OLTP & DeltaLake

Lecture 7

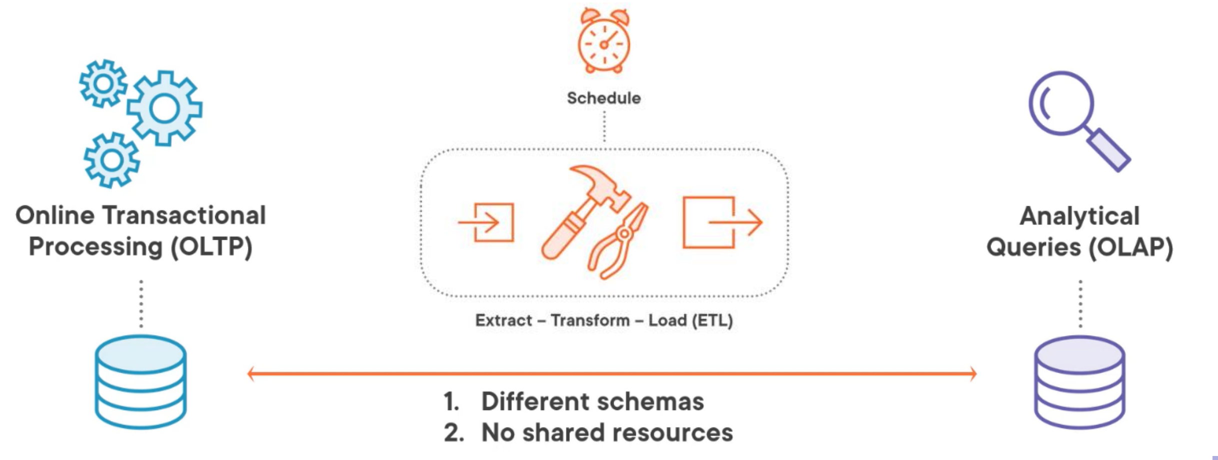
Databases

- Historically databases stored structured data
  - Follows a schema (table, records, keys, …), typically handled by database
- 2 types of workloads: transactional processing, analytical processing
- Transactional workloads (OLTP)
  - High volume of record-keeping txns (many queries that touch few data)
  - ACID properties
- Analytical workloads
  - Lower volume of read-only queries (fewer queries that touch more data)
  - Batch based (warehousing) also called OLAP – data stored and analyzed
  - Streaming based -- data not stored at all, analyzed as it comes in

- Typically, OLAP workloads are used to analyze data generated by OLTP workloads

OLTP – ETL – OLAP: Batch Processing

- Run separate OLTP and OLAP databases
  - OLAP database also called data warehouse
  - Separation helps to optimize each separately
  - Periodically transfer data from OLTP -> OLAP using ETL pipeline
- Pros
  - Repeated quering of data once loaded
  - Good for combining multiple datasets
  - Can be very efficient
- Cons
  - “Stale” data in warehouse



Example Casestudy

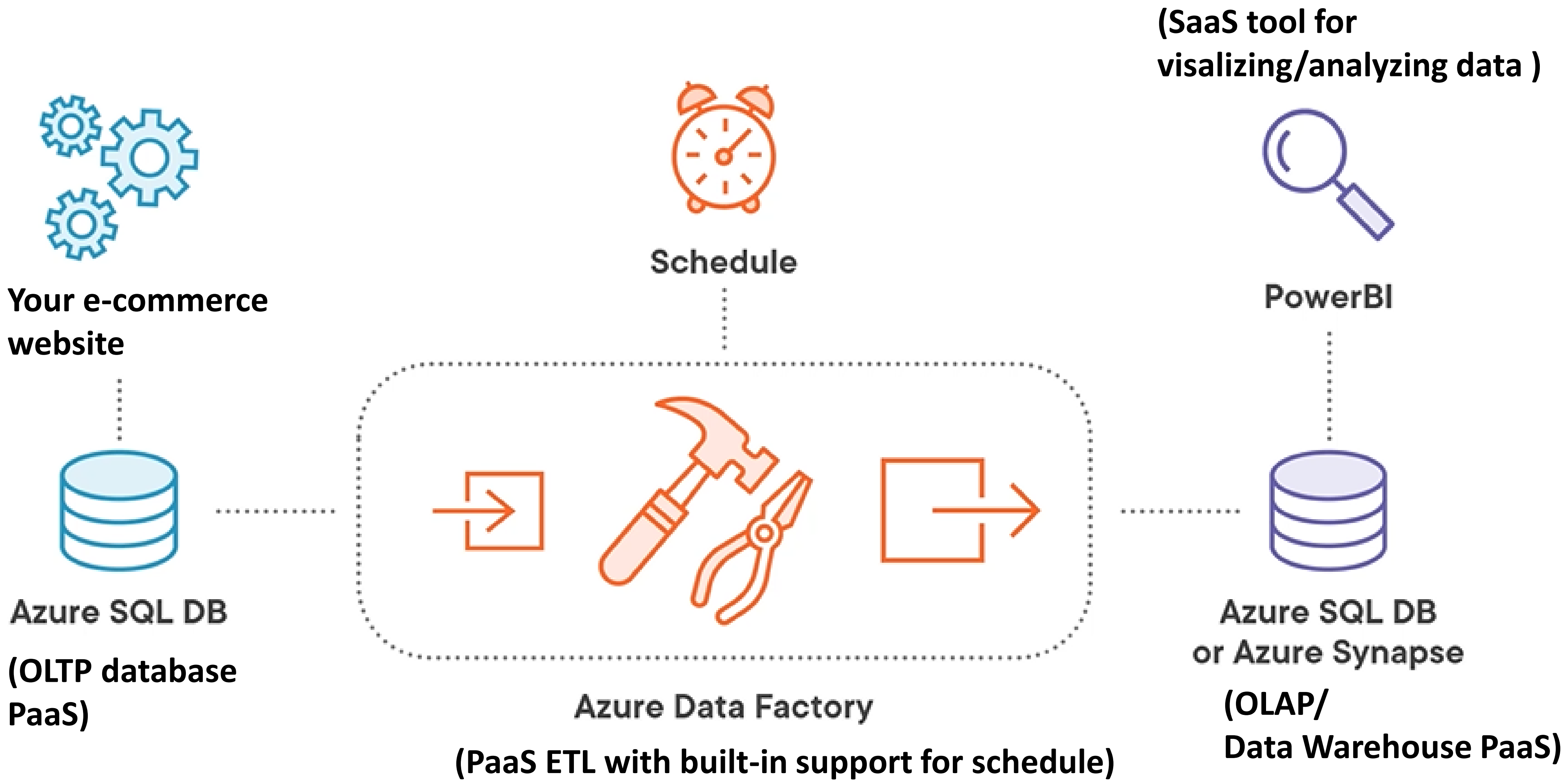
- Consider a e-commerce application with following schema

Customers		
Id	Name	Age
1	Jane Doe	21
2	Tobias Mason	33
3	Huseyn Abbasov	21
4	Gerja Bas	33

Orders			
Id	CustomerId	...	Total
1	1		51.99
2	2		155.97
3	3		51.99
4	4		519.90

- Suppose we want to find “What is average order amount by age?”
  - This is a Business Intelligence (BI) query

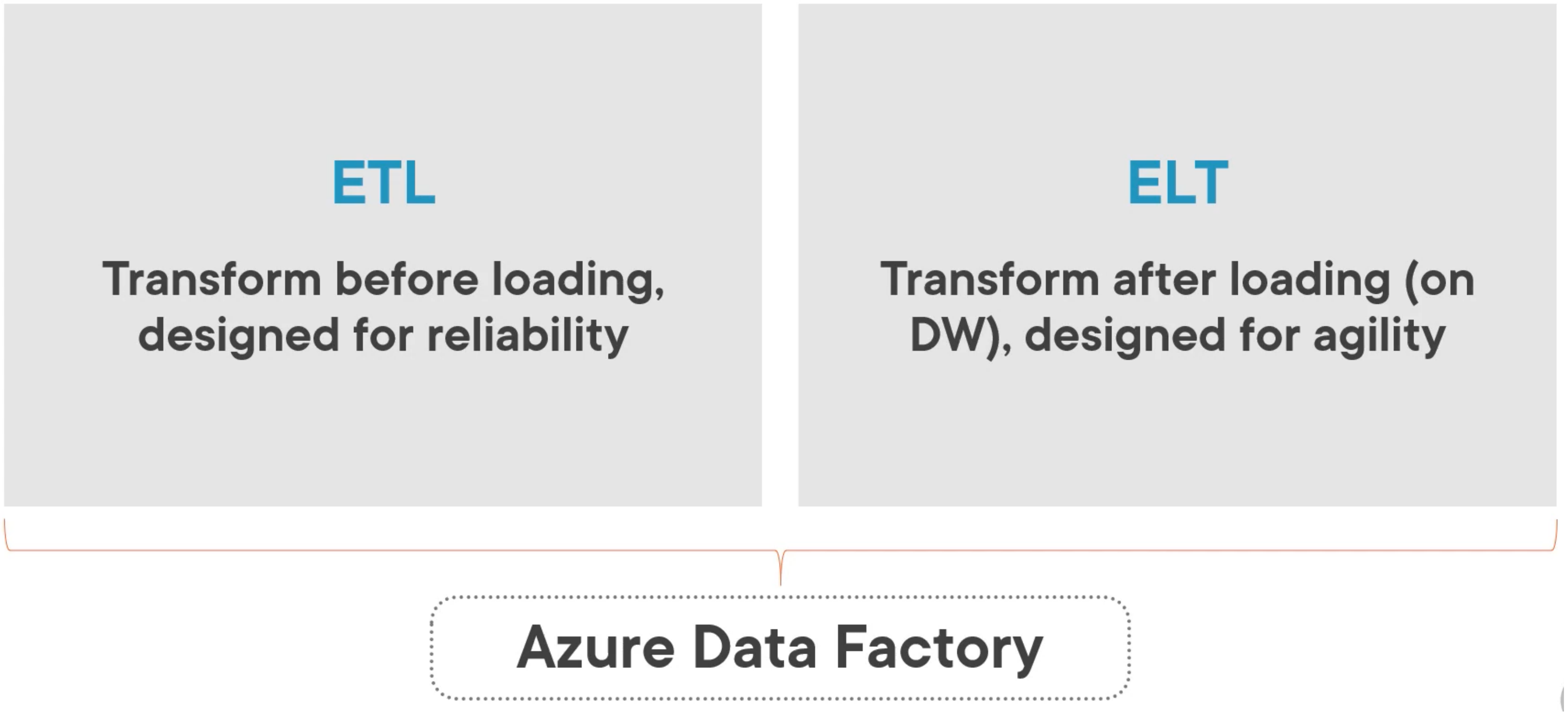
Batch processing on Azure



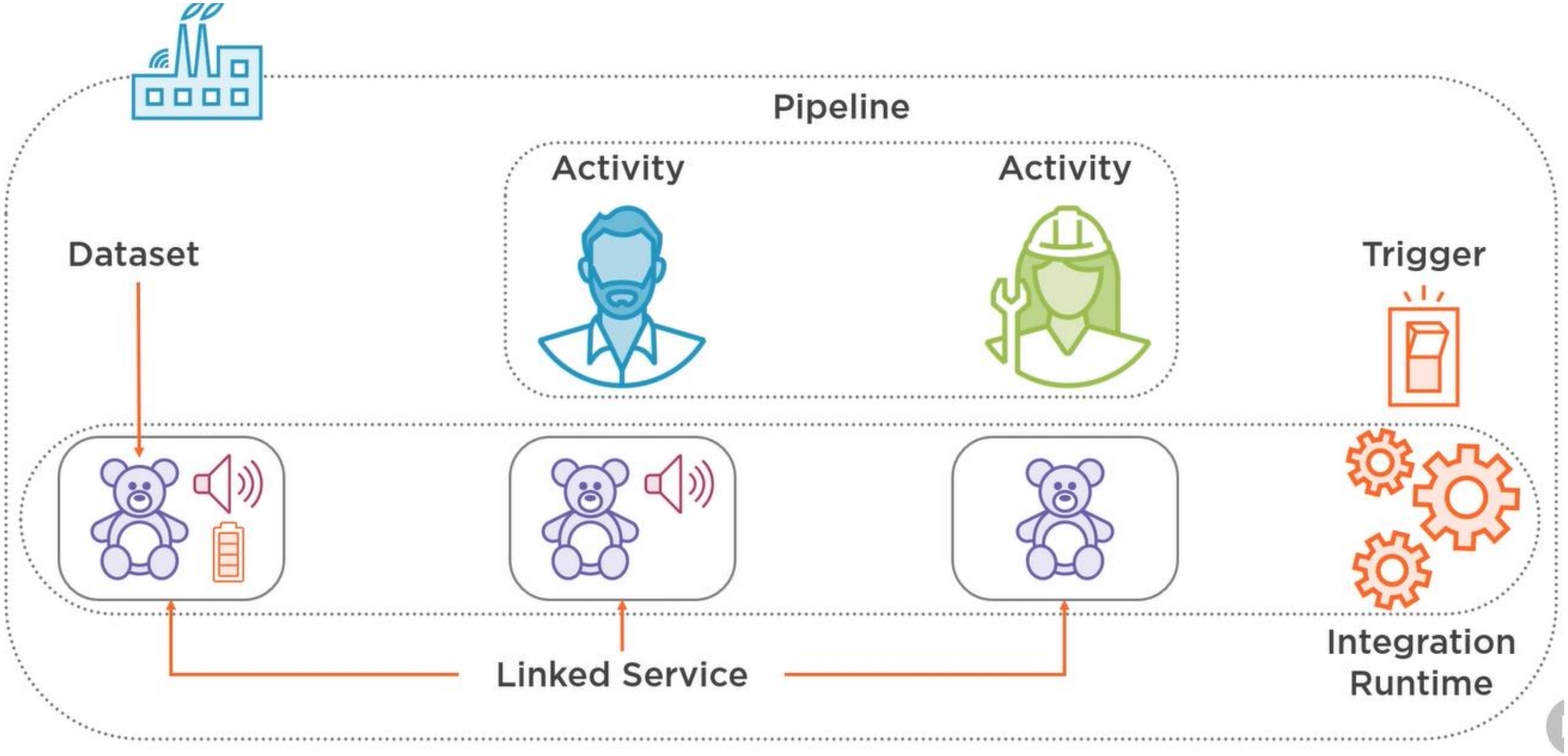
A word about Azure Data Factory

- Cloud-based
  - PaaS service, serverless offering
- Data integration service
  - Consolidate data from multiple sources
- Allows you to orchestrate and automate data movement
  - Connect to > 90 systems and move data between them
- Allows you to perform data transformation
  - Mapping data transformations: code-free transformations over data

ETL vs ELT


ETL
Transform before loading, designed for reliability
ELT
Transform after loading (on DW), designed for agility
Azure Data Factory

Data Factory Elements



Problems with Batch Processing: New Data Types

- Structured data
  - Follows a schema (table, records, keys, …), typically handled by database
  - Ex: Tables
- Unstructured data
  - No structure, often processed using ML to generate structured data
  - Ex: videos, images, audio files
- Semistructured data
  - Has observable structure (not necessarily tabular) but schema not defined
  - Ex: log files (timestamp and some info), CSV, JSON, XML files
  - Can easily change "shape" of data as there is no schema

Database engines had poor support for new types

Upfront schema enforcement an issue

Problems with Batch Processing: New Workloads

- Warehousing & OLAP
  - Data staleness: ETL needs to complete before fresh results in OLAP
  - So only analytical queries for BI that can tolerate data staleness
  - Ex: “What is average order amount by age in last year?”

