# Local Transactions

## COMS4113: Fundamentals of Distributed Systems

Instructor: Hubertus Franke

Columbia University

https://frankeh.github.io/cu-ds1-2026-fall

# Transactions: A Primer

The abstraction, and what it hides

# Context



- We now **break from the distributed setting** to introduce **transactions**, a core concept in state management, and to see how they are implemented in a **single-node** system.
- Afterwards we return to the distributed setting and describe how **distributed transactions** are implemented.
- As part of that we will discuss **atomic commitment** and **consensus** protocols.

# Why Transactions?



- A key component of any distributed application is a (distributed) **database** that maintains shared state.
- Two challenges of building even a **non-distributed** DB:
  - **Handling failures** — failures are inevitable, and they create the potential for *partial* computations, plus the question of what is correct after restart.
  - **Handling concurrency** — concurrency is vital for performance (I/O is slow, so overlap it with computation), but it creates **races**, so some form of synchronization is needed.

# Transaction



- A **Turing-award-winning** idea.
- An abstraction provided to programmers that **encapsulates a unit of work** against a database.
- Guarantees that the unit of work is executed **atomically** in the face of failures, and is **isolated** from concurrency.

*Jim Gray received the 1998 ACM Turing Award for his work on transaction processing; see [Gray-1978] on the closing slide.*

# Transaction API



Simple, but very powerful:

| | |
|---|---|
| `txID = Begin()` | Starts a transaction. Returns a unique ID for the transaction. |
| `outcome = Commit(txID)` | Attempts to commit. Returns whether the commit succeeded. If it did, **all** operations in the transaction have been applied to the DB; if it did not, **none** of them has. |
| `Abort(txID)` | Cancels all operations of the transaction and erases their effects on the DB. Can be invoked by the programmer **or by the database engine itself**. |

# Semantics



By wrapping a set of accesses in a transaction, the database can **hide failures** and **concurrency** under meaningful guarantees.

One such set of guarantees is **ACID**:

**A** — **Atomicity** — Either all operations in the transaction complete successfully (**commit**), or none of them does (**abort**) — regardless of failures. ← hides **failures**

**I** — **Isolation** — A transaction's behaviour is not impacted by the presence of concurrently executing transactions. ← hides **concurrency**

**D** — **Durability** — The effects of committed transactions **survive failures**. ← survives **crashes**

Where is the **C**? Atomicity, isolation and durability are properties the *database* enforces. **Consistency** is the *application's* invariant — the DB has no mechanism for it, which is why this lecture has mechanisms for A, I and D only.

# Example: Without Transactions



```
TRANSFER(src, dst, x)
01   src_bal = Read(src)
02   if (src_bal > x):
03       src_bal -= x
04       Write(src_bal, src)
05       dst_bal = Read(dst)
06       dst_bal += x
07       Write(dst_bal, dst)
```

Invocation: `TRANSFER(A, B, 50)`

```
REPORT_SUM(acc1, acc2)
01   acc1_bal = Read(acc1)
02   acc2_bal = Read(acc2)
03   Print(acc1_bal + acc2_bal)
```

Invocation: `REPORT_SUM(A, B)`

**Without transactions:** what could go wrong?

**Crash after line 04**
Money leaves src and never arrives at dst. The DB is left in a state no correct execution could produce.

**REPORT_SUM between 04 and 07**
It reads src *after* the debit and dst *before* the credit, and prints a total that is 50 short.

# Example: With Transactions



```
TRANSFER(src, dst, x)
00   txID = Begin()
01   src_bal = Read(txID, src)
02   if (src_bal > x):
03       src_bal -= x
04       Write(txID, src_bal, src)
05       dst_bal = Read(txID, dst)
06       dst_bal += x
07       Write(txID, dst_bal, dst)
09       return Commit(txID)
10   Abort(txID)
11   return FALSE
```

```
REPORT_SUM(acc1, acc2)
00   txID = Begin()
01   acc1_bal = Read(txID, acc1)
02   acc2_bal = Read(txID, acc2)
03   Print(acc1_bal + acc2_bal)
04   Commit(txID)
```

The same code, wrapped.

