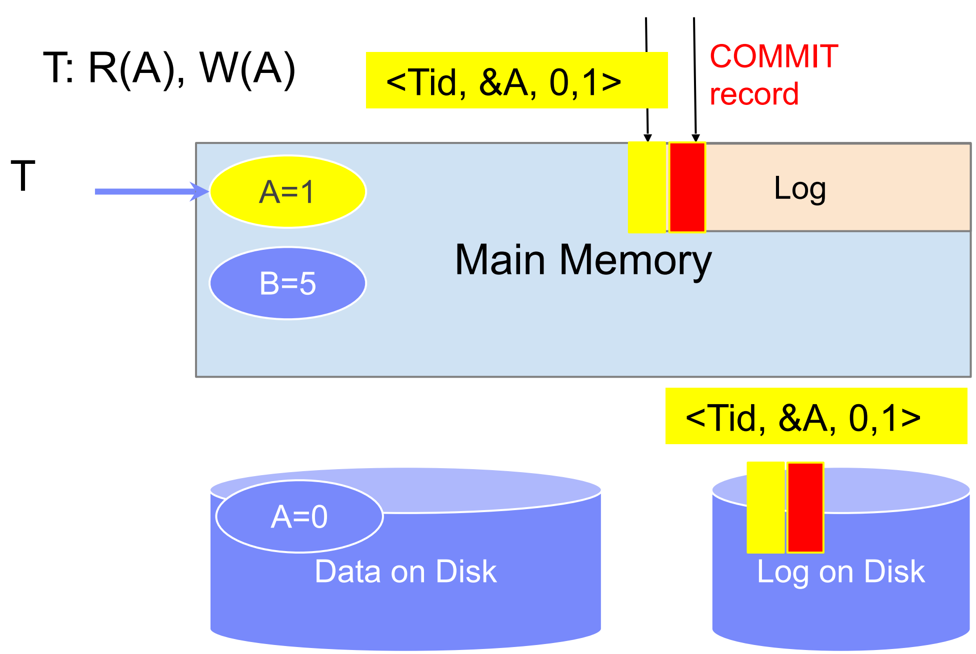
- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and without performance constraints, this could work…*

- However, we **need write partial results of TXNs to disk** 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!

Write-ahead Logging (WAL) Commit Protocol

[Update Record]



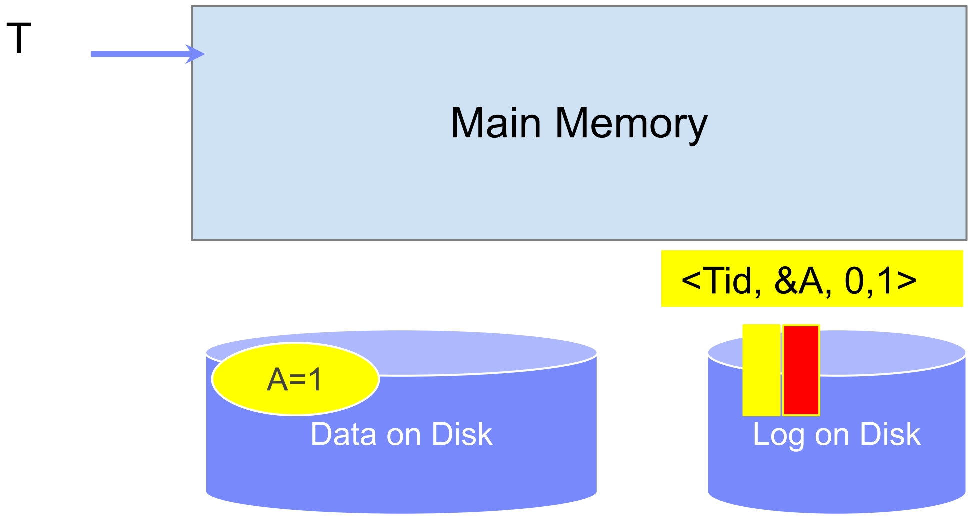
Commit after we've written log to disk but before we've written data to disk…

OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



Commit 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!

Write-Ahead Logging (WAL)

Algorithm: WAL

For each record update, write Update Record into LOG

Follow two Flush rules for LOG

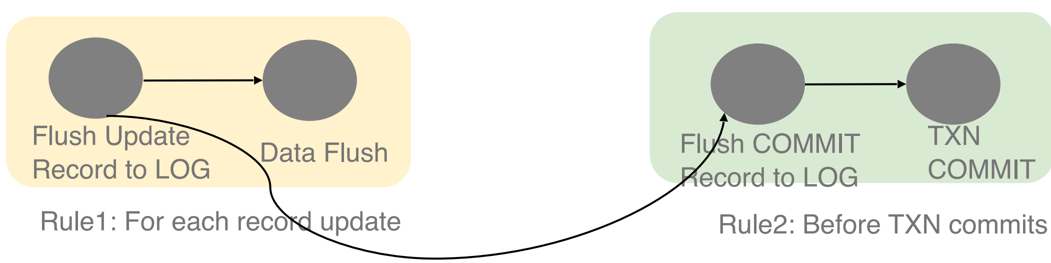
- **Rule1**: Flush Update Record *into LOG before* corresponding data page goes to storage
- **Rule2**: Before TXN commits,
  - Flush all Update Records to LOG
  - Flush COMMIT Record to LOG