- Predictive analytics with ML
  - Need to feed data to ML models

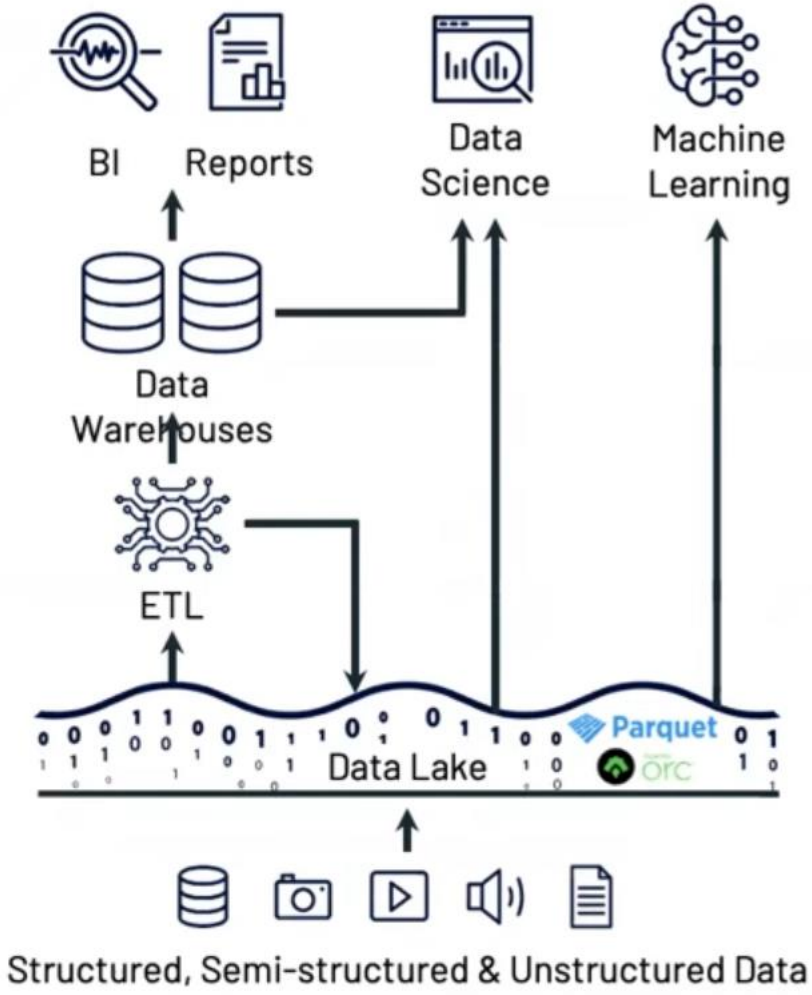
Upfront data loading & SQL interface an issue

High cost (licensing) to scale and support large datasets

Data warehouse to Data Lake

- Low-cost storage
  - Hold all raw data with file API (HDFS)
- Open file formats
  - Ex: Parquet, ORC
  - Data directly accessible to ML engines
- ETL to load data into warehouses
  - Traditional SQL analytics

- Over 90% of enterprise data in data lakes now



Azure Data Lake Gen2

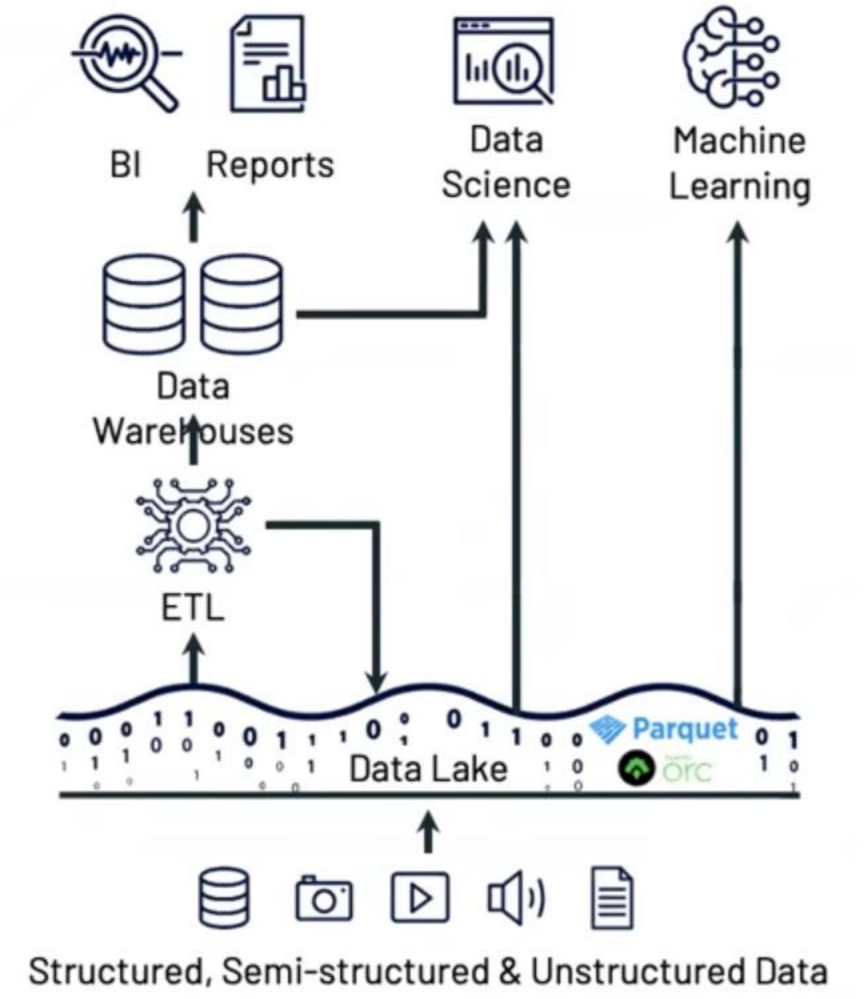
- Build on Azure Blob Storage to offer
  - HDFS compatible access
  - Fast data access with hierarchical organization for analytics
  - Secure data storage with support for POSIX permissions
  - Redundant storage (across nodes, zones, and regions)

Azure Blob Storage	Azure Data Lake Storage
Flat namespace	Hierarchical namespace
Good storage retrieval performance for analytical use cases	Better storage and retrieval performance for analytical use cases
High cost of analysis	Low cost of analysis

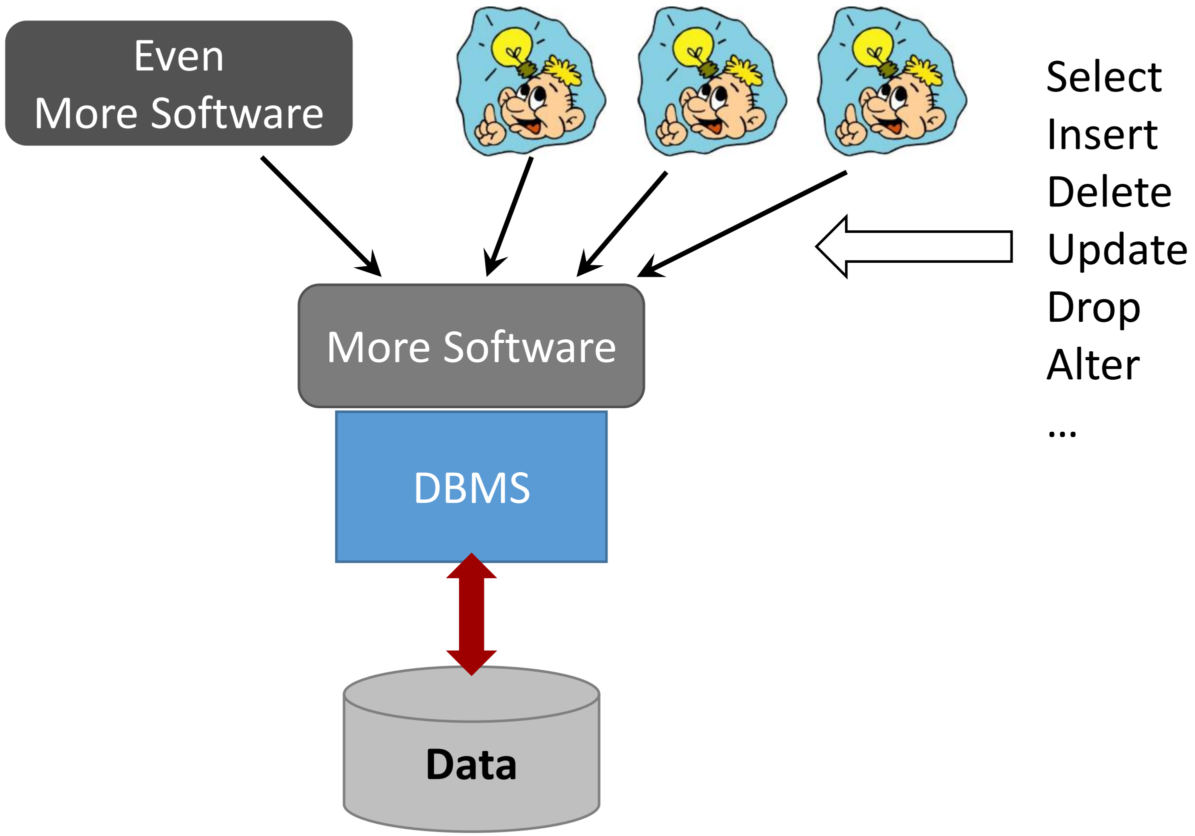
Problems with Traditional Data Lake

- **Lack of transactional updates**

Lets see how databases update data to understand why this is a problem



Concurrent database access



Why concurrency is a problem?

Two concurrently executing queries

X

Accid	balance
101	100
102	1000
104	1000

```
update account
set balance=balance-20
where accid=101
```

fetch; modify; update

$R_1(X)$
$X=X-20$
$W_1(X)$

```
update account
set balance=balance+10
where accid=101
```

Read; modify; Write

$R_2(X)$
$X=X+10$
$W_2(X)$

t0: $R_1(X)$
t1: $X=X-20$
t2: $W_1(X)$
t3: $R_2(X)$
t4:
t5:

X=90

t0: $R_2(X)$
t1: $X=X+10$
t2: $W_2(X)$
t3: $R_1(X)$

X=90

t0: $R_1(X)$
t1: $R_2(X)$
t2: $X=X-20$
t3: $X=X+10$

X=110

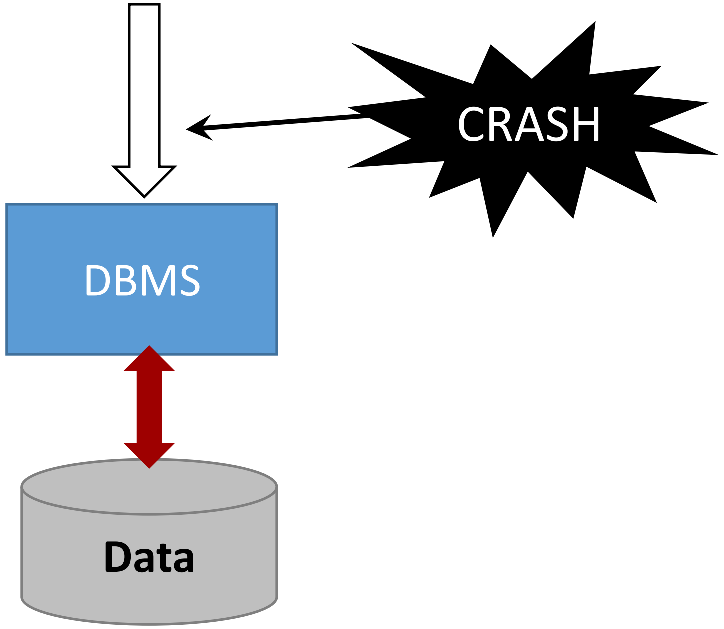
Arbitrary interleaving can lead to inconsistencies

Goal of concurrency control

- Execute sequence of SQL statements so they appear to be running in isolation
- Obvious way: execute them in isolation
  - **Is this acceptable?**
- Enable concurrency whenever possible and safe to do
  - utilization/throughput ("hide" waiting for I/Os)
  - response time
  - fairness

Resilience to system failures

```
update account set balance=balance-50 where accid=101
update account set balance=balance+50 where accid=102
```



Solution to both problems

- Concurrent database access
- Resilience to system failures

Transactions

- A transaction is a sequence of one or more SQL operations treated as a unit
  - Transactions appear to run in isolation
  - If the system fails, each transaction's changes are reflected either entirely or not at all

Correctness: The **ACID** properties

- **A**tomicity: All actions in the transaction happen, or none happen
- **C**onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent
- **I**solation: Execution of one transaction is isolated from that of other transactions
- **D**urability: If a transaction commits, its effects persist

