

6.5830 / 6.5831

Quiz 2 Review

November 15, 2023

Logistics

- **What:** Covers lectures 10 to 18 (inclusive)
 - **When:** Wednesday during lecture (November 20, 2023 at 2:30 pm)
 - **Where:** Lecture classroom (45-230)
 - **Length:** 80 minutes
 - **How:** Quiz will be on paper
-
- Open book/notes/calculator. No electronic devices with internet access.
 - Email staff for conflicts and accommodations (6.5830-staff@mit.edu) or make a private Piazza post by next Monday 5pm ET at the latest!

Topics

- Database layout for analytic databases
- Transactions and locking
- Logging and recovery (ARIES)
- Optimistic concurrency control and snapshot isolation
- Parallel/distributed databases (analytics and transactions)
- Cardinality estimation
- Eventual Consistency

Column Store

Linearizing a Table – Column Store

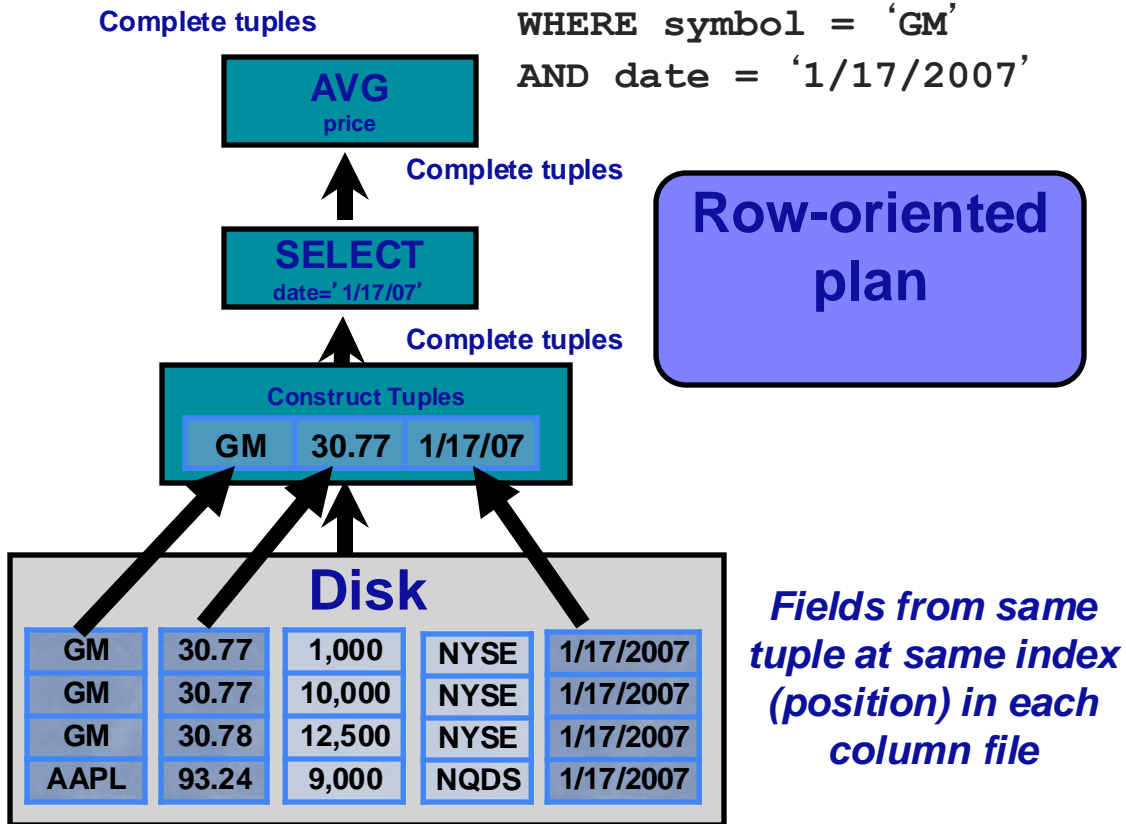
C1	C2	C3	C4	C5	C6

- Memory/Disk
(Linear Array)
- R1 C1
 - R2 C1
 - R3 C1
 - R4 C1
 - R5 C1
 - R6 C1
 - R1 C2
 - R2 C2
 - R3 C2
 - R4 C2
 - R5 C2
 - R6 C2
 - R1 C3
 - R2 C3
 - R3 C3
 - R4 C3
 - R5 C3
 - R6 C3
 - R1 C4
 - R2 C4
 - R3 C4
 - R4 C4
 - R5 C4
 - R6 C4