Flush means write to disk

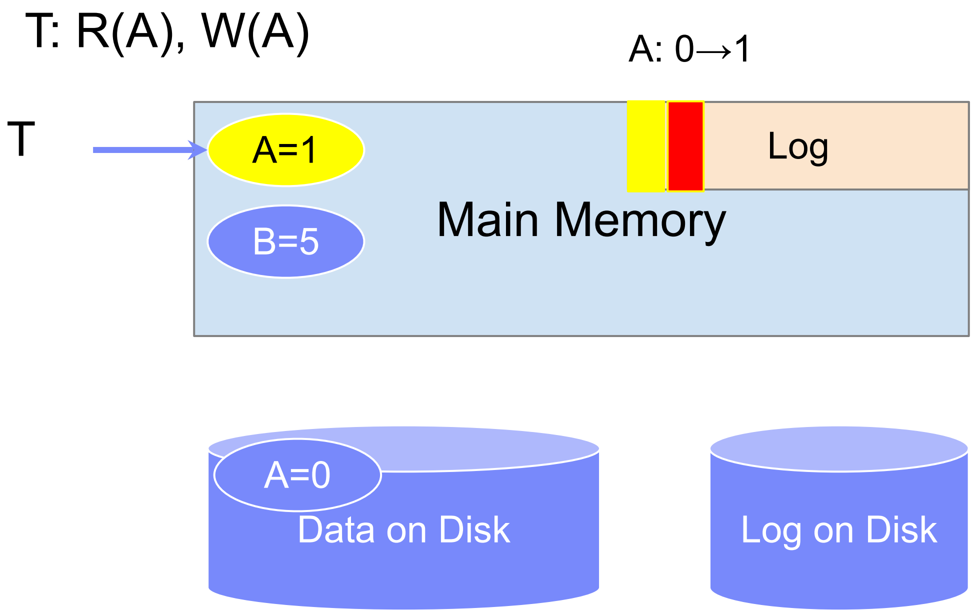
→ **Durability**

→ **Atomicity**

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



Incorrect Commit Protocol #1



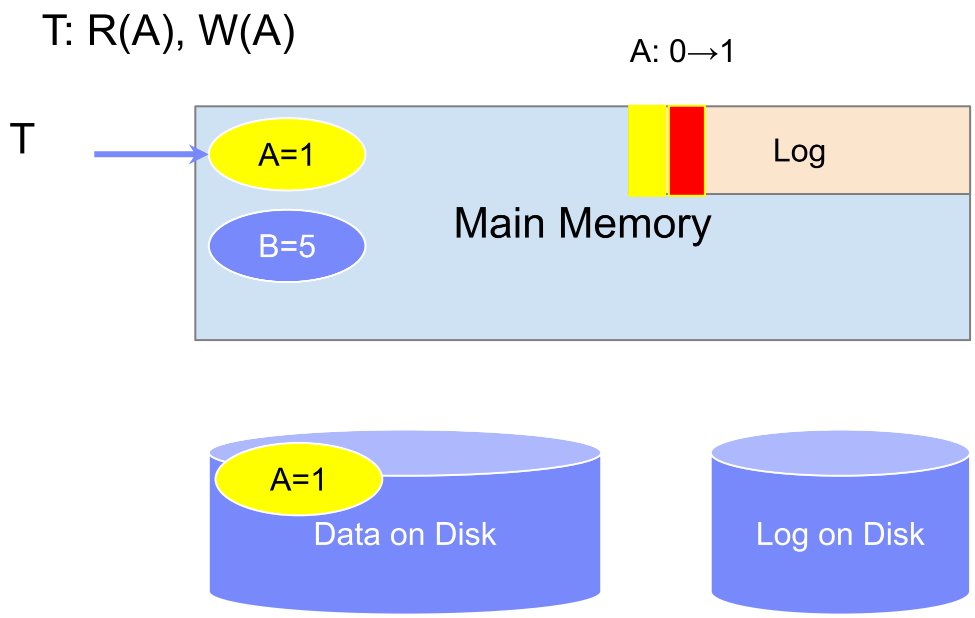
Let's try committing *before* we've written either data or log to disk…

OK, Commit!

If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2



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??

Bank interest example: 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 Balance = Balance * 1.10
COMMIT
```

Bank interest example: 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

Bank interest example: with 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

[redacted]

<u>System recovery (after 10:45 am)</u>

1. Undo 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

A word on performance

- Question: why is a WAL good for performance?
  - Answer 1: updates to WAL are in sequential order
  - Answer 2: flushing the actual entries (i.e., the data in the tables) can be done lazily after the transaction was committed
    - Lets us have sequential writes also for the data, not just for the WAL!
- Sequential writes are very important both for flash and magnetic disk
  - In a couple of lectures we will understand why

An example of why sequential writes matter

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)

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 individual random writes? (worst case)

(@10 ms per write for magnetic disk, that's 100,000 secs)

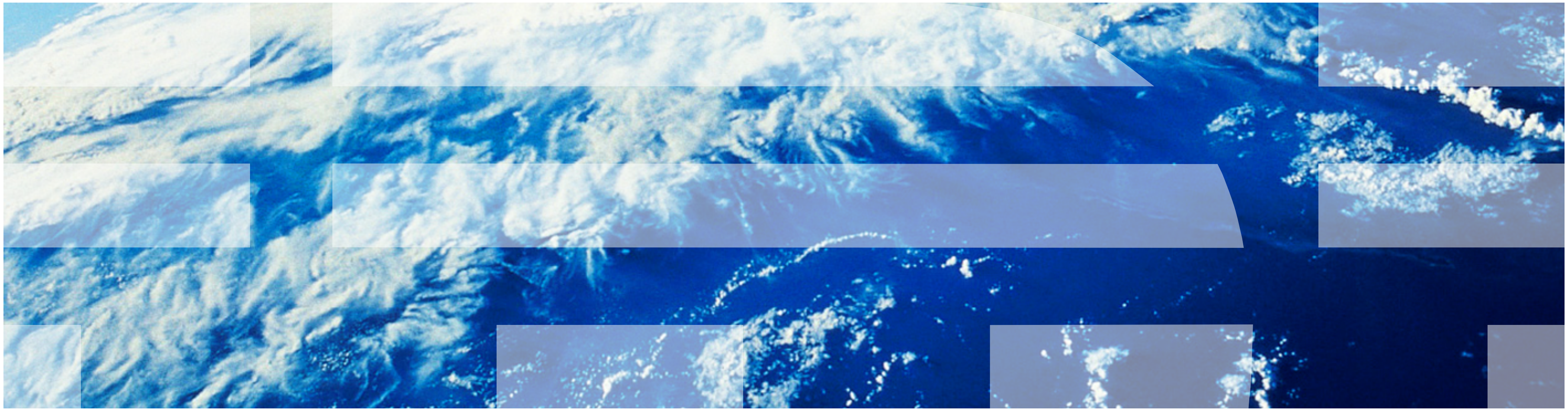
<u>Speedup for commit</u>

100,000 secs vs 1 sec when written sequentially!!!

Summary so far

- If we follow the WAL rules, we get 2/4 of the ACID properties:
  - **A**tomicity
  - **D**urability
- We'll ignore consistency, since databases usually don't implement many constraints
- But what about **I**solation?
  - What happens if concurrent transactions interfere with each other?

Motivation: Concurrent Transactions



Back to our bank example

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

<u>‘T-Monthly-423’</u>

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run