A Atomicity of transactions

- Two possible outcomes of executing a transaction:
  - Transaction might *commit* after completing all its actions
  - or it could *abort* (or be aborted by the DBMS) after executing some actions

- DBMS guarantees that transactions are *atomic.*
  - From user's point of view: transaction always either executes all its actions, or executes no actions at all

A Mechanisms for ensuring atomicity

- One approach: SHADOW PAGING

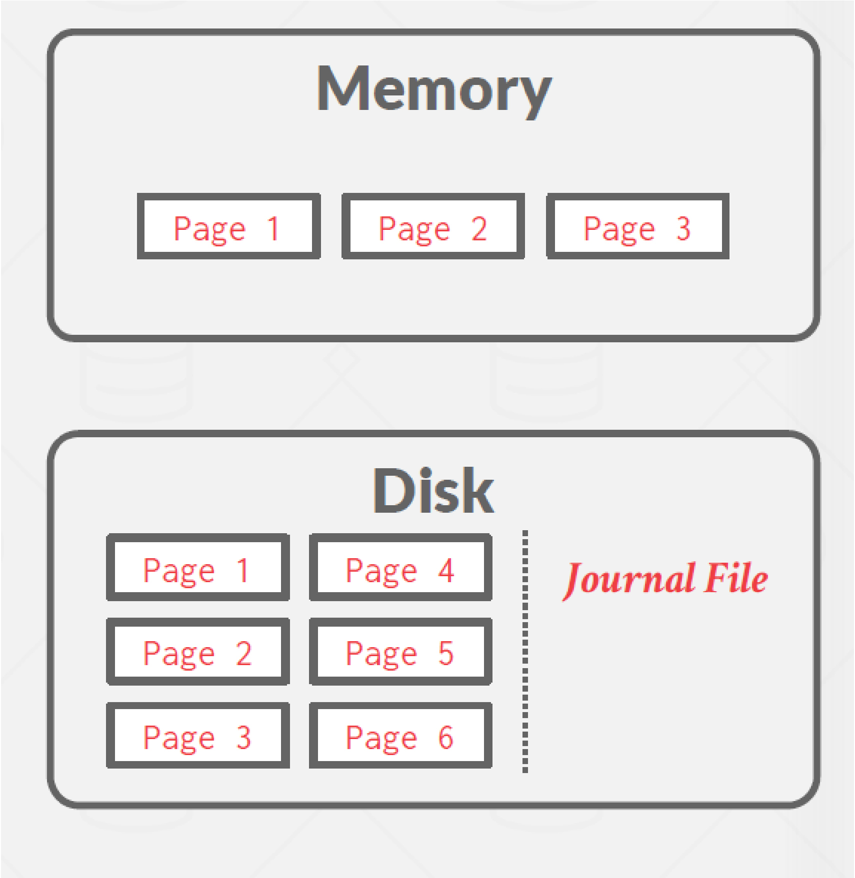
- The DBMS copies pages on write to create two versions:
  - **Master**: Contains only changes from committed txns.
  - **Shadow**: Temporary database with changes made from uncommitted txns.

- To install updates when a txncommits, overwrite the root so it points to the shadow, thereby swapping the master and shadow.

Example: SQLITE (Pre-2010)

- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

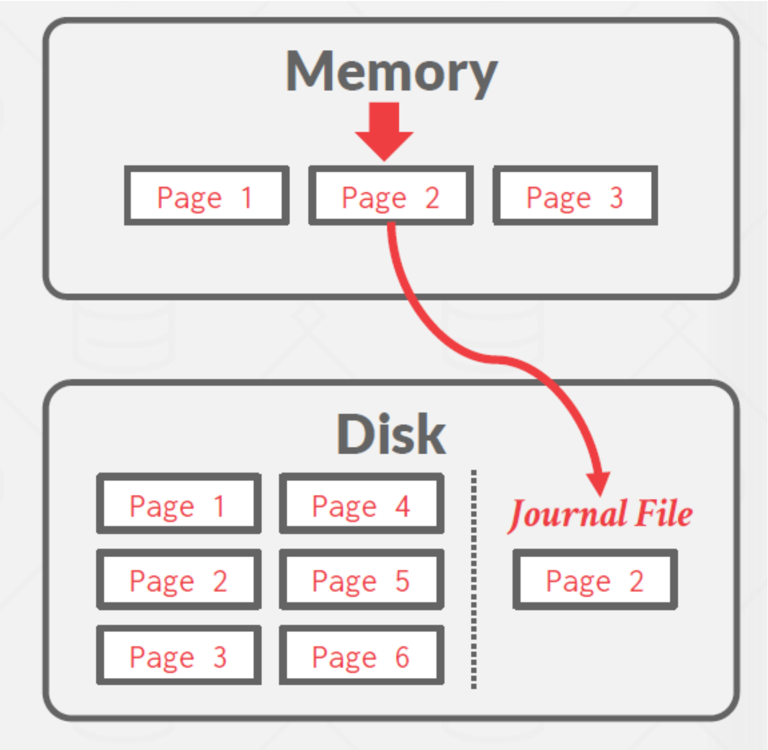
- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

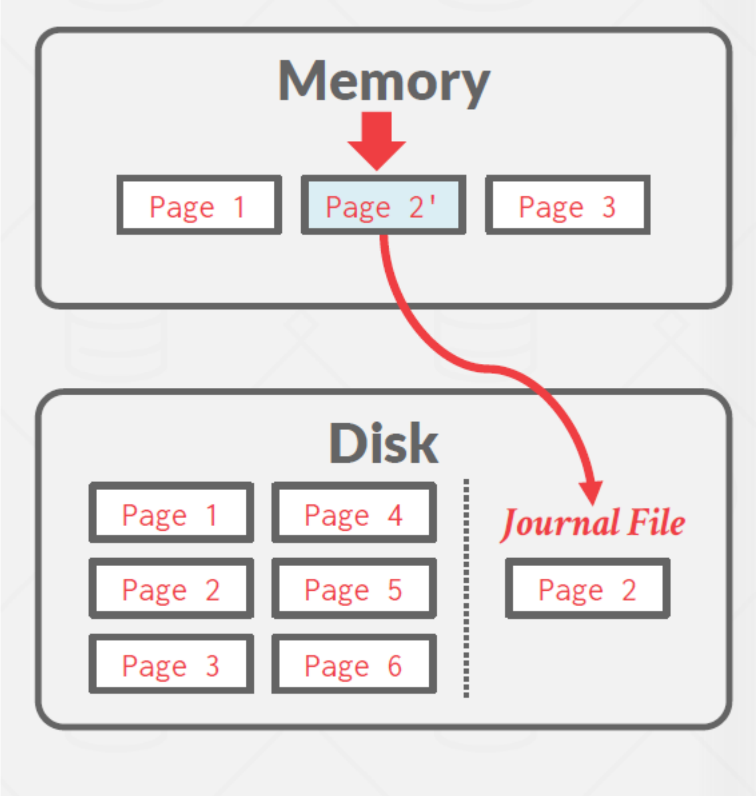
- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

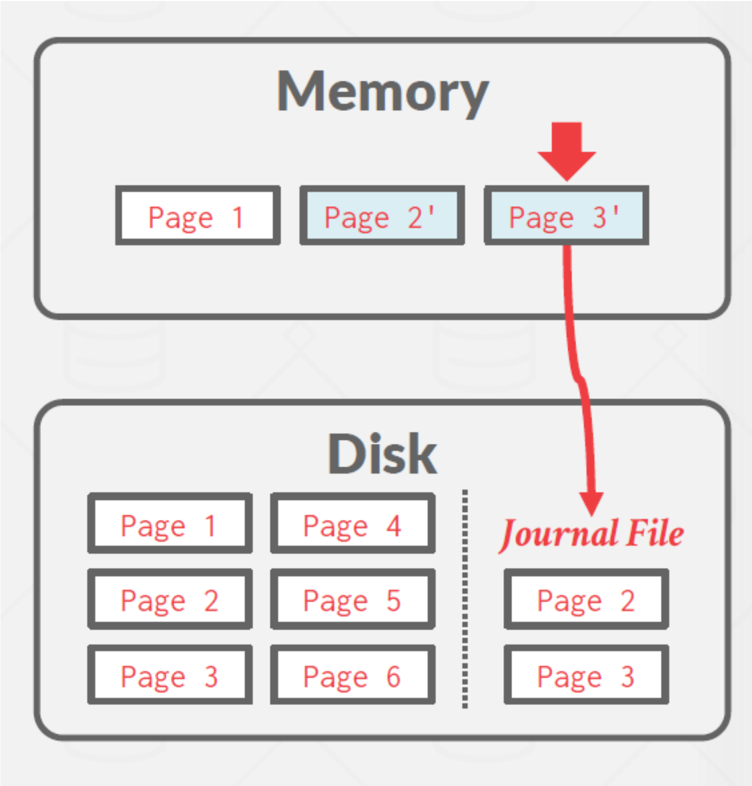
- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

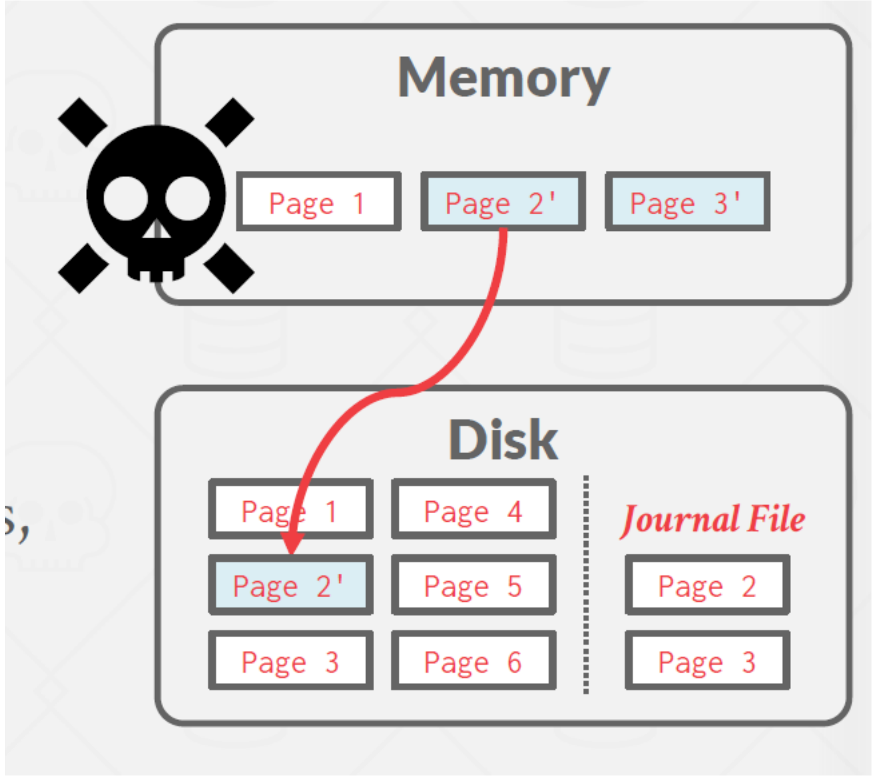
- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

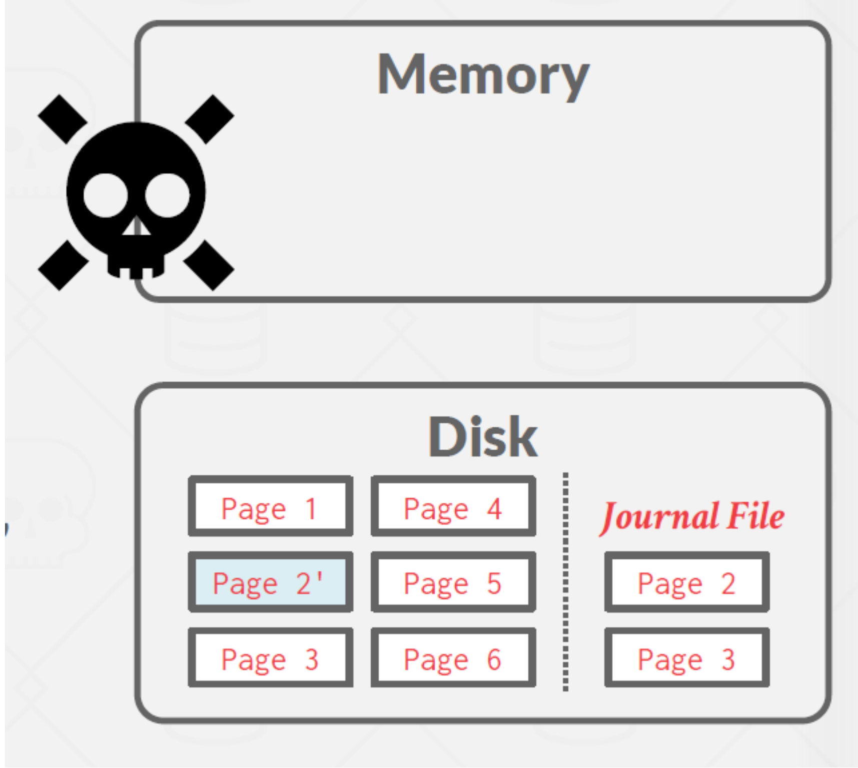
- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

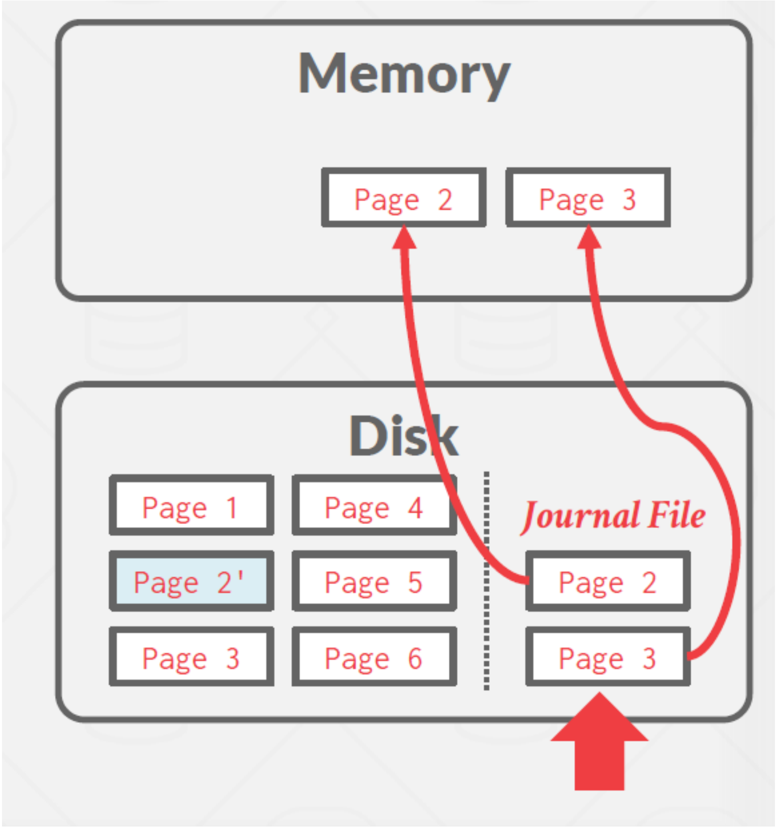
- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