Query Processing Example

- Basic Column Store
- “Early Materialization”

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```



Query Processing Example

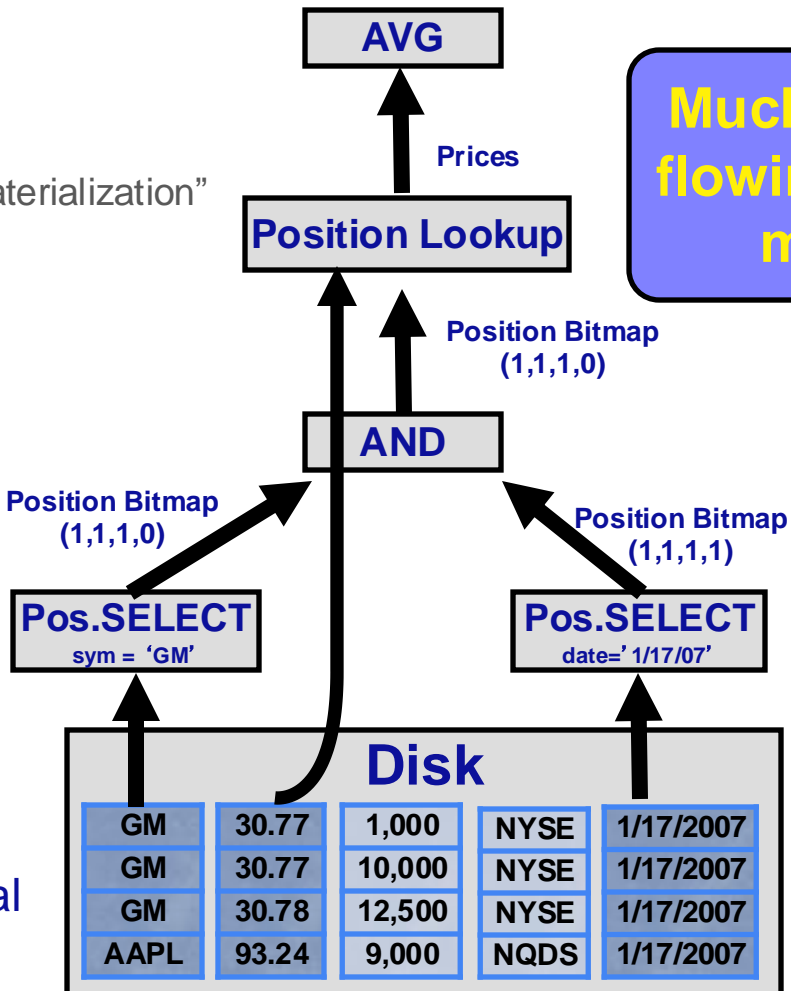
- C-Store
 - “Late Materialization”