Begin/Commit/Abort tell the DB where the unit of work starts and ends — and that is the whole of what the programmer has to do.

Both hazards are now the **database's** problem. It must make TRANSFER all-or-nothing across a crash, and it must stop REPORT_SUM from observing the half-done state — without being told which lines conflict.

# Implementing Transactions



- **Atomicity and durability**
  - Operations either all succeed or none do, despite *temporary* failures of the process or machine running the DB (assume the disk itself does not fail). If they succeed, they **persist**.
  - Key mechanism is **write-ahead logging**: log enough information about each operation to disk *before* applying it to the database, so that a failure mid-transaction can be undone.
- **Isolation**
  - Operations in a transaction all witness the database in a coherent state, independent of other transactions.
  - Key mechanism is **locking**: the DB acquires locks on rows read or written and holds them until the end of the transaction.

# Two Mechanisms

Named here, developed next

# Two Key Mechanisms



Everything ACID asks for is delivered by two mechanisms:

| Two-phase locking (2PL) | Write-ahead logging (WAL) |
| --- | --- |
| A **grow-then-shrink** discipline on locks: acquire everything you need, and only then start releasing. | Record the change in an on-disk **log** *before* applying it, so any partial work can be undone — or redone. |
| delivers **I**solation | delivers **A**tomicity + **D**urability |

They are **independent**: locking says nothing about crashes, and logging says nothing about concurrency. A real DB needs both.

*Both mechanisms are developed in full later in this lecture — 2PL from the next slide, WAL from the write-ahead logging section.*

# Two-Phase Locking (2PL)

Isolation, and what it costs

# Lock-Based Concurrency Control

```
TRANSFER(src, dst, x)
00   txID = Begin()
01   src_bal = Read(txID, src)
02   if (src_bal > x):
03       src_bal -= x
04       Write(txID, src_bal, src)
05       dst_bal = Read(txID, dst)
06       dst_bal += x
07       Write(txID, dst_bal, dst)
09       return Commit(txID)
10   Abort(txID)
11   return FALSE
```

```
REPORT_SUM(acc1, acc2)
00   txID = Begin()
01   acc1_bal = Read(txID, acc1)
02   acc2_bal = Read(txID, acc2)
03   Print(acc1_bal + acc2_bal)
04   Commit(txID)
```

**Questions:** what locks to take, when to take them, and how long to keep them?

The next three slides answer the last question by getting it wrong twice. **Option 1** keeps one lock too long, **option 2** releases too early, and **2PL** is what is left.

# Option 1: One Global Lock



```
TRANSFER(src, dst, x)
00   txID = Begin()
01   src_bal = Read(txID, src)
02   if (src_bal > x):
03       src_bal -= x
04       Write(txID, src_bal, src)
05       dst_bal = Read(txID, dst)
06       dst_bal += x
07       Write(txID, dst_bal, dst)
09       return Commit(txID)          <- unlock(table)
10   Abort(txID)                      <- unlock(table)
11   return FALSE
```

<- lock(table)

```
REPORT_SUM(acc1, acc2)
00   txID = Begin()
01   acc1_bal = Read(txID, acc1)
02   acc2_bal = Read(txID, acc2)
03   Print(acc1_bal + acc2_bal)
04   Commit(txID)
```

REPORT_SUM does the same: `lock(table)` at 00, `unlock(table)` after `Commit`.

**Problem?**

**Problem: poor performance.**

Serializes *all* transactions against that table, even ones that never touch the same rows.

# Option 2: Release After Access



```
TRANSFER(src, dst, x)
00   txID = Begin()
01   src_bal = Read(txID, src)          <- lock(src)
02   if (src_bal > x):
03       src_bal -= x
04       Write(txID, src_bal, src)      <- unlock(src)
05       dst_bal = Read(txID, dst)      <- lock(dst)
06       dst_bal += x
07       Write(txID, dst_bal, dst)      <- unlock(dst)
09       return Commit(txID)
10   Abort(txID)
11   return FALSE
```

**Problem?**

**Problem: insufficient isolation.**

Another transaction can read `src` *after* the debit and `dst` *before* the credit — the missing-$50 report from earlier.