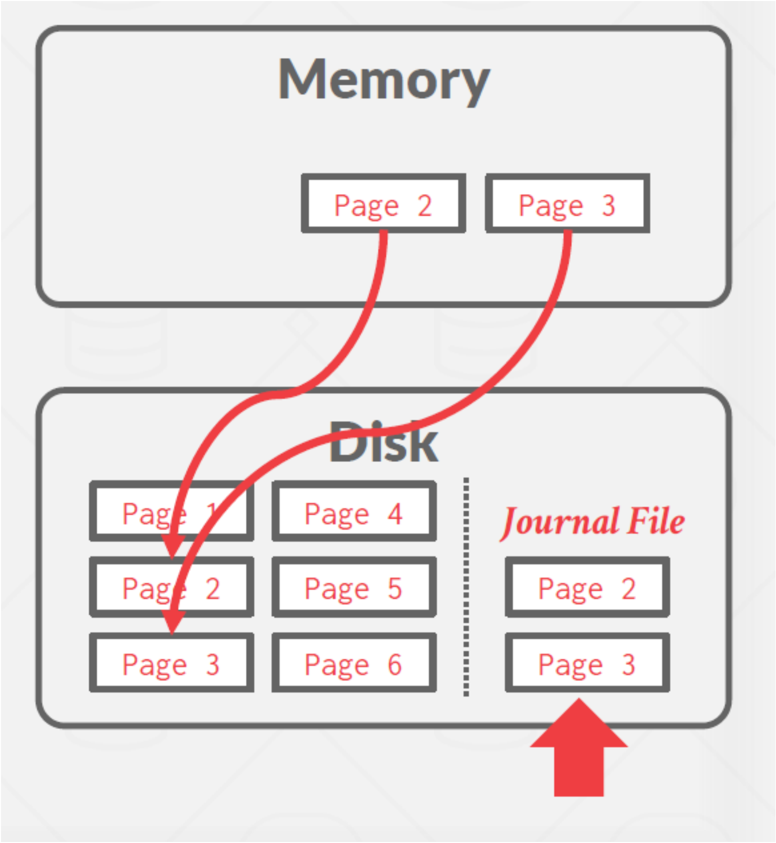
- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.
- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



Example: SQLITE (Pre-2010)

- When a txnmodifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

- After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



A Mechanisms for ensuring atomicity

- Another approach: LOGGING
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions
- Logging used by modern systems, because of the need for audit trail and for efficiency

Write-ahead Logging (WAL)

- Maintain a log file separate from data files that contains the changes that txns make to database.

- DBMS must write to disk the log file records that correspond to changes made to a database object before it can flush that object to disk.

WAL Protocol

- The DBMS stages all a txn's log records in DRAM (usually in a buffer pool)
- All log records pertaining to an updated page are written to non-volatile storage (SSD/HDD) before the page itself is over-written in non-volatile storage.
- A txn is not considered committed until all its log records have been written to non-volatile storage.

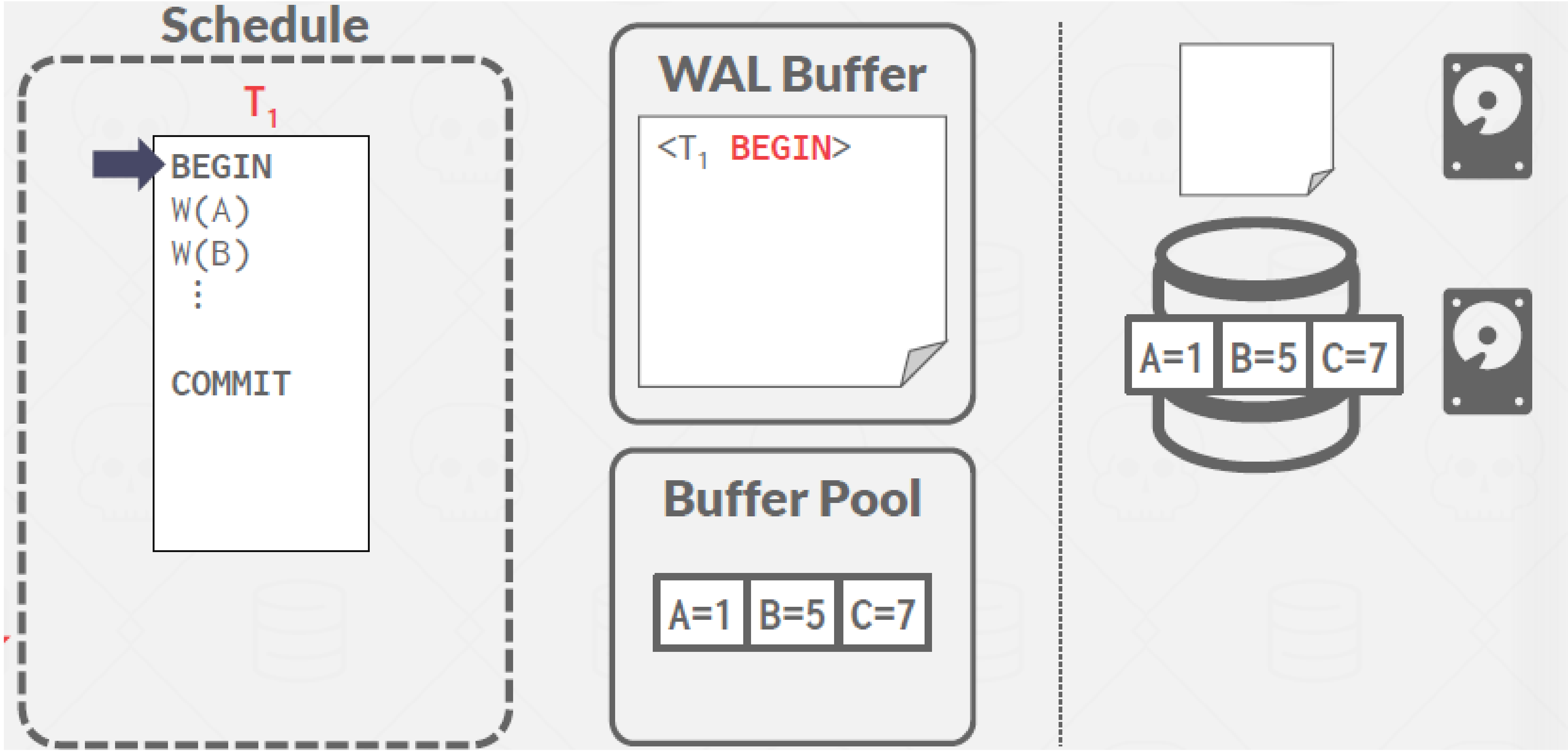
WAL Protocol

- Write a **<BEGIN>** record to the log for each txn to mark its starting point.

- When a txnf inishes, the DBMS will:
  - Write a **<COMMIT>**record on the log
  - Make sure that all log records are flushed before it returns an acknowledgement to application.

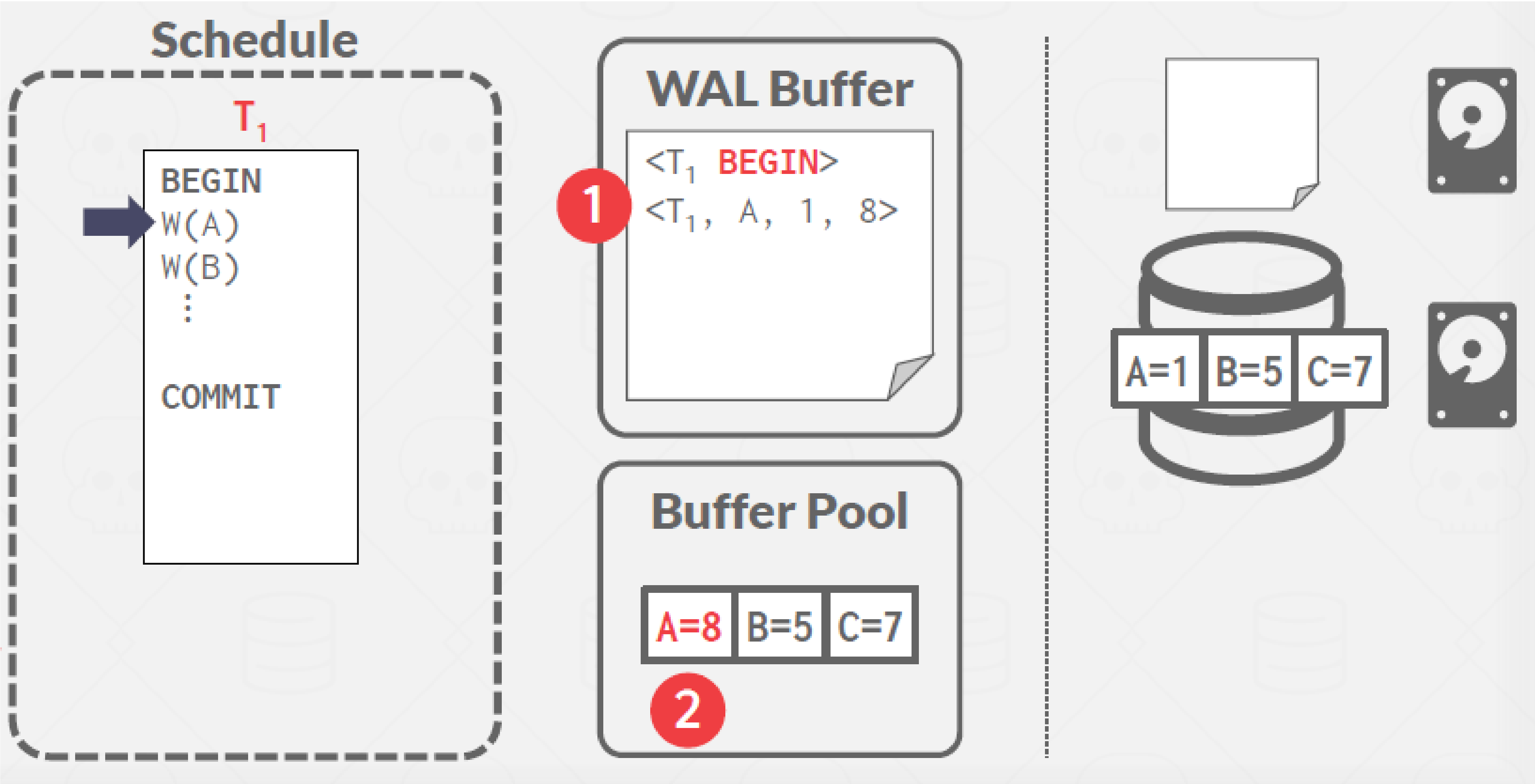
- Each log entry contains information about the change to a single object:
  - Transaction Id
  - Object Id
  - Before Value (**UNDO**)
  - After Value (**REDO**)

WAL Example



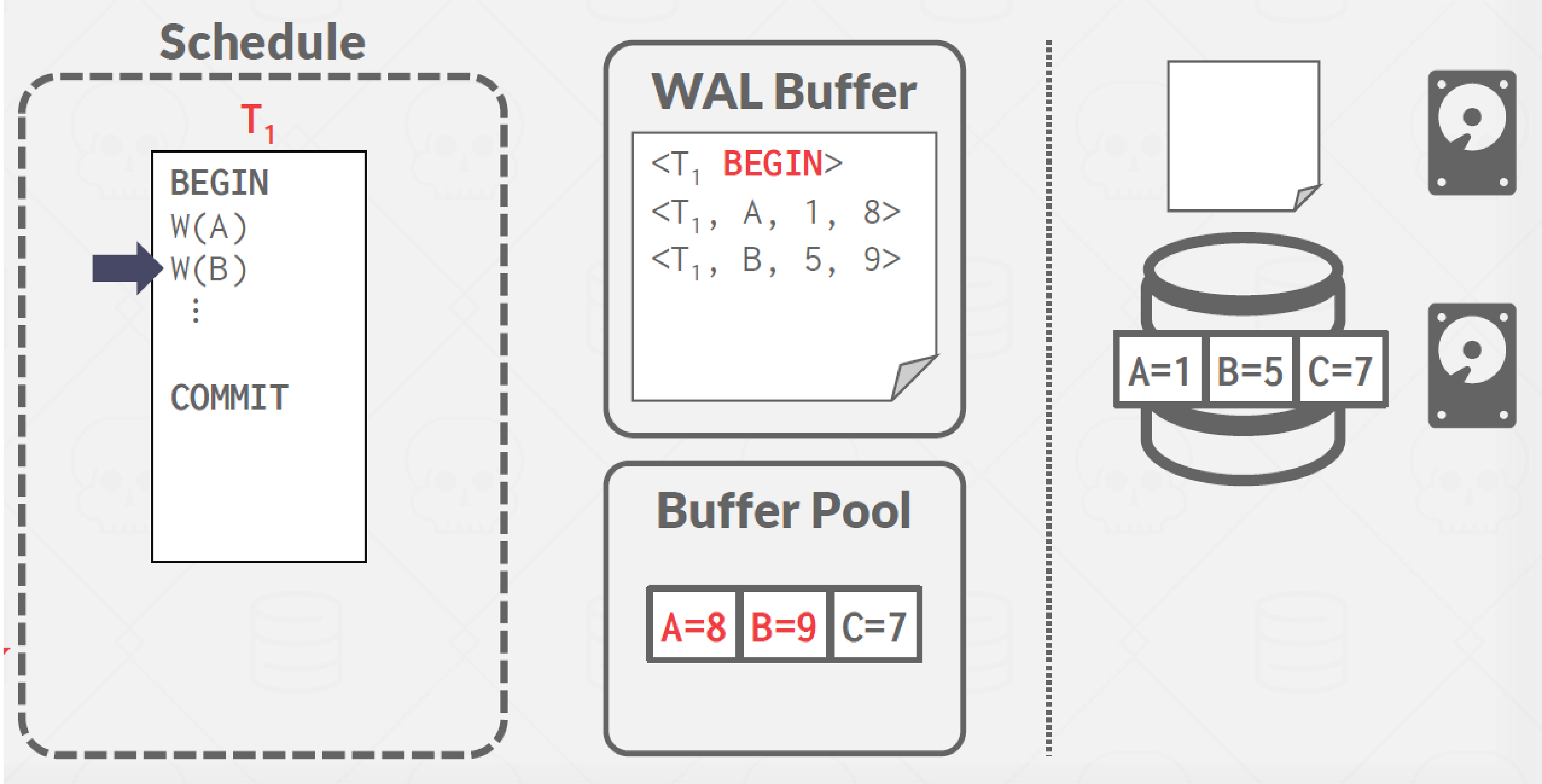
Schedule
T1
BEGIN
W(A)
W(B)
COMMIT
WAL Buffer
<T1 BEGIN>
Buffer Pool
A=1
B=5
C=7
A=1
B=5
C=7