Much less data flowing through memory

Column Compression

3xGM
1xAPPL

See Abadi et al
ICDE 07



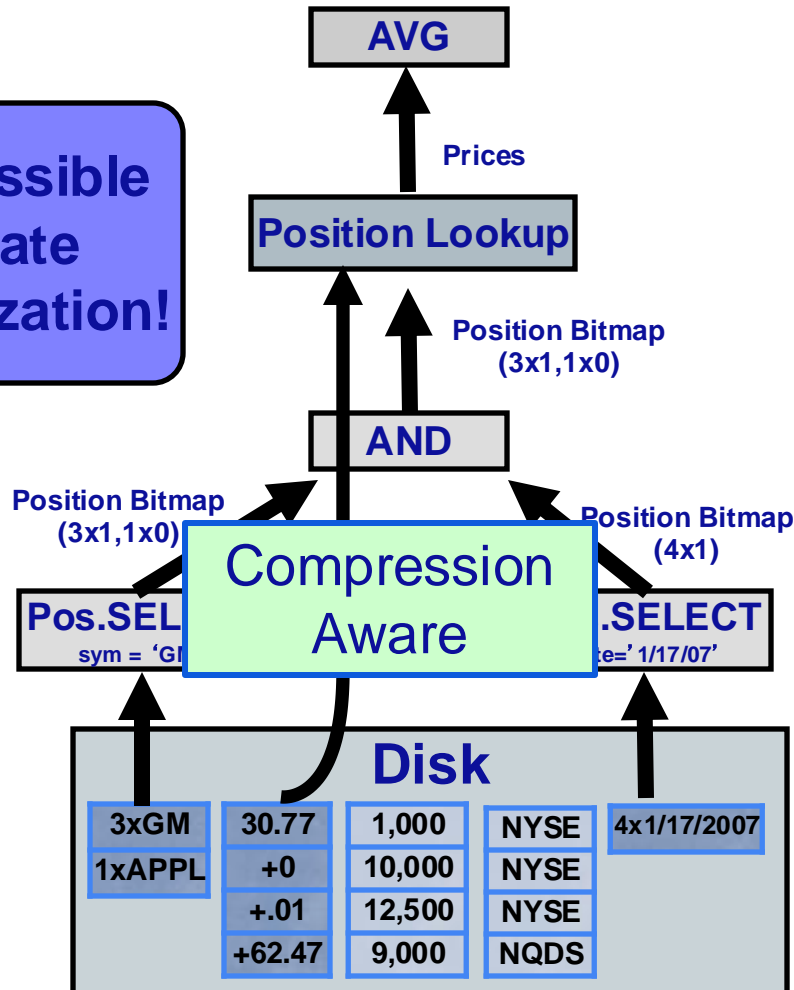
4 x 1/17/2007

Column Compression Schemes

Scheme Name	Brief Description
Run Length Encoding	Replace repeated values with a count and a value
Dictionary Encoding	A variant is using integers to represent string
Lempel Ziv Encoding	Builds data dictionary dynamically
Bit Packing	Encode values with fewest possible bits
Delta Encoding	Encode different with previous value and then bit pack
Bitmap Encoding	Encode few valued columns as bitmaps, then subsequently use e.g. RLE to further compress

Operating on Compressed Data


Only possible with late materialization!



Deployment optimization for column-oriented store

- Problem: complicated compression not friendly for heavy write workload
- Solution: disaggregate storage: write to write-optimized column store, read from WOS and read-optimized column store, async tuple mover to move data from WOS to ROS
- Tuple mover write new objects (why)
- ROS objects are periodically merged

Transactions

-  Groups a sequence of operations into an all-or-nothing unit
 - A powerful abstraction!
- Desirable properties (ACID)
 - Atomicity: All or nothing
 - Consistency: Maintains application-specific invariants
 - ➡ ○ Isolation: Transaction “appears” to run alone on the database
 - ➡ ○ Durability: Committed transactions’ writes persist even if the system crashes
- Transactions can be **aborted** by the user or DBMS

Serializability

- An ordering of actions in concurrent transactions that is serially equivalent

<u>T1</u>	<u>T2</u>	RA: Read A
RA		WA: Write A, may depend on anything read previously
	RA	
	WA	A/B are “objects” – e.g., records, disk pages, etc
WA		
RB		Assume arbitrary application logic between reads and writes
WB		
	RB	
	WB	

Not serially equivalent – T2’s write to A is lost, couldn’t occur in a serial schedule

In T1-T2, T2 should see T1’s write to A

In T2-T1, T1 should see T2’s write to A

View Serializability

A particular ordering of instructions in a schedule S is *view equivalent* to a serial ordering S' iff:

- Every value read in S is the same value that was read by the same read in S' .
- The final write of every object is done by the same transaction T in S and S'
- Less formally, all transactions in S “view” the same values they view in S' , and the final state after the transactions run is the same.

Conflict Serializability

A schedule is *conflict serializable* if it is possible to swap non-conflicting operations to derive a serial schedule.

Equivalently

🔑 For all pairs of conflicting operations {O1 in T1, O2 in T2} either

- O1 always precedes O2, or
- O2 always precedes O1.