```
UPDATE Money
SET Balance = Balance * 1.1
```

<u>Other Transactions</u>

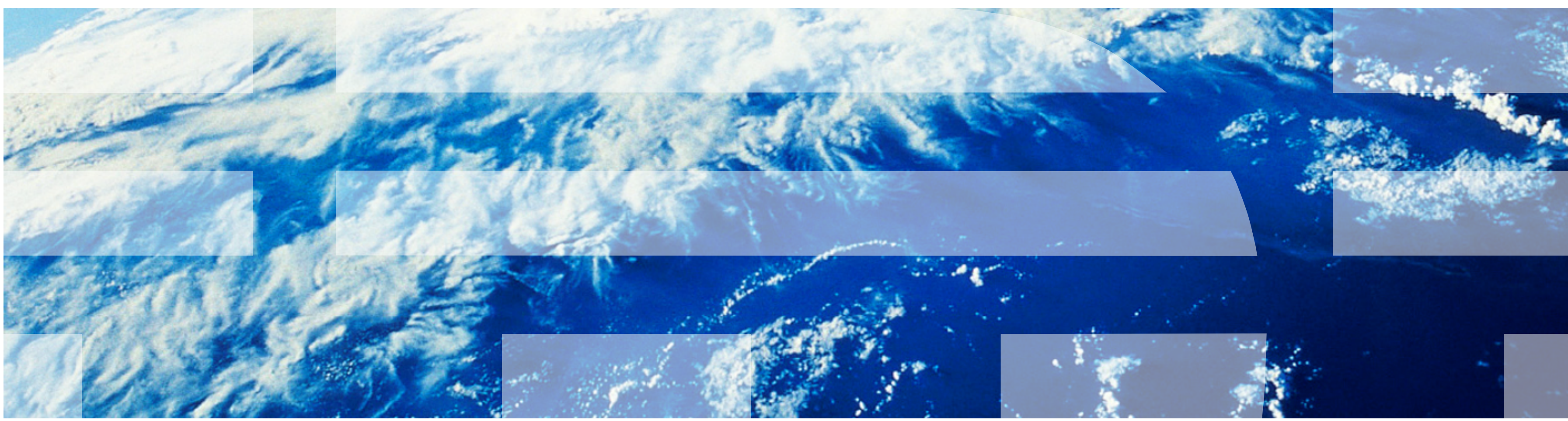
10:02 am Acct 3001: Wants 600$
11:45 am Acct 5002: Wire for 1000$
. . . . .
. . . . .
2:02 pm Acct 3001: Debit card for $12.37

Q: How do I not wait for a day to access my $$$s?

Big idea: locks

- Intuition
  - ”Lock” each record for shortest time possible
- Key questions
  - Which records?
  - For how long?
  - What is the algorithm for holding them?

Concurrency, Scheduling and Anomalies



Concurrency: Isolation & Consistency

- DB is responsible for concurrency so that…

Isolation is maintained: Users must be able to execute each TXN as if they were the only user

AC**I**D

Consistency is maintained: TXNs must leave the DB in consistent state

A**C**ID

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
```

```
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

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)

Example

T_1 | A += 100 | B -= 100 | T1 transfers $100 from B's account to A's account

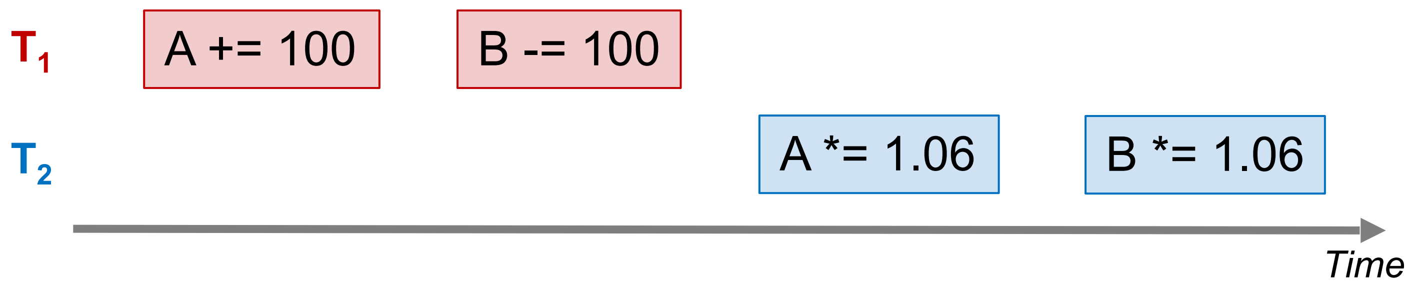
T_2 | A *= 1.06 | B *= 1.06 | T2 credits both accounts with a 6% interest payment