WAL Example



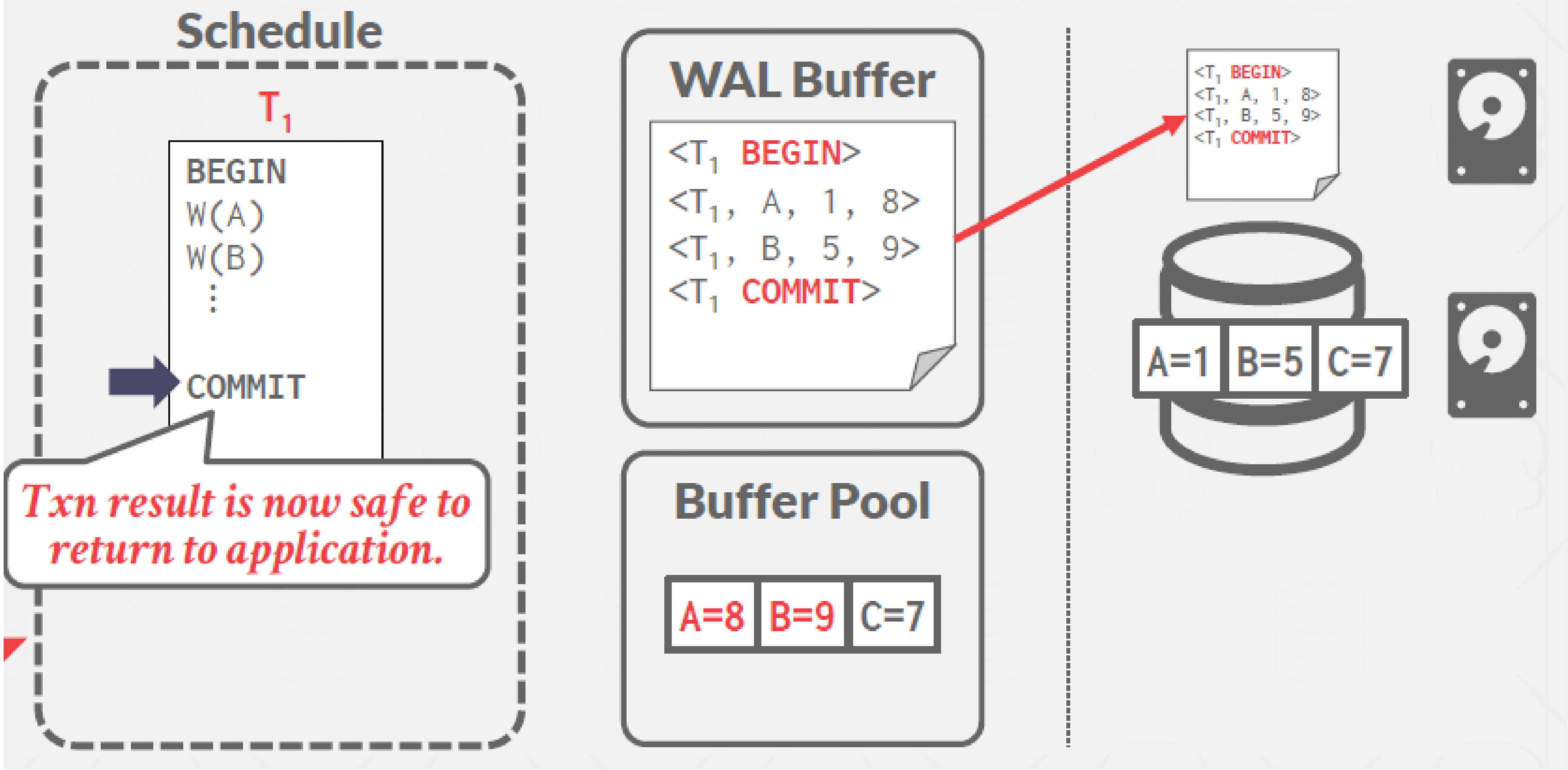
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
1
<T1 BEGIN>
<T1, A, 1, 8>
Buffer Pool
A=8
B=5
C=7
2
A=1
B=5
C=7

WAL Example



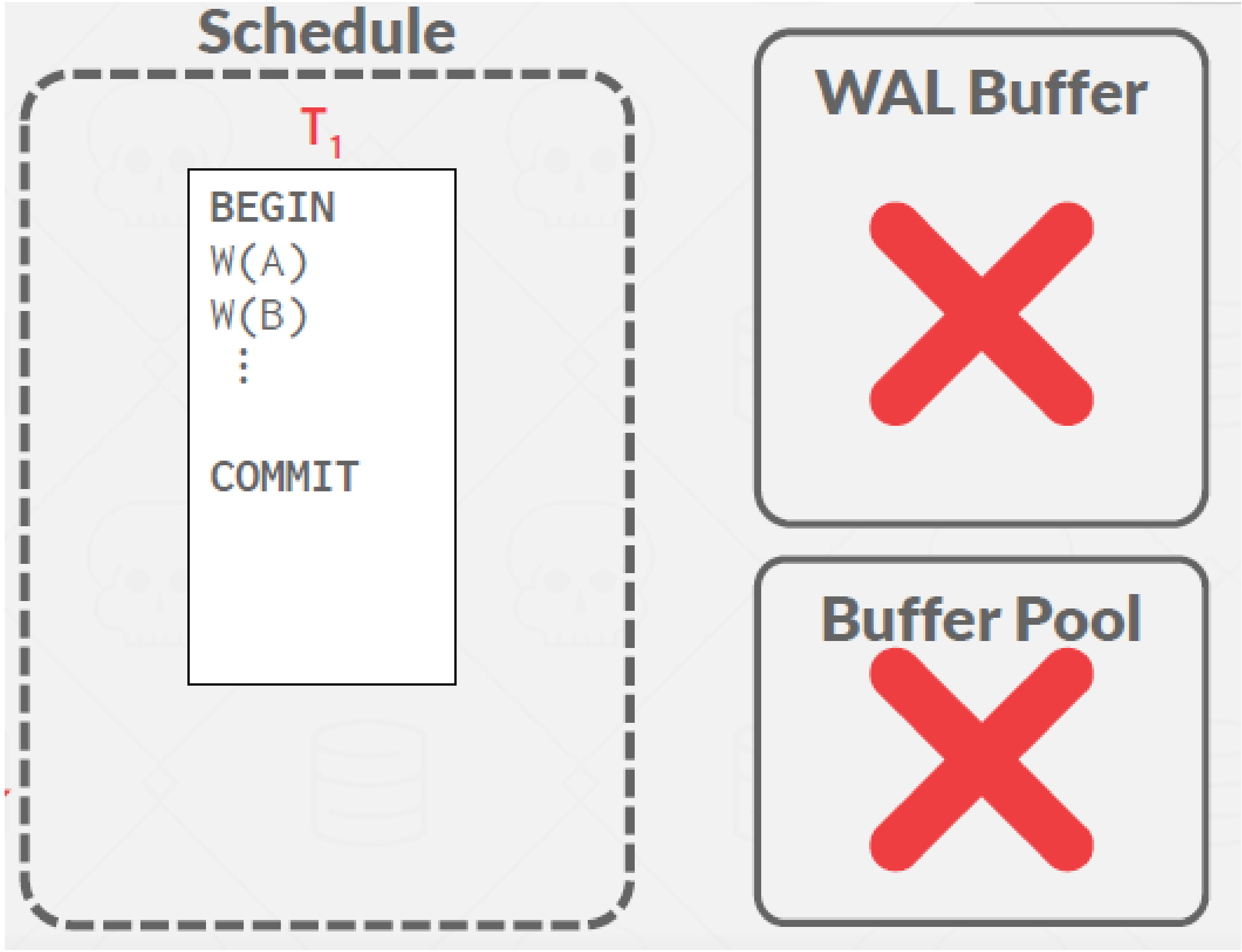
Schedule
T1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
Buffer Pool
A=8
B=9
C=7
A=1
B=5
C=7

WAL Example

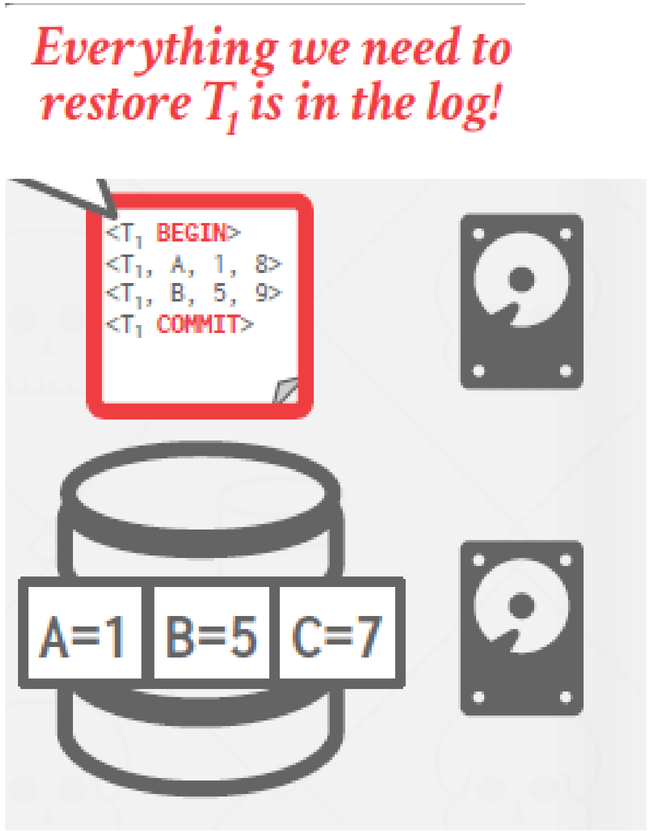


Schedule
T1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to application.
WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
Buffer Pool
A=8
B=9
C=7
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A=1
B=5
C=7

WAL Example



Schedule
T_1
BEGIN
W(A)
W(B)
COMMIT
WAL Buffer
Buffer Pool



Everything we need to restore T_1 is in the log!
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1 COMMIT>
A=1
B=5
C=7

D Durability - Recovering from a crash

- The famous IBM DB2 ARIES recovery protocol from 90s
- Three phases
  - ***Analysis*****:** Scan the log (forward from the most recent *checkpoint*) to identify all transactions that were active at the time of the crash
  - ***Redo*****:** Redo updates as needed to ensure that all logged updates are in fact carried out and written to disk
  - ***Undo*****:** Undo writes of all transactions that were active at the crash, working backwards in the log
- At the end – all committed updates and only those updates are reflected in the database
- Some care must be taken to handle the case of a crash occurring during the recovery process!

Checkpoints

- The WAL will grow forever.
- After a crash, the DBMS must replay the entire log, which will take a long time.
- The DBMS periodically takes a checkpoint where it flushes all buffers out to disk.
  - This provides a hint on how far back it needs to replay the WAL afteracrash.

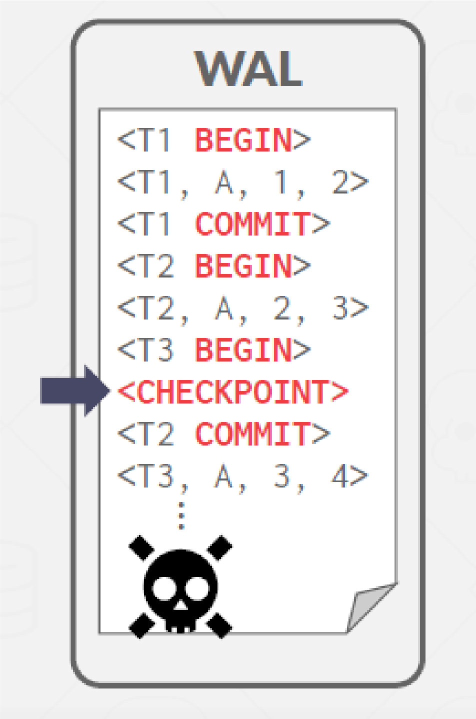
Checkpoints

- **Blocking / Consistent Checkpoint Protocol:**
  - Pause all queries.
  - Flush all WAL records in memory to disk.
  - Flush all modified pages in the buffer pool to disk.
  - Write a **<CHECKPOINT>** entry to WAL and flush to disk.
  - Resume queries.

- Use the **<CHECKPOINT>**record as the starting point for analyzing the WAL.