`REPORT_SUM(src, dst)`

*the window*

Releasing `src` before taking `dst` opens a window in which the database is **visibly** half-updated. Correctness needs the two locks to *overlap*.

# Two-Phase Locking (2PL)



```
TRANSFER(src, dst, x)
00   txID = Begin()
01   src_bal = Read(txID, src)
02   if (src_bal > x):
03       src_bal -= x
04       Write(txID, src_bal, src)
05       dst_bal = Read(txID, dst)
06       dst_bal += x
07       Write(txID, dst_bal, dst)
09       return Commit(txID)
10   Abort(txID)
11   return FALSE
```

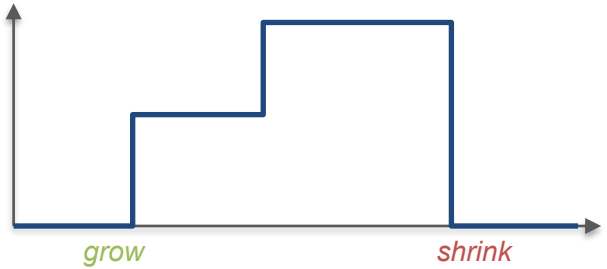
<- lock(src)

<- lock(dst)

<- unlock(src,dst)

<- unlock(src,dst)

**Locks held**



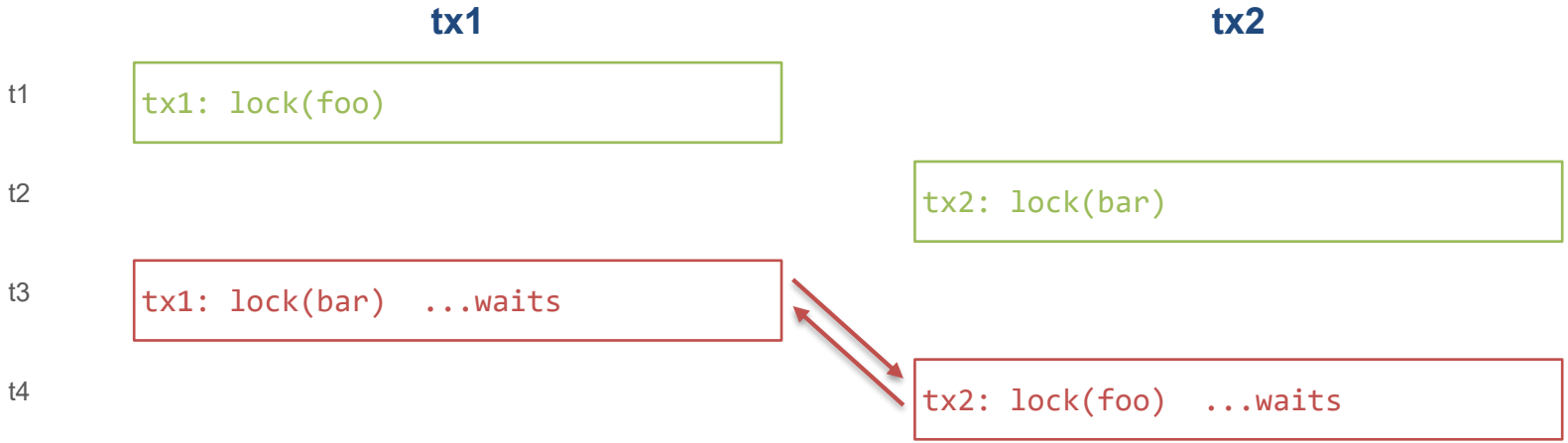

**Phase 1** — acquire locks.
**Phase 2** — release locks.
You may not acquire *any* lock after releasing one.

Usually implemented by the **database** releasing all locks automatically at the end of `commit()` or `abort()` — so the peak is held to the very end.