Not conflict serializable:

<u>T1</u>	<u>T2</u>
RA	
	RA
	WA
WA	
RB	
WB	
	RB
	WB

Conflict Serializable => View Serializable
Conflict Serializable => Serializable

3 Ways to Test for Conflict Serializability

1. Check: For all pairs of conflicting operations $\{O_1 \text{ in } T_1, O_2 \text{ in } T_2\}$ either
 1. O_1 always precedes O_2 , or
 2. O_2 always precedes O_1 .
2. Swap non-conflicting operations to get serial schedule
3. Build precedence graph, check for cycles

Two Phase Locking (2PL) Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- 🔑 • Release locks only after last lock has been acquired, and ops on that object are finished

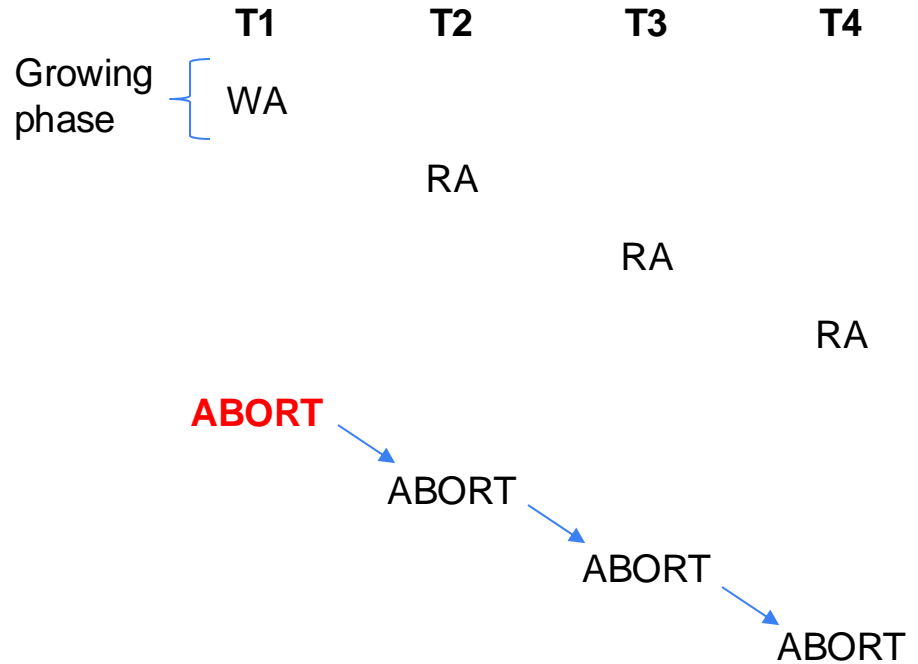
T2 \ T1	S	X
S	✓	✗
X	✗	✗

Lock Compatibility Table

Prevents sneaky updates!

Problem: Cascading aborts

- If T1 aborts, T2, T3 and T4 also need to abort
- **Solution:** Just keep write locks until the end!



Strict Two-Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release *shared* locks only after last lock has been acquired, and ops on that object are finished
- 🔑 • Release *exclusive* locks only after the transaction commits
- Ensures cascadeless-ness

Rigorous Two-Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- 🔑 • Release locks only after the transaction commits
 - Ensures cascadeless-ness, and
 - *Commit order = serialization order*

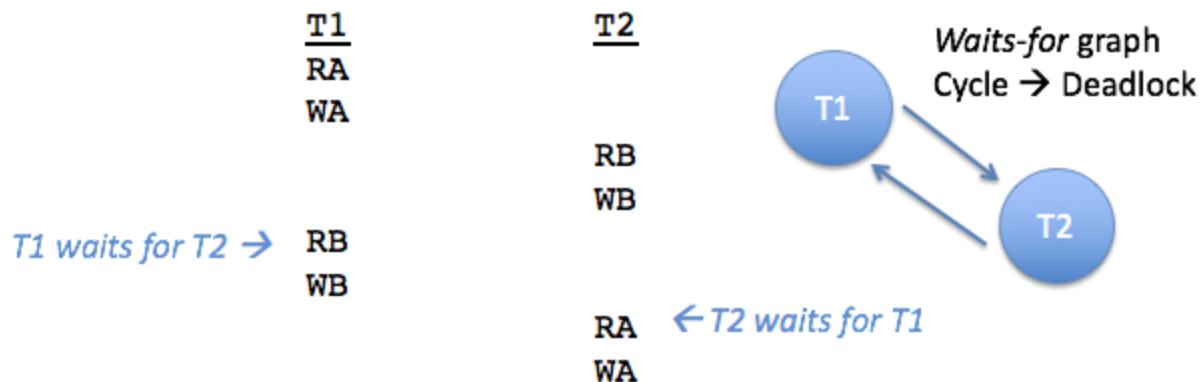
Example

- Permitted under strict 2PL but not rigorous 2PL:

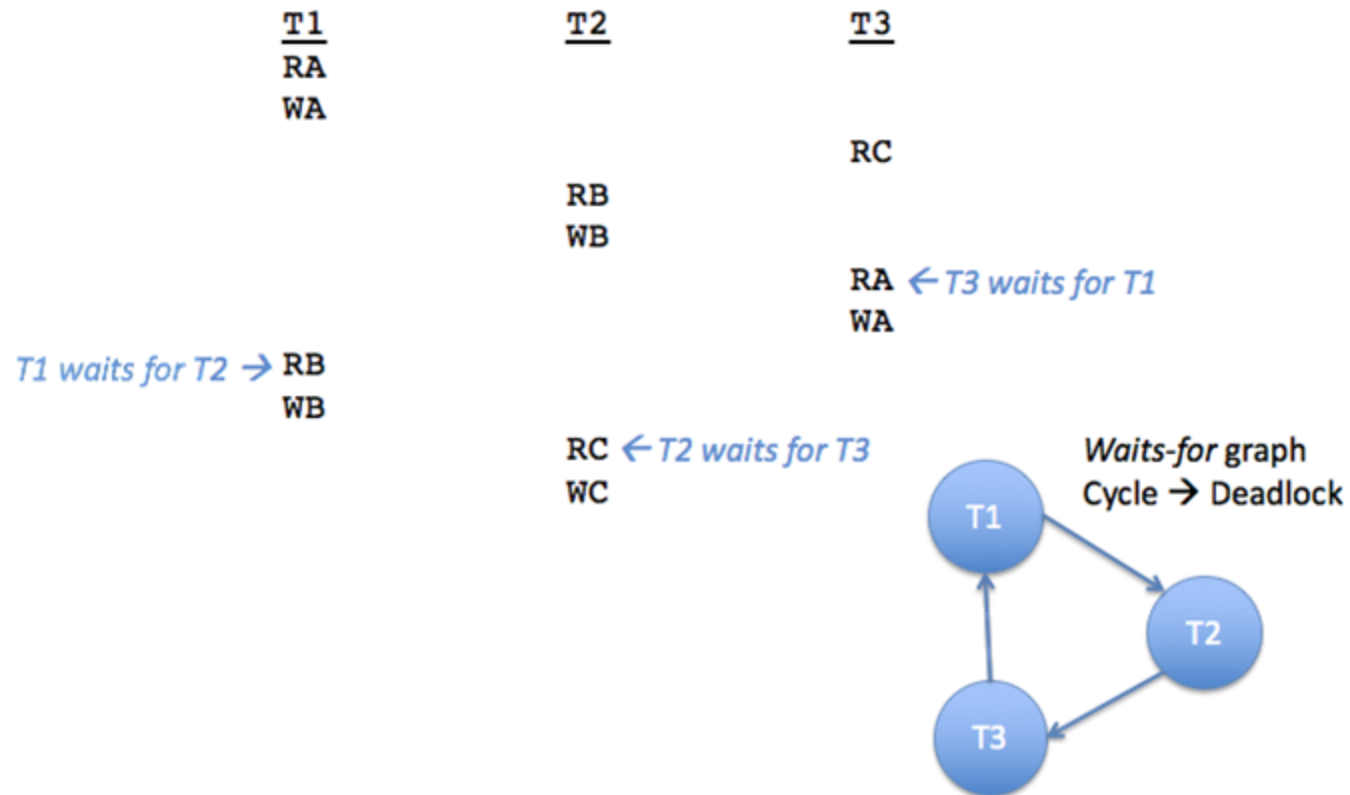
T1:	T2:
RA	
	WA
	COMMIT
COMMIT	

Deadlocks

- Possible for T_i to hold a lock T_j needs, and vice versa



Complex Deadlocks Are Possible



Resolving Deadlock

- Solution: abort one of the transactions
 - Recall: users can abort too

T1
RA
WA

T2

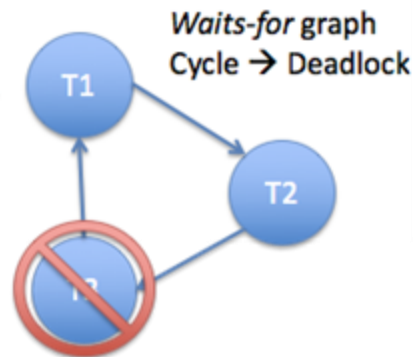
RB
WB

T1 waits for T2 → RB
WB

RC
WC

~~*T2 waits for T3*~~

Equivalent to T2 - T1



Final Wrinkle: Phantoms



- T1 scans a range; T2 later inserts into that range
- If T1 scans the range again, it will see a new value

```
T1
BEGIN
  SELECT * FROM emp WHERE SAL > 100
  ...
  SELECT * FROM emp WHERE SAL > 200
END

T2
BEGIN
  INSERT INTO EMP VALUES(...,sal=225)
END
```

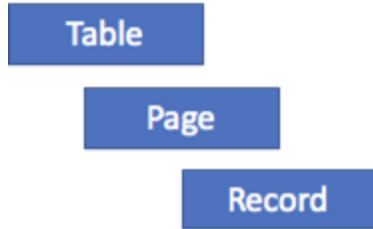
If we are just locking, e.g., records, this insertion would be allowed in all 2PL algos we have studied, but is not serializable (since this couldn't happen in a serial execution).

Preventing phantom reads

- Easy way: Acquire table locks
- 🔑 ● But if we have a clustered index we don't need to scan the whole table for the range query. We can do better using “gap locks” / next key locking.
- Example:
 - DB Entries: 10, 11, 13, 20
 - T1: Scan entries > 18 → Also lock “gaps”, i.e. lock gap 14 - 20 and 20 - ∞ .
 - T2: Insert record 19 → Also needs to acquire the gap lock that T1 holds.
 - T1: Scan entries > 18 → No phantom read since T2 is waiting on the gap lock.

See Lecture 11-12 for details

Locking Granularity / Intention Locks



- Suppose T1 wants to read record R1
- Needs to acquire *intention lock* on the Table and Page that T1 is in
- Intention lock marks higher levels with the fact that a transaction has a lock on a lower level
- Intention locks

- Can be read intention or write intention locks
- 🔑 Prevent transactions from writing or reading the whole object when another transaction is working on a lower level