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:

We can look at the TXNs in a timeline view- serial execution:

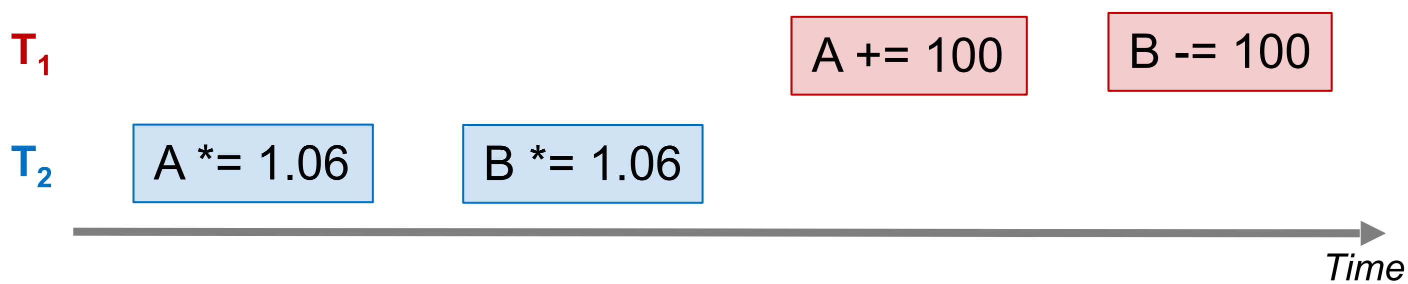


T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

The TXNs could occur in either order… DB allows!

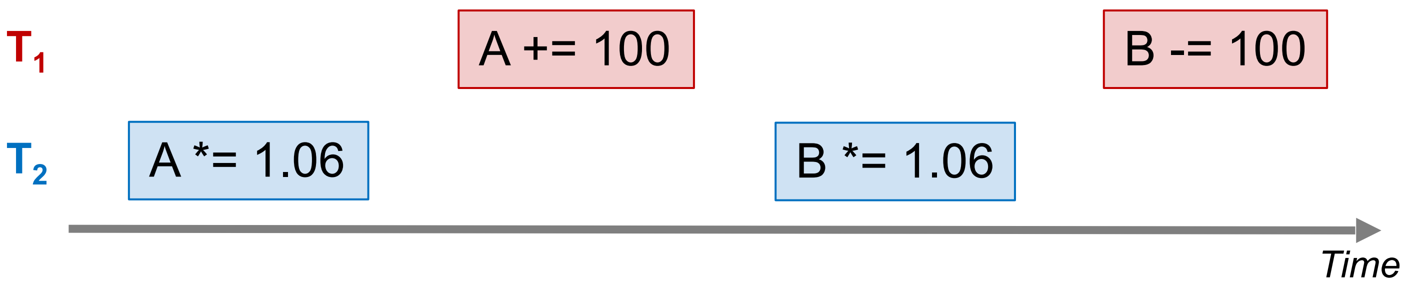


T2 credits both accounts with a 6% interest payment

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

Example- consider two TXNs:

The DB can also **interleave** the TXNs



T2 credits A's account with 6% interest payment, then T1 transfers $100 to A's account…

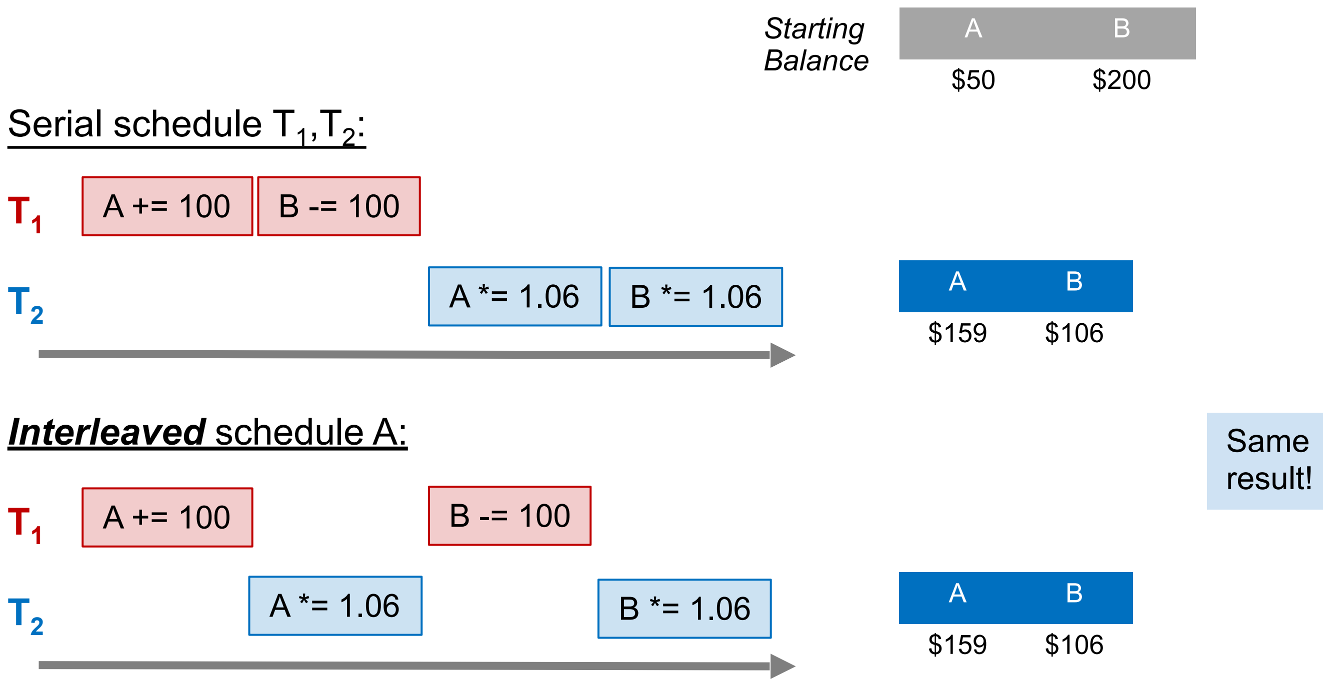
T2 credits B's account with a 6% interest payment, then T1 transfers $100 from B's account…

Interleaving & Isolation

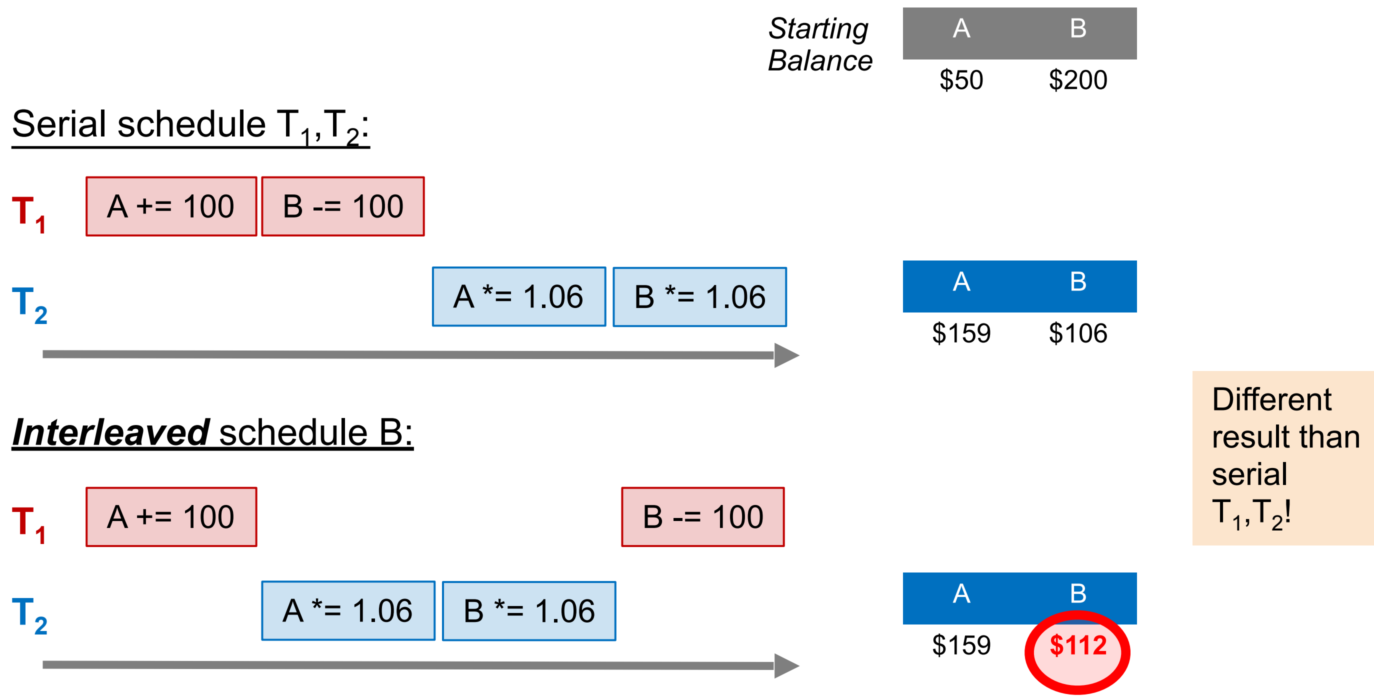
- The DB has freedom to interleave TXNs
- However, it must pick an interleaving or schedule such that isolation and consistency are maintained
- ⇒ Must be *as if* the TXNs had executed serially!

DB must pick a schedule which maintains isolation & consistency

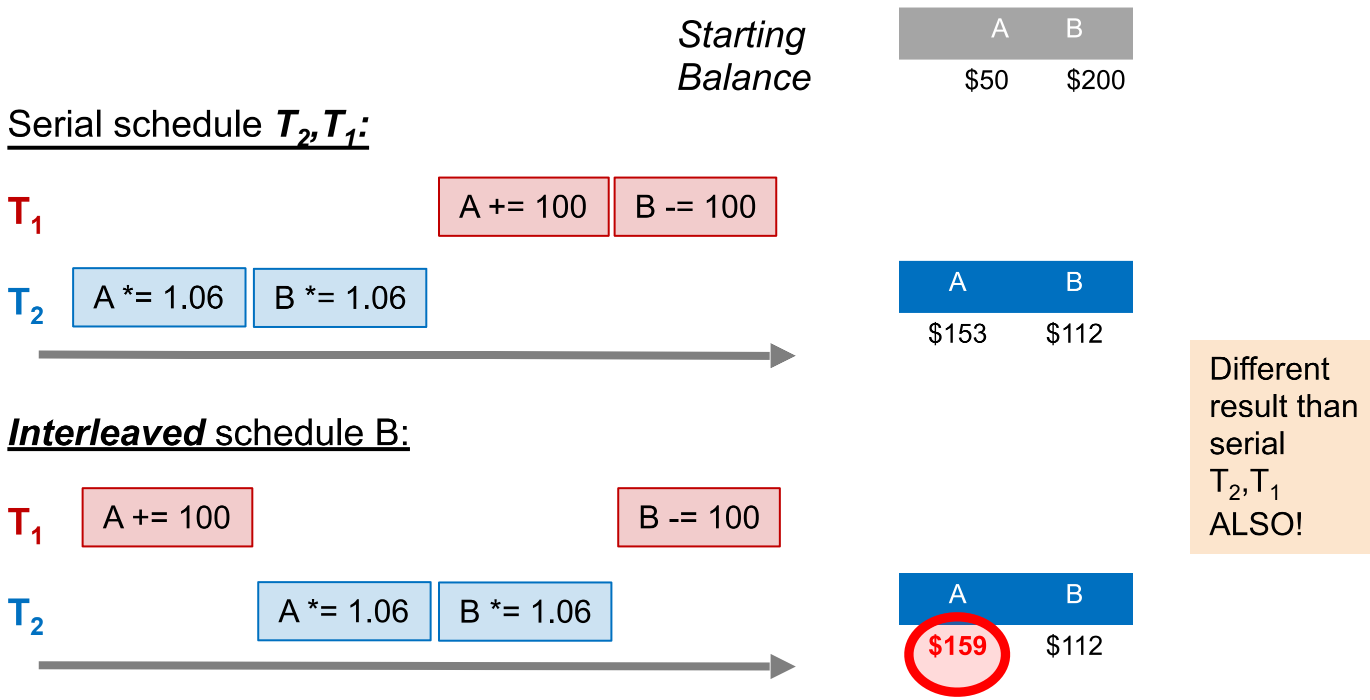
Scheduling examples



Scheduling examples

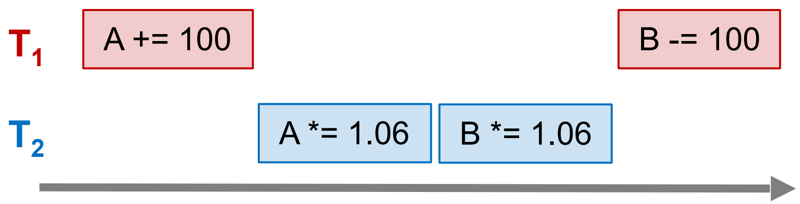