**Problems?**

# 2PL Can Lead to Deadlocks






**tx1** holds `foo` and wants `bar`; **tx2** holds `bar` and wants `foo`. Neither will ever release, so both wait forever. Nothing is wrong with either transaction — the deadlock is a property of the *pair*, which is why no amount of local checking can prevent it.

# Preventing Deadlock



- **Option 1:** each transaction acquires all its locks at once.
  - Not always possible — with foreign-key navigation the rows to lock are only determined at runtime, so you cannot name them up front.
- **Option 2:** each transaction acquires its locks in a predefined global order.
  - As before, not always possible, and it forces every developer to know the global order.
- **Typically:** allow deadlocks to happen, then **detect** them and **abort** transactions as needed to break them.

# Deadlock Detection and Resolution

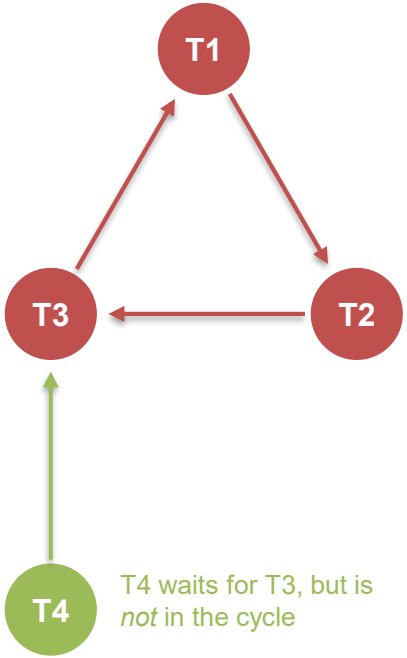


Construct a **waits-for graph of transactions**:

Each **vertex** is a running transaction — **T1**...**T4** below.

An **edge** T1 -> T2 means *T1 is waiting for a lock that T2 holds*.

There is a deadlock **iff** the graph has a **cycle**.




**Cycle T1 -> T2 -> T3 -> T1** — all three are stuck.

To resolve, the database **unilaterally** calls `Abort()` on one or a few transactions to break the cycle.

Aborting **T2** alone is enough — one victim, chosen to minimise lost work.

# To Remember



For concurrency control, a database may decide **on its own** to kill ongoing client transactions.

So **Abort()** is a critical function: it addresses both concurrency-control *and* atomicity problems.

But how exactly do you **Abort()** a transaction that has already written to the database?

**Answer: write-ahead logging.**

# What to Lock: Granularity



2PL says *when* to release locks. It does not say **what** a lock covers:

| | | |
|---|---|---|
| **Table** | one lock per table | cheap bookkeeping, no concurrency |
| **Page** | one lock per disk page | a compromise; false conflicts between unrelated rows |
| **Row** | one lock per tuple | real concurrency, and many more locks to track |

Finer locks mean more concurrency and more overhead. Systems therefore **escalate**: start row-level, and switch to a coarser lock once one transaction holds too many.

Also two **modes**, not one: a **shared** lock for readers (many at once) and an **exclusive** lock for writers (one). Option 1's failure was partly that it made readers exclusive.

# Write-Ahead Logging (WAL)

Atomicity and durability, from a log

# Write-Ahead Logging



In addition to evolving state in RAM and on disk, keep a separate, **on-disk log** of all operations:

transaction **begin / commit / abort**

all **updates** — `X = X - $20`

A transaction's operations are **provisional** until the *commit* record is on disk — results are not revealed to other clients in the meantime.

**Observation:** disk writes of a **single** page or block are atomic — but writes **across** pages may not be.

That is the whole reason the log exists: a **single** atomic write of a commit record can make an **arbitrarily large** set of changes durable at once.