- New *compatibility table*

		Reading / updating whole level		Reading / updating lower level	
T1 holds		S	X	IX	IS
T2 trying to acquire	S	Y	N	N	Y
	X	N	N	N	N
	IX	N	N	Y	Y
	IS	Y	N	Y	Y




Optimistic Concurrency Control (OCC)

- Alternative to locking for isolation
- Approach:
 - Store writes in a per-transaction buffer
 - Track read and write sets
 - At commit, check if transaction conflicted with earlier (concurrent) transactions
 - Abort transactions that conflict
 - Install writes at end of transaction
- 🔑 • “Optimistic” in that it does not block, hopes to “get lucky” arrive in serial interleaving

OCC Implementation

- Divide transaction execution in 3 phases
 - **Read**: transaction executes on DB, stores local state
 - **Validate**: transaction checks if it can commit
 - **Write**: transaction writes state to DB

What If Serializability Isn't Needed?

- E.g., application only needs to read committed data
- Databases provide different isolation levels
 - READ UNCOMMITTED →  Do not acquire read locks.
 - Ok to read other transaction's dirty data
 - READ COMMITTED →  "Short" read locks.
 - REPEATABLE READS →  No gap locks.
 - If R1 read $A=x$, R2 will read $A=x \forall A$
- Many database systems default to READ COMMITTED

Topics

- Transactions
- Logging and recovery (ARIES)
- Parallel/distributed databases (analytics and transactions)
- Systems “potpourri”
 - High-performance transactional systems (H-Store / Calvin / Aurora)
 - Eventual consistency (DynamoDB)
 - Cluster computing (Spark)
 - Cloud analytics (Snowflake)
- Cardinality estimation

Assumptions about crash

- Assume any data in memory is gone
- Data on disk is preserved
- → Recovery algorithms depend on when your system flushes pages
- Recovery ensures **atomicity** and **durability** in event of crash.