Scheduling examples



Scheduling examples

Interleaved schedule B:



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

Scheduling Definitions

- A **serial schedule** 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** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some"** makes this definition powerful & tricky!

Serial Schedules

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

S1

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

S2

Interleaved Schedules

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

S3

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

S4

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

S5

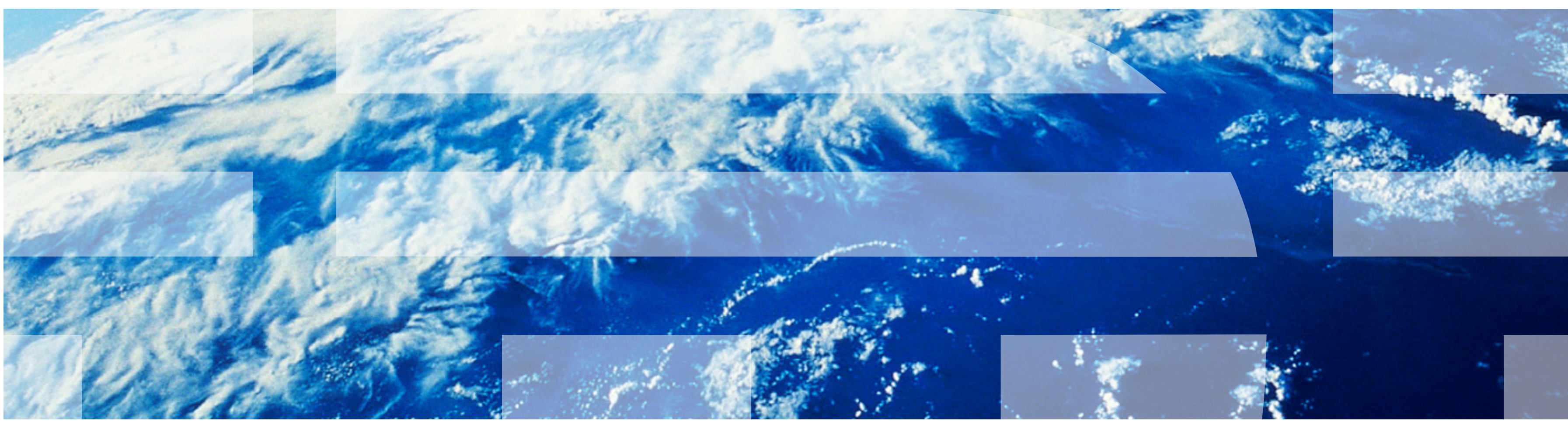
T1	A += 100			B -= 100
T2		A *= 1.06	B *= 1.06	

S6

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

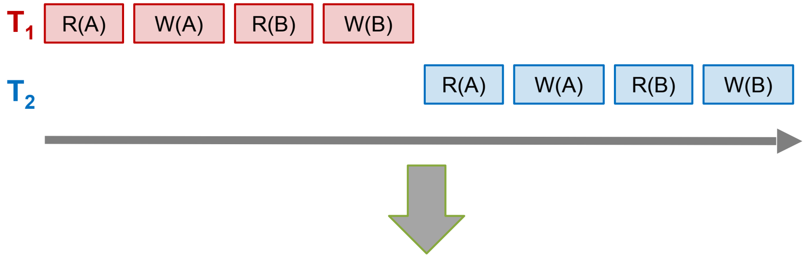
https://www.educative.io/blog/taylor-swift-ticketmaster-meltdown

Conflicts and Anomalies

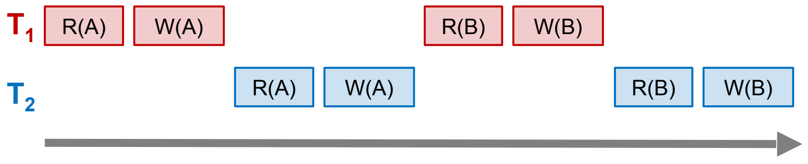


General DB model:
Concurrency as Interleaving TXNs

Serial Schedule



Interleaved Schedule



Each action in the TXNs *reads a value* and then *writes one back*