Checkpoints

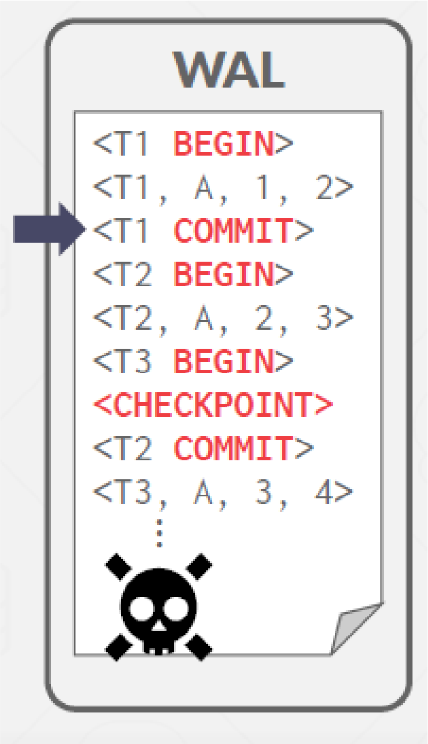
- Use the **<CHECKPOINT>**record as the starting point for analyzing the WAL.



Checkpoints

- Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.

- Any txn that committed before the checkpoint is ignored (T1).

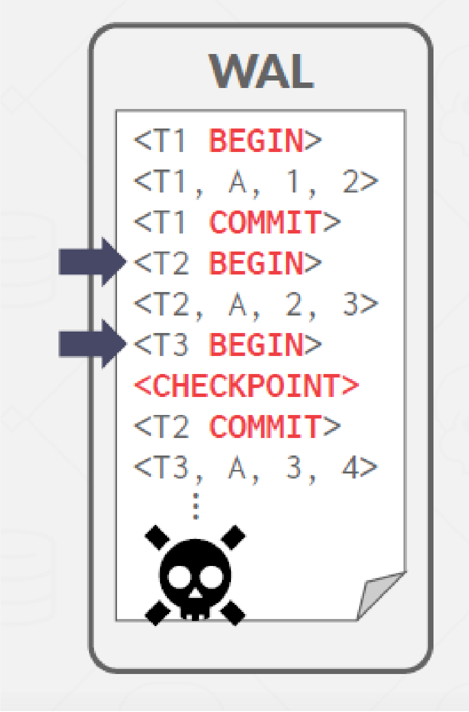


Checkpoints

- Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.

- Any txn that committed before the checkpoint is ignored (T1).

- T2, T3 did not commit before last checkpoint
  - Need to redo **T2** because it committed after checkpoint.
  - Need to undo **T3** because it did not commit before the crash.



Transaction consistency

- "Consistency" - data in DBMS is accurate in modeling real world and follows integrity constraints
- User must ensure that transaction is consistent
- Key point:

C Transaction consistency (cont.)

- Recall: Integrity constraints
  - must be true for DB to be considered consistent
  - **Examples:**
    1. FOREIGN KEY R.sid REFERENCES S
    2. ACCT-BAL >= 0

- System checks integrity constraints and if they fail, the transaction rolls back (i.e., is aborted)
  - Beyond this, DBMS does not understand the semantics of the data
  - e.g., it does not understand how interest on a bank account is computed

Isolation of transactions

- Users submit transactions concurrently

- Each transaction executes as if it was running by itself
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.

Example

- Consider two transactions:

```
T1:        BEGIN   A=A+100,   B=B-100   END
T2:        BEGIN   A=1.06*A,   B=1.06*B   END
```

A

B

Accid	balance
101	100
102	1000
104	1000

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 6% interest
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2?
  - $2000 *1.06 = $2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order

Example (contd.)

- Legal outcome: A=1166,B=954
- Consider a possible interleaved *schedule*:

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

- This is OK (same as T1;T2). But what about:

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

- **Result: A=1166, B=960; A+B = 2126, bank loses $6**

Anomalies with interleaved execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),		R(A), W(A), C
T2:		R(A), W(A), C	

- Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

How do we allow concurrency while preventing these anomalies?
(Theory of serializability)

Transactions & Schedules: Definitions

- A program may carry out many operations on the data retrieved from the database
- The DBMS is only concerned about what data is read/written from/to the database
- *Database*
  - a fixed set of named data objects *(A, B, C, …)*
- *Transaction*
  - a sequence of actions *(read(A), write(B),commit, abort …)*
- *Schedule*
  - an interleaving of actions from various transactions

Formal properties of schedules

- *Serial schedule:* Schedule that does not interleave the actions of different transactions

T_1:	$R_1(X)$	T_2:	$R_2(X)$
	X=X-20		X=X+10
	$W_1(X)$		$W_2(X)$

T_1, T_2

	T_1	T_2
t0:	$R_1(X)$	
t1:	X=X-20	
t2:	$W_1(X)$	
t3:		$R_2(X)$
t4:		X=X+10
t5:		$W_2(X)$

T_2, T_1

	T_1	T_2
t0:		$R_2(X)$
t1:		X=X+10
t2:		$W_2(X)$
t3:	$R_1(X)$	
t4:	X=X-20	
t5:	$W_1(X)$	

Formal properties of schedules

- *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions
  Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Conflicting operations

- We need a formal notion of equivalence that can be implemented efficiently
  - Base it on the notion of “conflicting” operations

- <u>Definition</u>: Two operations conflict if:
  - They are done by different transactions,
  - And they are done on the same object,
  - And at least one of them is a write

$R_1(A)$, $W_2(A)$

$W_1(A)$, $R_2(A)$

$W_1(A)$, $W_2(A)$

$R_1(B)$, $W_2(B)$

$W_1(B)$, $R_2(B)$

$W_1(B)$, $W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

Conflict serializable schedules

- Definition: Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions,
  - And every pair of conflicting actions is ordered the same way

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

$S_1 \equiv S_2$

$S_1 \equiv S_3$??

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_2:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
$R_1(B)$	
	$W_2(A)$
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_3:

T_1	T_2
$R_1(A)$	
	$R_2(A)$
$W_1(A)$	
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	
$W_1(B)$	

Conflict serializable schedules

- <u>Definition</u>: Schedule S is conflict serializable if:
    - S is conflict equivalent to some serial schedule

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_2:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$

S_3:

T_1	T_2
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	

Conflict serializability: Definition

- A schedule S is conflict serializable if:
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- *Example:*

R(A)	W(A)			R(B)	W(B)			
		R(A)	W(A)			R(B)	W(B)	

$\equiv$

R(A)	W(A)	R(B)	W(B)				
				R(A)	W(A)	R(B)	W(B)

Conflict serializability (cont.)

- Here's another example:

R(A) W(A)

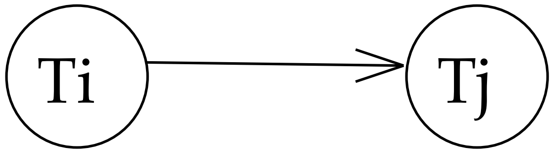
R(A) W(A)

- Conflict serializable or not?

NOT!

Testing for conflict serializability

- *Precedence graph*:
    - One node per transaction
    - Edge from Ti to Tj if:
        - An operation Oi of Ti conflicts with an operation Oj of Tj and
        - Oi appears earlier in the schedule than Oj



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

Precedence graph

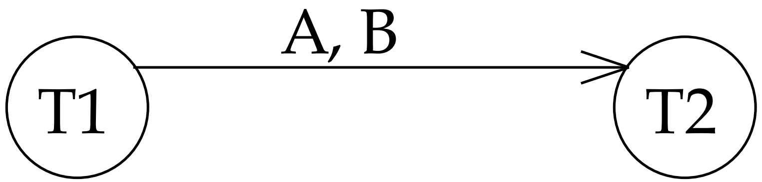
$R_1(A), W_2(A)$
$W_1(A), R_2(A)$
$W_1(A), W_2(A)$
$R_1(B), W_2(B)$
$W_1(B), R_2(B)$
$W_1(B), W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$



Precedence graph

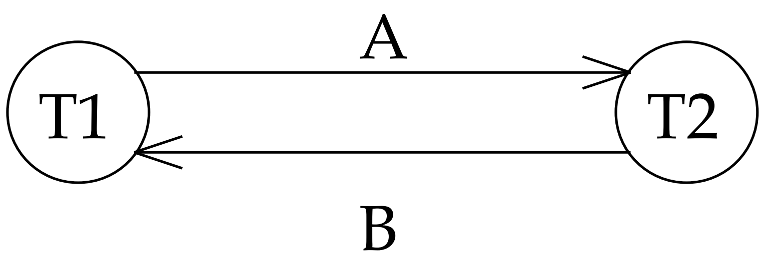
$R_1(A), W_2(A)$
$W_1(A), R_2(A)$
$W_1(A), W_2(A)$
$R_1(B), W_2(B)$
$W_1(B), R_2(B)$
$W_1(B), W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	
$W_1(B)$	



Not conflict serializable

The cycle in the graph reveals the problem.
The output of T1 depends on T2, and vice-versa

Two-Phase Locking (2PL)

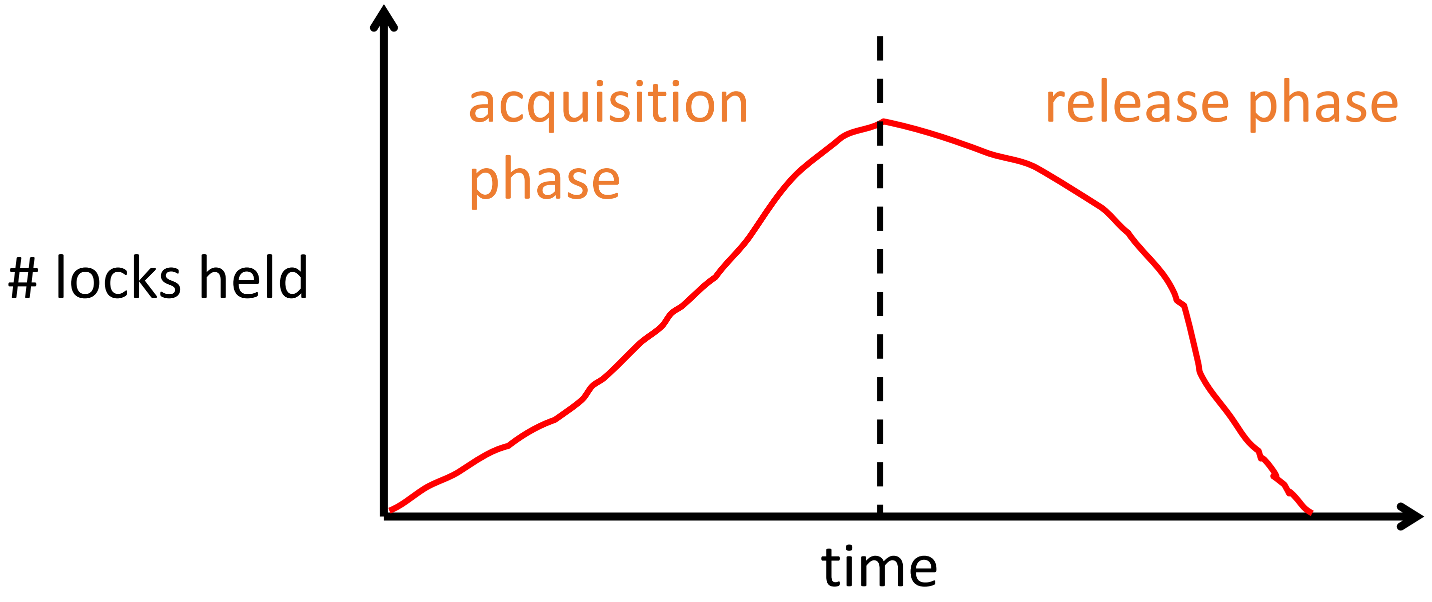
- Locking protocol
  - Each transaction must obtain an S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing
  - ***A transaction cannot request additional locks once it releases any locks***
  - Thus, there is a "growing phase" followed by a "shrinking phase"

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

2PL & Serializability

- 2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to **Cascading Aborts**



Strict 2PL

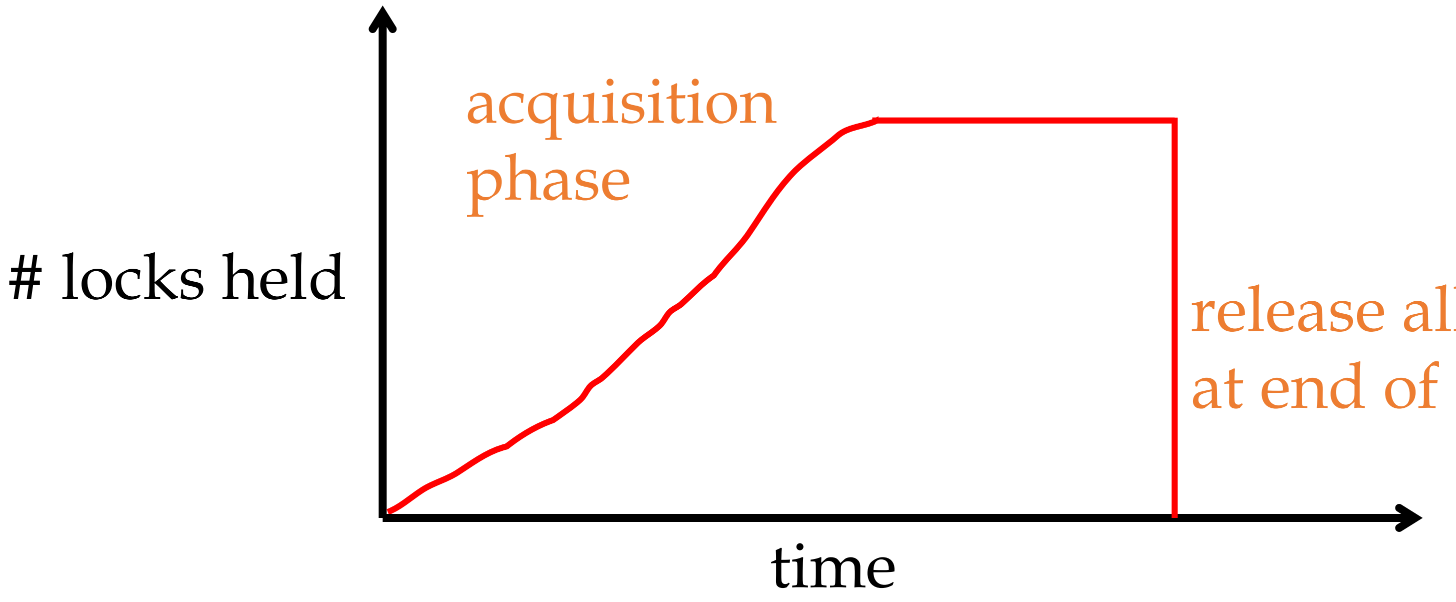
- Problem: Cascading Aborts
- Example: rollback of T1 requires rollback of T2!