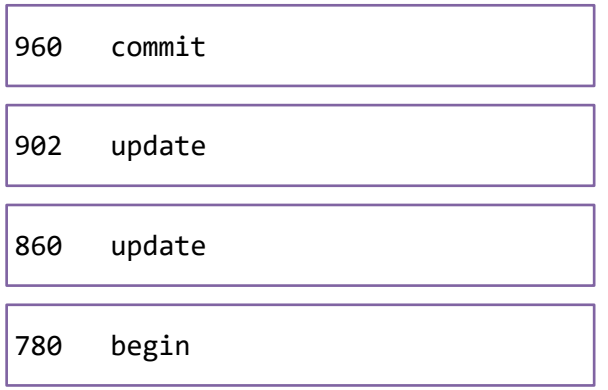
# Log Records: begin / commit / abort



| | |
|---|---|
| `LSN` | **Log Sequence Number** — usually implicit: the address of the first byte of the log entry. |
| `prevLSN` | LSN of the **previous record for this transaction** — a linked list threading each transaction's records together. |
| `xID` | Which transaction this record belongs to. |
| `type` | The operation: begin, commit, abort, update. |

The LSN is an **address**, not a counter: it identifies where the record sits in the log, so a page can cheaply record the last LSN that touched it (slide 32).

### The prevLSN chain




*Each record points back at the same transaction's previous one — this is how undo walks a transaction backwards.*

# Log Records: updates



An update record needs **everything required to undo and to redo** the change:

| `prevLSN + xID + type` |
| --- |
| as before |

| **the update itself** | |
| --- | --- |
| `page ID` | which page |
| `offset` | where in the page |
| `length` | how many bytes |

The pair is the key idea.

**old-value** lets recovery roll a change *back*; **new-value** lets it roll a change *forward*. One record, both directions.

**old-value**
*for undo*

**new-value**
*for redo*

This is **physical** (byte-level) logging. Logging the *operation* instead — logical logging — is smaller, and slide 31 returns to it.

# WAL: A Worked Example



```
xId = begin();  // xId <- 42
src.bal  -= 20;
dest.bal += 20;
commit(xId);
```

**Disk:**

| | |
|---|---|
| 11 | src.bal: 100 |
| 14 | dest.bal: 3 |

**Page cache:**

| | |
|---|---|
| 11 | src.bal: 80 |
| 14 | dest.bal: 23 |

**Transaction table:**

42: prevLSN = 780

**Dirty page table:**

11: firstLSN = 860, lastLSN = 860

14: firstLSN = 902, lastLSN = 902

*non-log pages may* stay in memory

**must flush the** log to disk!

**Log:**

780
```
prevLSN:        0
xId:            42
type:           begin
```
:

860
```
prevLSN:        780
xId:            42
type:           update
page/off/len:   11 / 10 / 4
old-val:        100
new-val:        80
```

902
```
prevLSN:        860
xId:            42
type:           update
page/off/len:   14 / 10 / 4
old-val:        3
new-val:        23
```

960
```
prevLSN:        902
xId:            42
type:           commit
```

# The Tail of the Log



The **tail** of the log can be kept in memory until a transaction commits — or until a buffer page is flushed to disk.

Batching the tail is why many transactions can share one disk flush — **group commit**.

# Recovering From a Crash



e.g. a system crash. For now, assume the log itself is readable.

| | | | |
|---|---|---|---|
| 1 | Analyze | Scan the log forward to rebuild the transaction table and the dirty page table: who was running, what was dirty. | *which transactions won, which lost* |
| 2 | Redo | Replay updates **forward**, using **new-value** — for *all* transactions, not just committed ones. | *repeating history* |
| 3 | Undo | Roll back uncommitted transactions **backward**, using **old-value**, following each prevLSN chain. | *losers are erased* |

After all three passes the database is exactly as if every committed transaction had run and no uncommitted one ever started.

# Why Redo All Operations?



Redo replays **even the losers** — transactions that never committed and are about to be undone anyway. Why bother?

## Interaction with concurrency control

Undo needs the database in the **exact state** the log describes. Replaying everything first restores that state, so undo can then walk backwards through a history it recognises.

## It generalizes to logical operations

Any operation with an undo and a redo can be logged — not just byte ranges. Logical logging can be far **smaller and faster** than byte-level.

**Repeating history** is the trick: restore the past exactly, *then* correct it. Trying to be clever and skip the losers is where recovery bugs live.