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

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

Conflict Types

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

Thus, there are three types of conflicts:

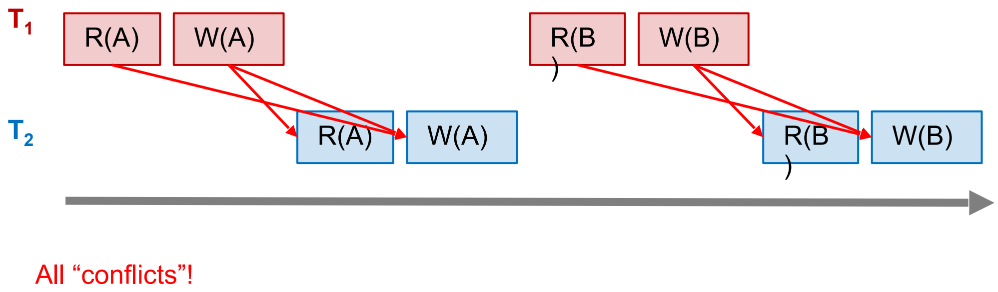
- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

Why no "RR Conflict"?

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

Conflicts

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



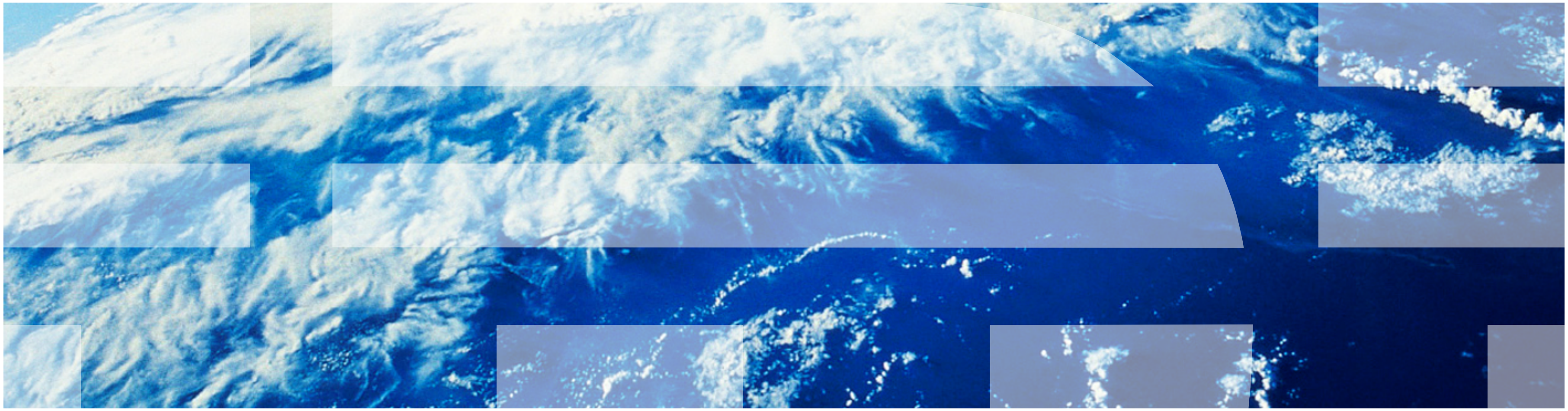
Note: Conflicts vs. Anomalies

<u>Conflicts</u> are in both “good” and “bad” schedules
(they are a property of transactions)

Goal: Avoid <u>Anomalies</u> while interleaving transactions with conflicts!

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

Conflict Serializability



Conflict Serializability

Two schedules are **conflict equivalent** if:

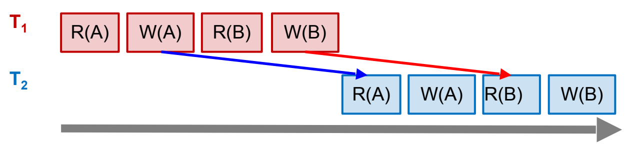
- *Each TXN's order of operations is the same*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

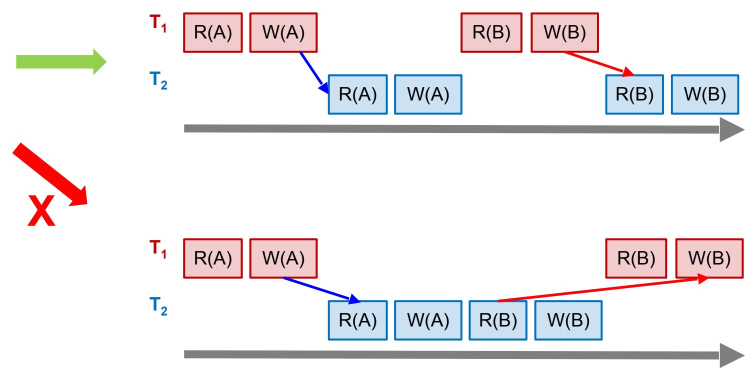
Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation!

Example “**Good**” vs. “**bad**” schedules