T1:	$R_1(A), W_1(A), R_1(B), W_1(B),$		Abort
T2:		$R_2(A), W_2(A)$	

- To avoid Cascading Aborts, use Strict 2PL
- Strict Two-Phase Locking (Strict 2PL) Protocol:
  - Same as 2PL, except: All locks held by a transaction are released only when the transaction completes

Strict 2PL (cont.)

- Allows only conflict serializable schedules
- In effect, “shrinking phase” is delayed until
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (locks can be released after rollback)



Non-2PL, A= 100, B=200, output =?

Lock_X(A)		
Read(A)	Lock_S(A)	
A: = A-50		
Write(A)		A=50
Unlock(A)		
	Read(A)	
	Unlock(A)	
	Lock_S(B)	
Lock_X(B)		
	Read(B)	
	Unlock(B)	
	PRINT(A+B)	**250**
Read(B)		
B := B +50		
Write(B)		B=250
Unlock(B)		

2PL, A= 100, B=200, output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

→ A=50

→ B=250

→ **300**

Strict 2PL, A= 100, B=200, output =?

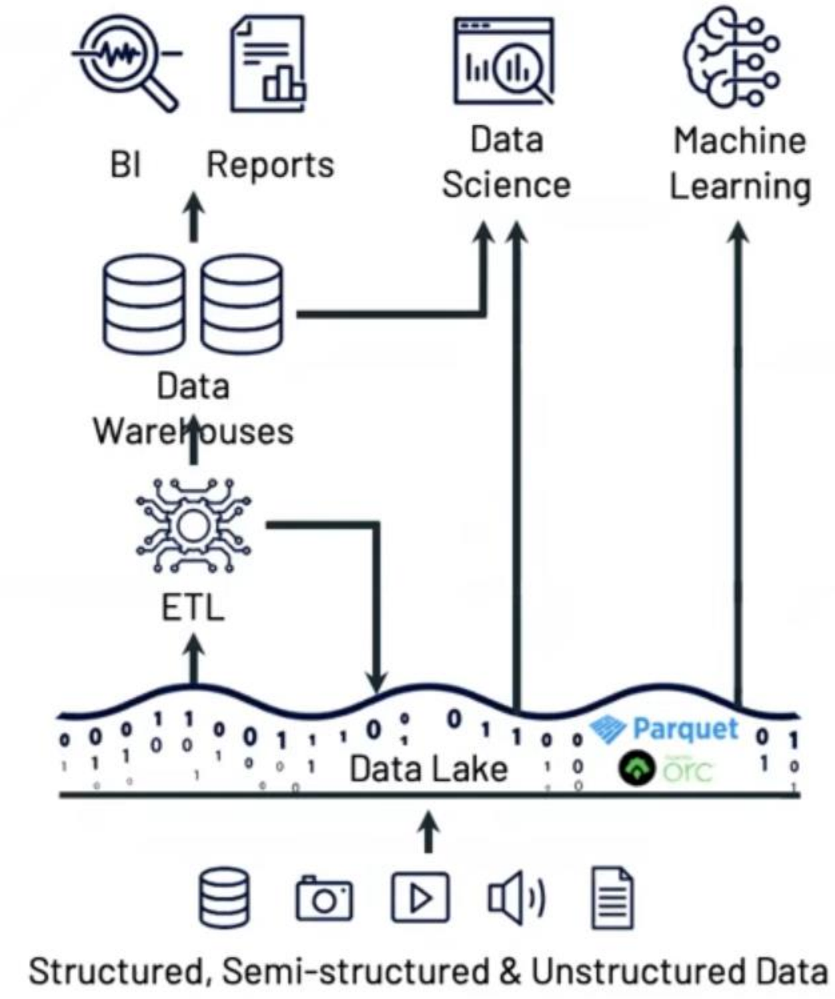
Lock_X(A)		
Read(A)	Lock_S(A)	
A: = A-50		
Write(A)		→ A=50
Lock_X(B)		
Read(B)		
B := B +50		→ B=250
Write(B)		
Unlock(A)		
Unlock(B)		
	Read(A)	
	Lock_S(B)	
	Read(B)	
	PRINT(A+B)	→ **300**
	Unlock(A)	
	Unlock(B)	

2PL: Summary

- Locks implement the notions of conflict directly
- 2PL has:
  - Growing phase where locks are acquired and no lock is released
  - Shrinking phase where locks are released and no lock is acquired
- Strict 2PL requires all locks to be released at once, when transaction ends

Problems with Traditional Data Lake

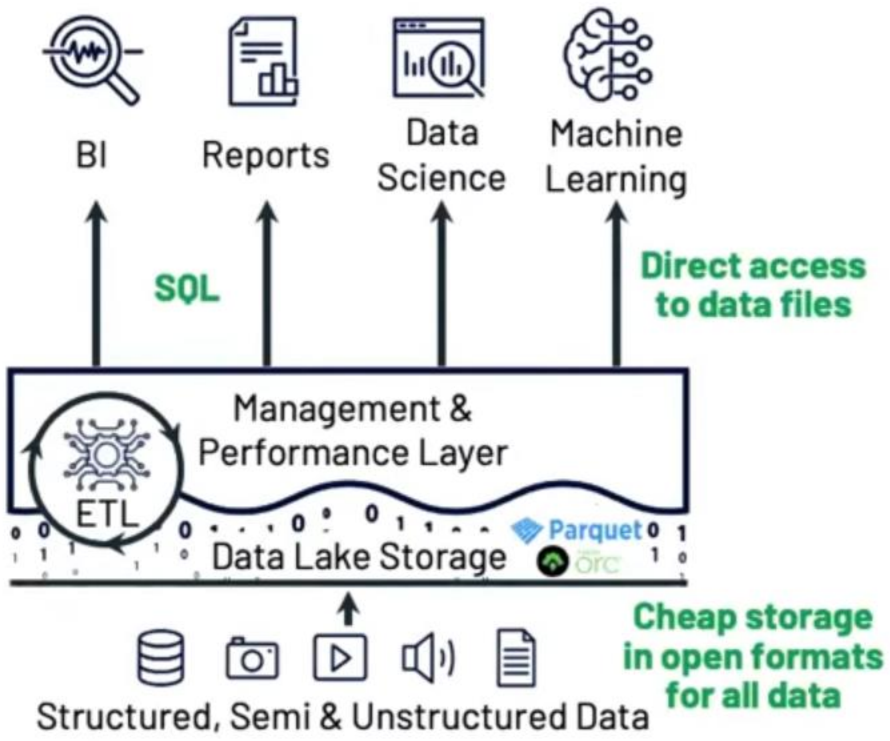
- Data reliability issues
  - Lack of transactional update support
- Also High cost
  - Duplicated storage, continuous ETL cost
- Also Timeliness issues
  - Still dependent on ETL for SQL analytics



From Data Lake to Lakehouse

- Implement data warehouse management and performance features on top of **directly-accessible data** stored using **open formats** in **data lake storage**
  - Traditional database stuff (transaction, index, security all in mgmt/perf layer)
  - SQL interface + direct file access

- Breaks physical data independence to a certain extent



Key Technologies

- **Metadata layers on data lakes**
  - **Adds transactions, versioning, indexing, etc**
  - **We will take a quick look at the DeltaLake project**

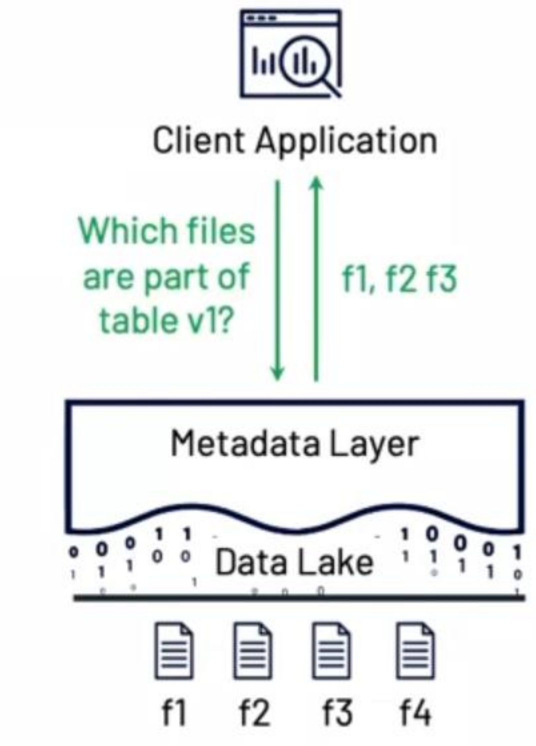
- Lakehouse engine design
  - Performant SQL on data lake storage

Metadata layer

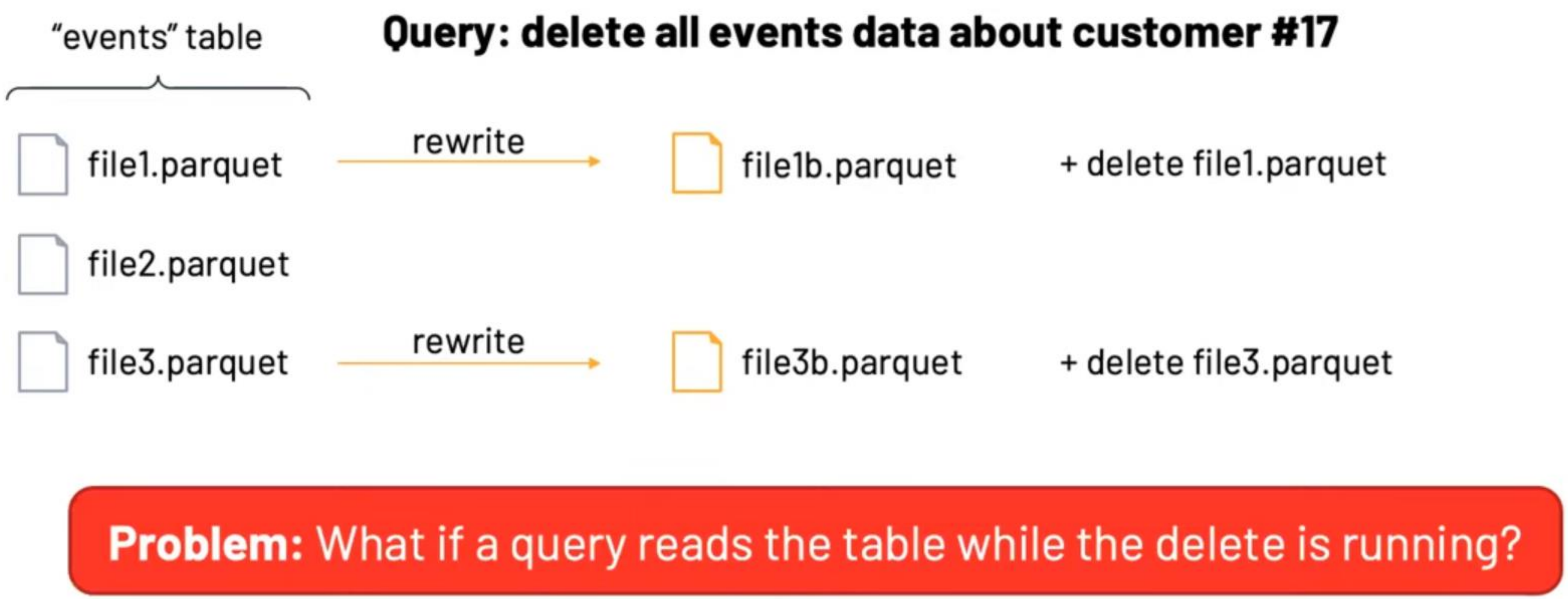
- Data lake stores files in whatever format
  - Can only get/put files, list files typically

- Build layer sits in front to track which files are part of table version to offer rich management features