# The Performance of WAL



**Objection:** must we write the disk **twice**?

**Not always.** The data page write is *deferred* and may be batched with others, or coalesced away entirely if the page is updated again before it is flushed.

For byte-level update records you must also **know the old value** — so an update may require a read first.

**Writing the log is sequential.**
One append-only stream instead of scattered random writes — this may *improve* performance outright.

**Acknowledge early.**
A commit can be reported to the client as soon as the *log* is written — the data pages can wait.

So WAL trades **random** writes for **sequential** ones, and trades write *volume* for write *latency*. On spinning disks that was decisive; on SSDs the sequential-write advantage is smaller but the early-acknowledge advantage remains.

The commit path costs **one sequential flush**. That single fact is why the log, and not the data, defines durability.

# Improvements to This WAL



### Store the LSN of the last write on each data page

**pageLSN.** Recovery compares it against the log record and skips redos already reflected on the page — avoiding unnecessary work.

### Log checkpoint records

Bound how far back recovery must read. Which pages were in memory, and which transactions were live. **Next slide.**

### Log the recovery actions themselves (CLRs)

**Compensation log records** make undo idempotent, so a crash *during* recovery does not redo work already undone.

### Ordered / metadata-only logging

Avoids having to record the old value of file data — useful in file systems, where the metadata is what must stay consistent.

# Checkpoint Records

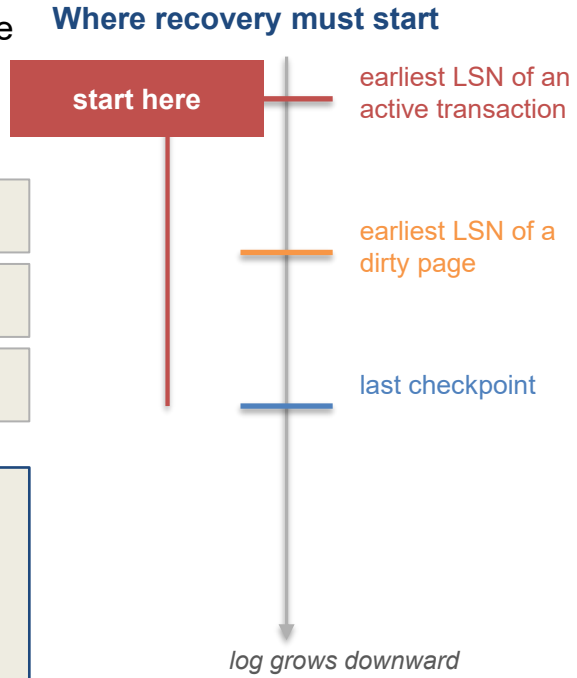


Analysis can **start at the last checkpoint** instead of the beginning of the log.

A checkpoint records:

- the table of **active transactions**
- the table of **dirty pages** in memory
- and the **earliest LSN** that might have affected them

Recovery must begin at the **earliest** of the three — the checkpoint alone is not enough, because a transaction or a dirty page may predate it.



# Transactions in Practice

What a 2026 database actually does

# WAL in Production



Recall the observation: *writes across pages may not be atomic.* A **partial** page write breaks redo, because a byte-level record cannot repair a page that is half old and half new.

| | | |
|---|---|---|
| **PostgreSQL** | `full_page_writes` | Writes the **whole page image** into the WAL on the first change after a checkpoint, so any torn page can be reconstructed. On by default; `wal_compression` exists to pay back the volume. |
| **MySQL / InnoDB** | `doublewrite buffer` | Writes every page **twice** — once to a sequential scratch area, then to its home location. A torn copy is always recoverable from the other. |

## The durability dial is a configuration option, not a law:

| | |
|---|---|
| `synchronous_commit = on` | commit waits for the log flush — no committed work is ever lost |
| `synchronous_commit = off` | commit returns early; a crash may lose **recent commits**, but never corrupts the database |

# Locking Is Not the Only Way



This lecture taught isolation via **locking**. Most databases you will actually use do it differently.

## Locking (2PL)

A reader **blocks** a writer and a writer blocks a reader. Correct, and the source of most contention.