Serial Schedule:



Interleaved Schedules:

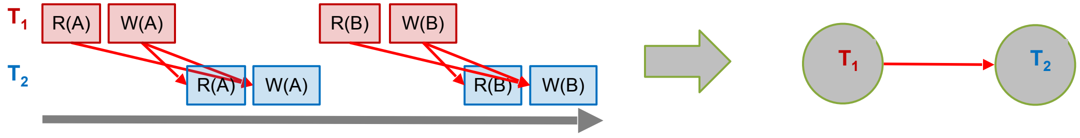


Note that in the “bad” schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

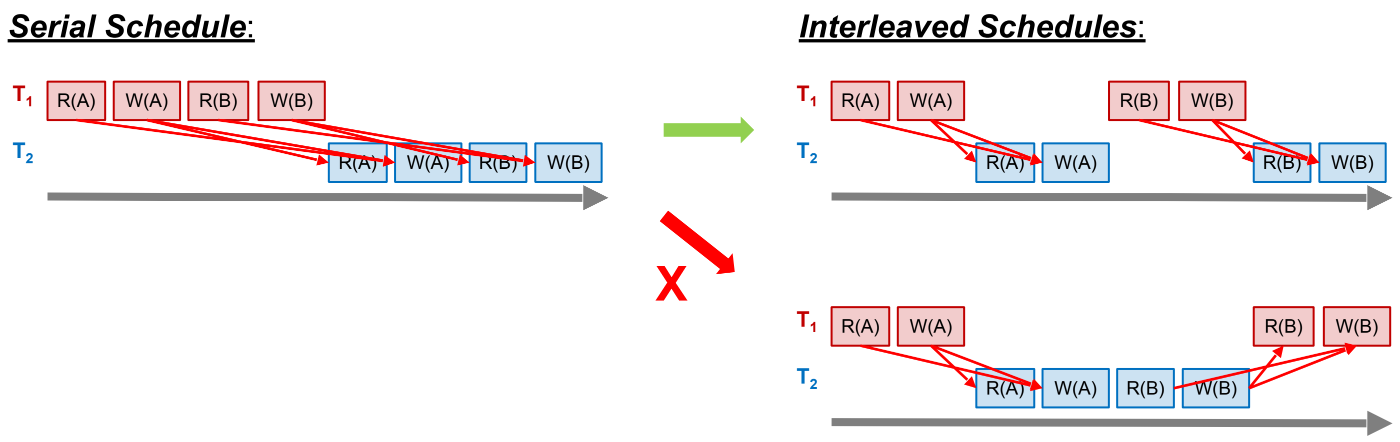
Conflict serializability provides us with an operative notion of “good” vs. “bad” schedules! “Bad” schedules create data Anomalies

The Conflict Graph

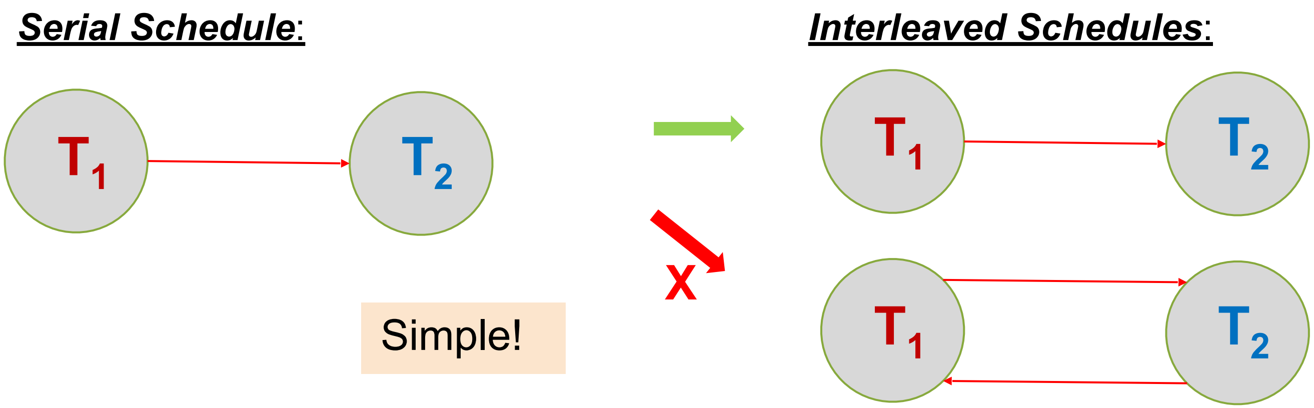
- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i <u>precede and conflict with</u> any actions in T_j**



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



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



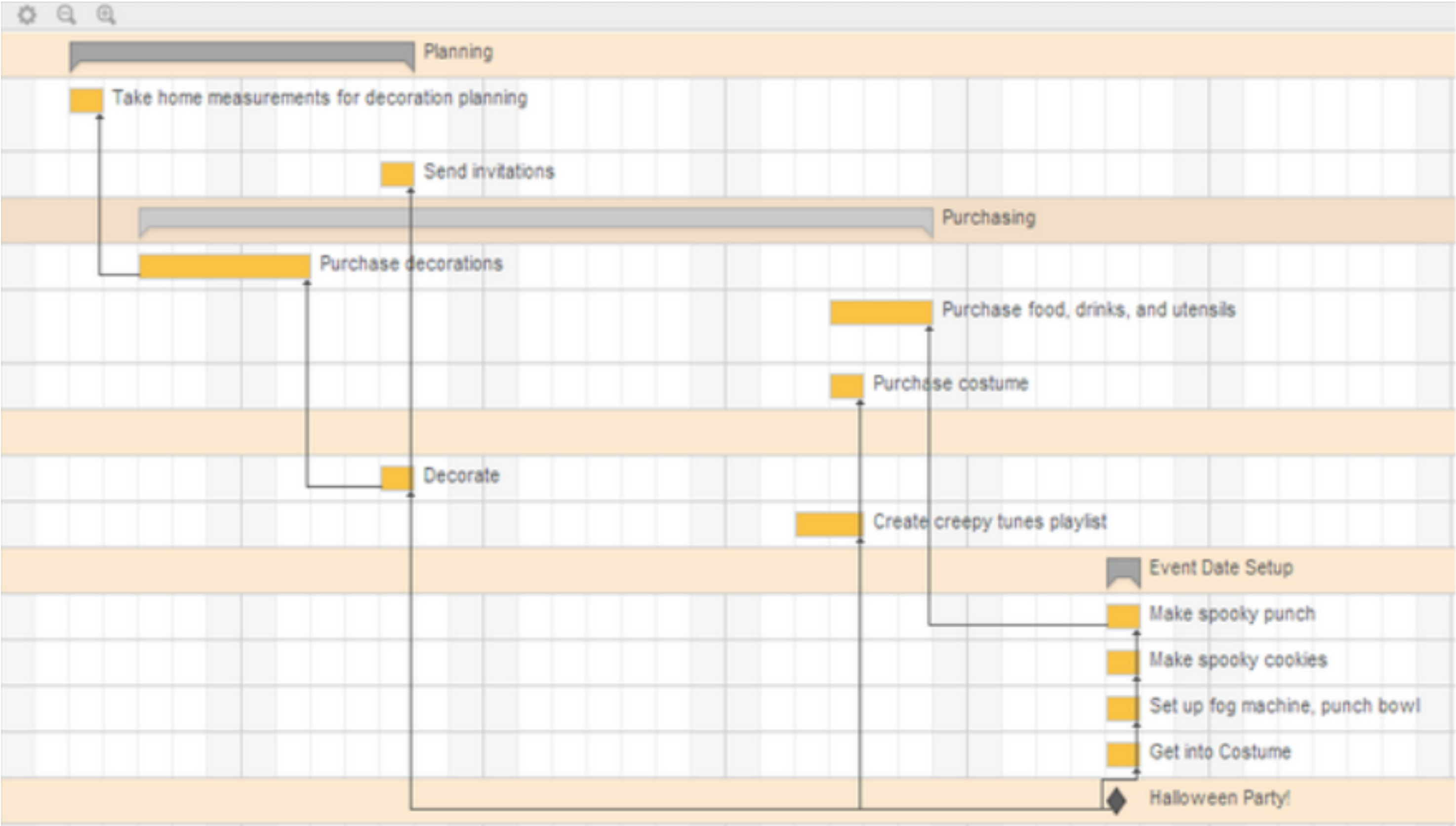
Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
  - E.g., if vertex i has a directed edge to vertex j, in any topological ordering vertex i would appear before vertex j

- A directed **<u>acyclic</u>** graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

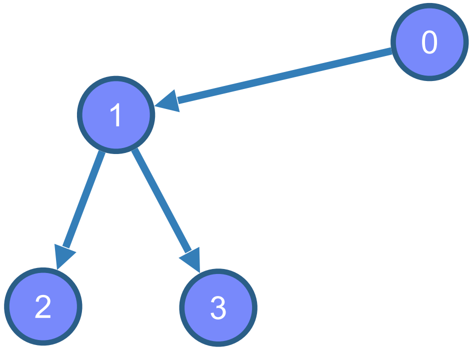
Example: Project dependencies



How would you plan?
What if there are cycles? (dependencies)

DAGs & Topological Orderings

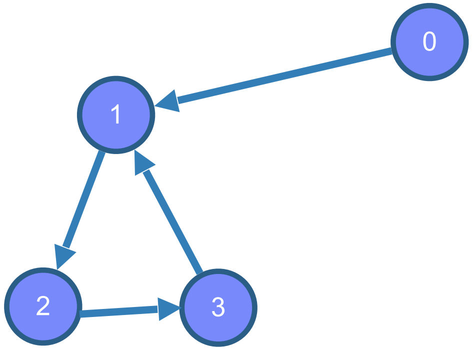
- Ex: What is one possible topological ordering here?