- Ex: Deltalake, Hive ACID, Netflix ICEBERG



Example: Traditional Data Lake

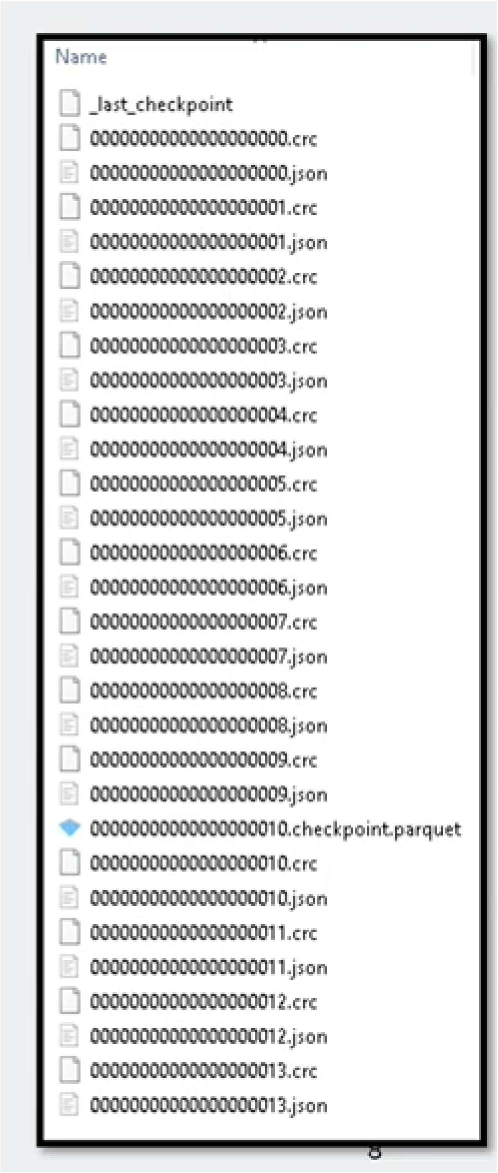


Delta Lake & Delta Log

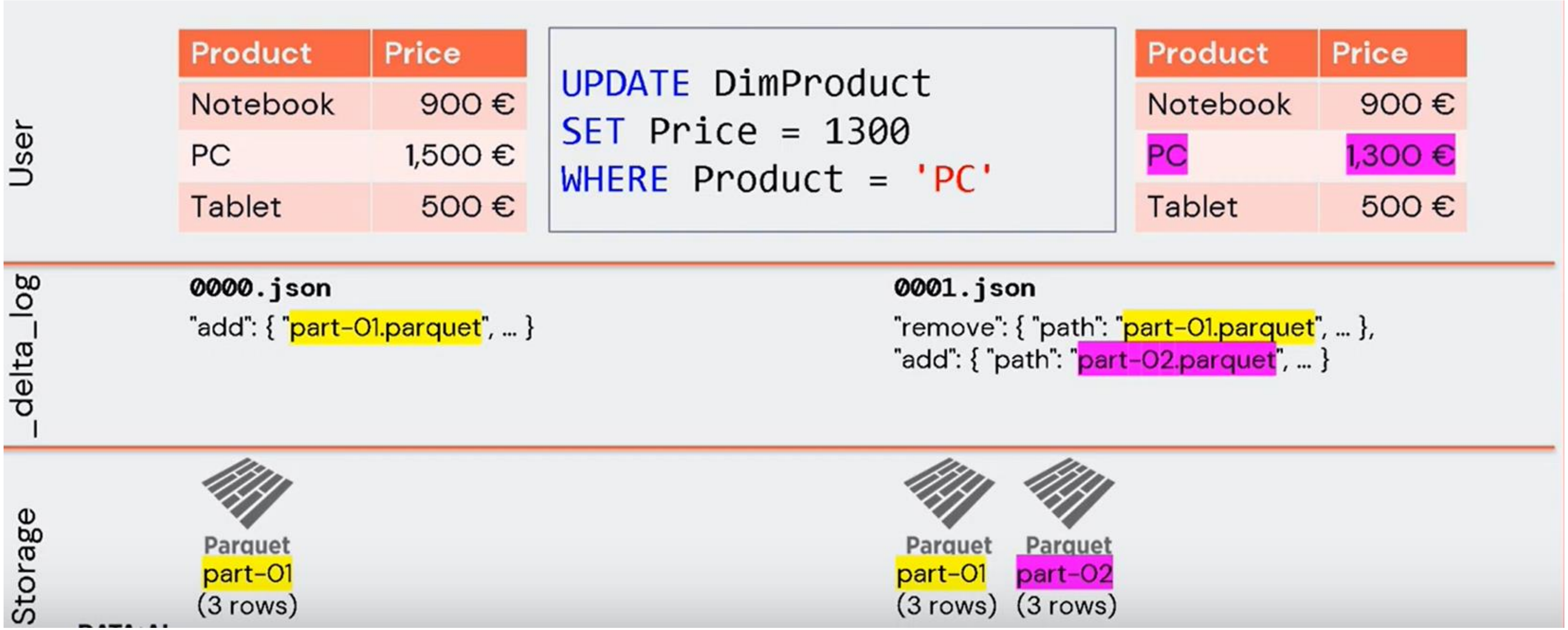
- Delta lake uses a delta log to track versions
- Delta log contains
  - Table schema + changes
  - References to files
  - Metadata and metrics about txns

- Transactions in delta log stored as JSON files
  - After 10 txns, checkpoint file is generataed
  - Checkpoint aggregates all previous txns in 1 parquet file

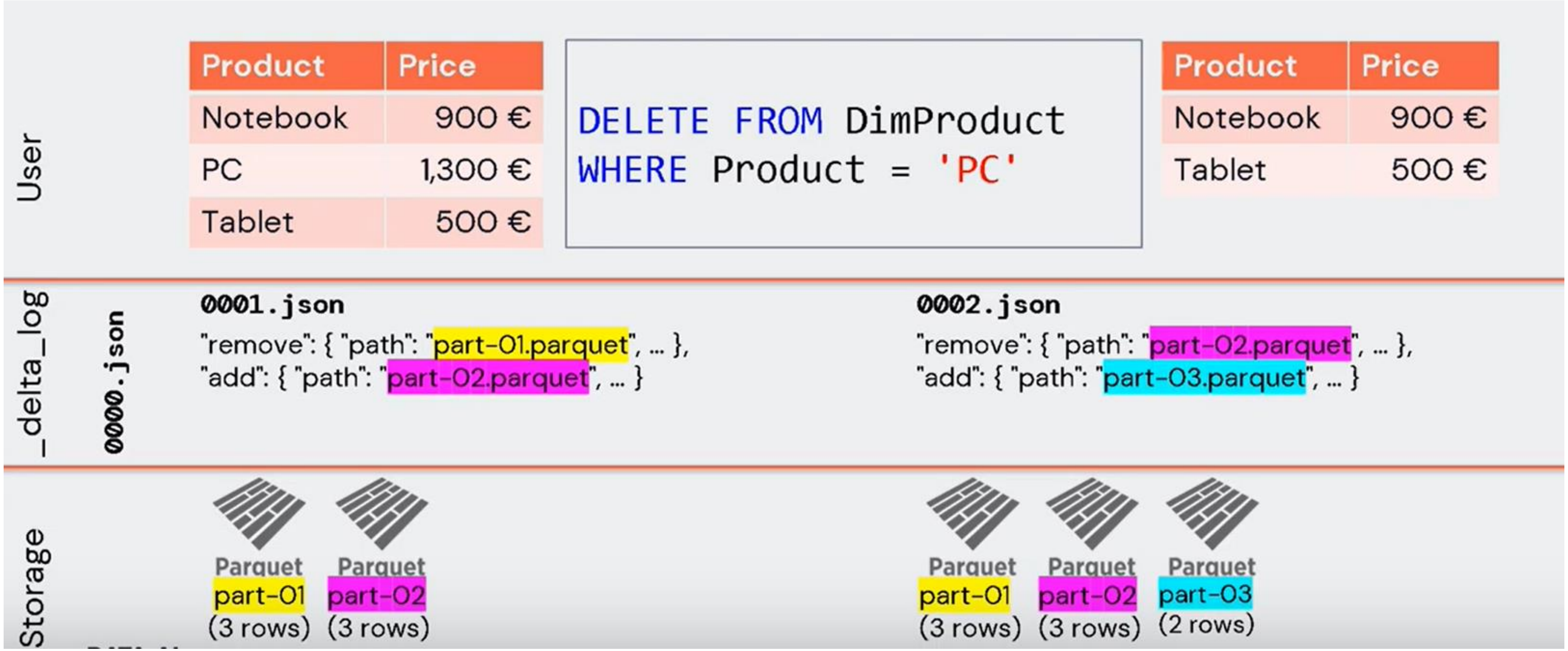
- Delta log allows optimistic concurrency control



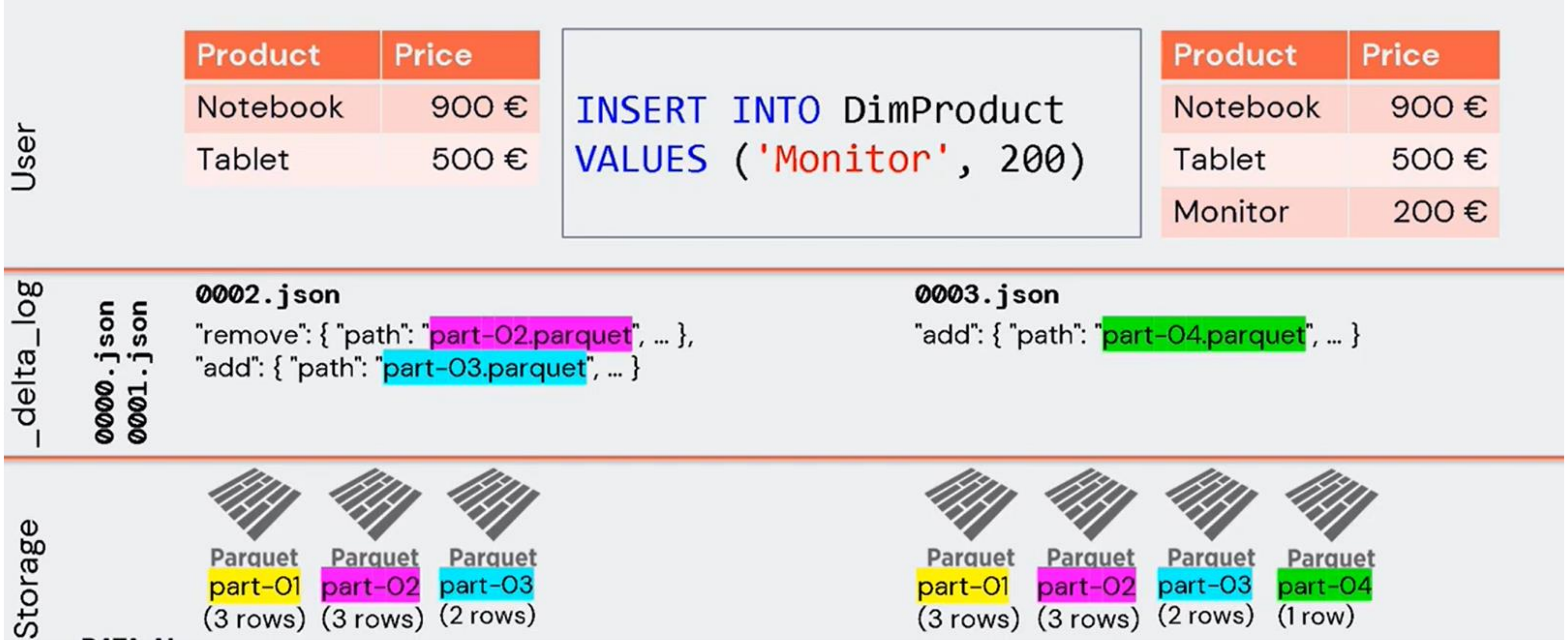
Delta Lake: Example DML UPDATE



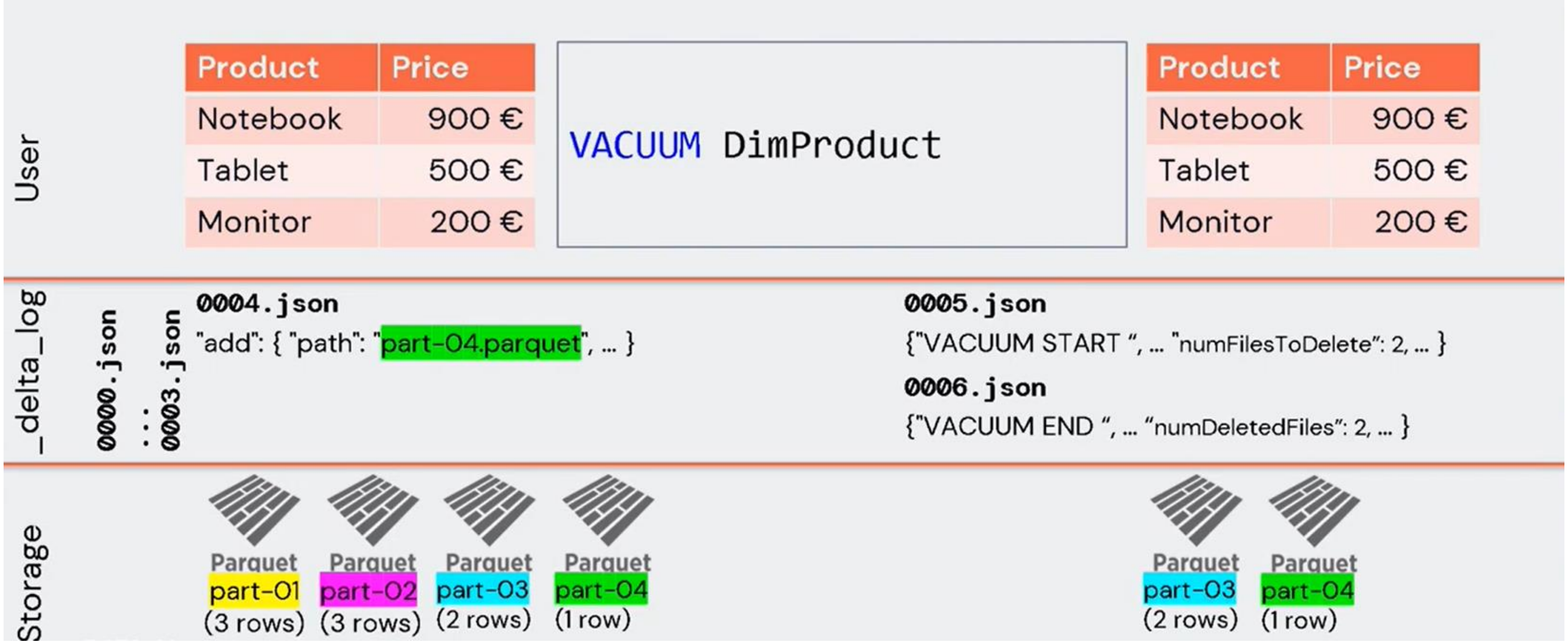
Delta Lake: Example DML DELETE



Delta Lake: Example DML INSERT



Delta Lake: Storage Mgmt. with VACUUM



Key Technologies

- Metadata layers on data lakes
  - Adds transactions, versioning, indexing, etc
  - Ex: DeltaLake project

- **Lakehouse engine design**
  - **Performant SQL on data lake storage**

- Declarative I/O interface for ML
  - See https://www.youtube.com/watch?v=LJb1tR3i9hU

Lakehouse Engine Design

Even with a fixed, directly-accessible storage format, 4 optimizations help:

- Auxiliary data structures like statistics and indexes
- Data layout optimizations within files
- Caching hot data in a fast format
- Execution optimizations like vectorization

Minimize I/Os for cold data

Match DW performance on hot data

- Apache Photon: A lakehouse query execution engine
  - Highly vectorized query execution

More details on DeltaLake site: https://delta.io/

Summary

- Database engines pioneered ACID semantics
- Theory of serializability gives us a definition of correctness
- Optimistic and pessimistic concurrency control protocols provide isolation
- DeltaLake brings many of these advantages to data lakes