Still used where conflicts are common — and 2PL remains the model to reason with.

## MVCC (Multi Version Concurrency Control)

Keep **multiple versions** of each row. A reader gets a consistent **snapshot** and never blocks; only writer/writer pairs conflict.

The cost moves to storage and to cleaning up old versions.

In **PostgreSQL** today: the default isolation level is **Read Committed**, `REPEATABLE READ` *is* snapshot isolation, and `SERIALIZABLE` is implemented as **Serializable Snapshot Isolation** — snapshots plus conflict detection, **not** two-phase locking.

Consequence for your code: transactions **fail and must be retried** rather than wait. Serialization failures arrive as SQLSTATE **40001**.

# Isolation levels: What can go wrong



- **Dirty read**: A transaction reads uncommitted data modified by another concurrent transaction. If the other transaction rolls back, the data read was never actually valid.
- **Non-repeatable read**: A transaction reads the same row twice and gets different values because another transaction modified and committed that row in between the reads.
- **Phantom read**: A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows has changed due to another recently committed transaction inserting or deleting rows.
- **Write skew**: Occurs when two concurrent transactions read overlapping sets of data, make local decisions based on what they read, and then update separate data items in a way that violates a global constraint (often seen in snapshot isolation).

# Isolation Levels



| | Dirty read | Non-repeatable read | Phantom read | Write skew |
|---|---|---|---|---|
| **Read Uncommitted** | possible | possible | possible | possible |
| **Read Committed** | — | possible | possible | possible |
| **Repeatable Read / snapshot** | — | — | — | possible |
| **Serializable** | — | — | — | — |

**Snapshot isolation permits write skew** — two transactions each read a state the other is about to invalidate, and both commit. This is what SSI adds detection for.

And note what the table is **not** about: every column here is an **isolation** phenomenon. The **C** in ACID — your application's invariants — appears nowhere, because no isolation level enforces it. Serializable protects your invariant only if your transactions were correct in the first place.

# Which Mechanism Buys What



| A | C | I | D |
|---|---|---|---|
| Atomicity | Consistency | Isolation | Durability |
| WAL | you | 2PL / MVCC | WAL |
| undo via old-value; abort is just undo | the DB checks only what you declare | locks, or snapshots plus conflict detection | one sequential flush of the commit record |

Both mechanisms exist because the hardware gives us exactly one primitive: a **single atomic page write**. WAL leverages it; locking has nothing to do with it.

Next lecture the disk becomes a *network*, and the same two problems come back much harder.

# Next



- We return to the distributed setting to discuss:
  - How to implement distributed transactions in a **sharded** database (for scalability): **atomic commitment** protocols — and 2PC's prepare record is a WAL record, so you have already seen the durable state it depends on.
  - How to implement distributed transactions in a **replicated** database (for fault tolerance): **consensus** protocols.
  - Several **case studies** on leveraging these protocols in practice: Spanner, Chubby, Bigtable.

# Key Papers



- **[Mohan-1992]** Mohan, Haderle, Lindsay, Pirahesh & Schwarz. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.* ACM TODS **17(1):94–162**, 1992. — the WAL half of this lecture.
- **[Gray-1978]** Jim Gray. *Notes on Data Base Operating Systems.* Advanced Course on Operating Systems, 1978, pp. **393–481**.
- **[Franklin-1997]** Michael J. Franklin. *Concurrency Control and Recovery.* In *The Computer Science and Engineering Handbook*, 1997, pp. **1058–1077**; revised in *Computing Handbook*, 3rd ed., 2014, ch. **12**.
- **[Bernstein-1987]** Bernstein, Hadzilacos & Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987 — the standard reference; freely available online.
- **[Adya-2000]** Adya, Liskov & O'Neil. *Generalized Isolation Level Definitions.* ICDE 2000, pp. **67–78**.
- **[Cahill-2009]** Cahill, Röhm & Fekete. *Serializable Isolation for Snapshot Databases.* ACM TODS **34(4):20:1–20:42**, 2009 — the basis of PostgreSQL's SERIALIZABLE.