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

DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



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 5 transactions

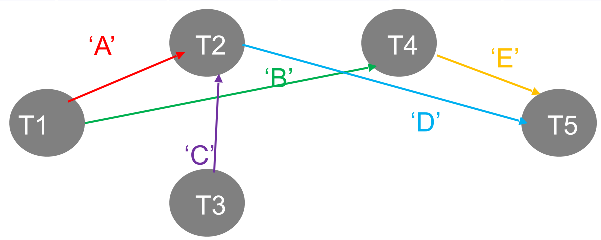
Schedule S1

Good or Bad schedule?
Conflict serializable?

	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)

<u>Step1</u>
Find conflicts
(RW, WW, WR)

<u>Step2</u>
Build Conflict graph
Acyclic?



Acyclic
⇒ Conflict serializable!
⇒ Serializable

<u>Step3</u>
Example serial schedule
Conflict Equiv to S1

S2

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

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**

A few parting observations

- Often, we can construct many conflict serializable schedules
  - Increased flexibility/degrees of freedom are great!
  - We can choose the best performing among the serial schedules
- How many transactions should we schedule at once?
  - The more transactions we schedule in a batch --> higher concurrency and throughput, more possible schedules and degrees of freedom
  - But…
    - Higher latency
    - The scheduler takes longer and becomes more complex