STEAL/NO FORCE \leftrightarrow UNDO/REDO

- If we STEAL pages, we will need to UNDO
- If we don't FORCE pages, we will need to REDO

	FORCE	NO FORCE
STEAL	UNDO	UNDO & REDO
NO STEAL	? UNDO	REDO

Steal: Can write dirty pages to disk before the txn commits.

Force: Force writes to disk on txn commit.



In GoDB, we do FORCE / NO STEAL and assume DB won't crash between FORCE and COMMIT

All commercial DBs do NO FORCE / STEAL for performance reasons

- If we FORCE pages, we will need to be able to UNDO if we crash between the FORCE and the COMMIT

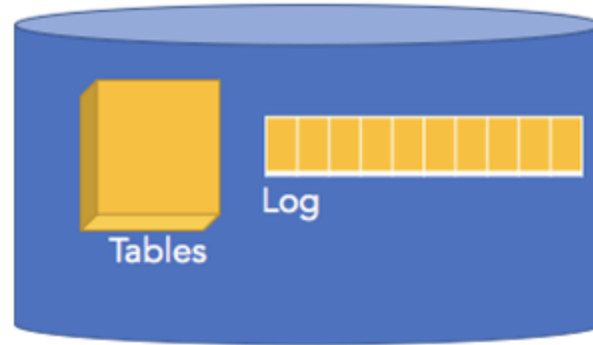
Database State During Query Execution



Memory

After crash, memory is gone!

Log records start and end of transactions, and contents of writes done to tables so we can solve both problems



Disk

🔑 Write-ahead log!



Problem 1: Some transactions may have written their uncommitted state to tables – need to **UNDO**

Problem 2: Some transactions may not have flushed all of their state to tables prior to commit – need to **REDO**

Types of Log Records

- **Start (SOT)** Log Sequence Number (LSN), Transaction ID (TID)
 - LSN is a monotonically increasing log record number
- **End (EOT)** LSN, TID, outcome (commit or abort)
- **UNDO** LSN, TID, before image
- **REDO** LSN, TID, after image

For next time:

- **CHECKPOINT** LSN, TID, state to limit how much is logged
- **CLR** LSN, TID, allows us to restart recovery

Recovery with NO FORCE / STEAL

- After crash, we must:
 - 😊 ○ REDO “winner” transactions that had committed
 - ☹ ○ UNDO “loser” transactions that had not committed
- Winner are transactions with SOT and COMMIT in log
- Losers are those with SOT and either (no EOT) or ABORT*
- Need to REDO winners from start to end
- Need to UNDO losers in reverse, from end to start
- Also need to UNDO aborted transactions

* Some disagreement in literature about whether ABORTed transactions are losers

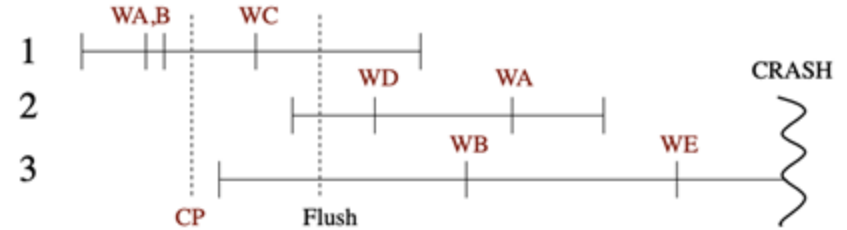
3 Phases of Recovery

- **Analysis**: Scan log to find winners and losers
- **REDO**: Scan log from beginning to end for winners
- **UNDO**: Scan log from end to beginning for losers

- Many possible ways to do this, e.g., UNDO then REDO or REDO then UNDO
 - Next time will see a specific proposal and analyze why

Simplistic protocol

- Normal execution: Physical write-ahead-logging
- Recovery: Replay log from beginning. We can recover the exact state at the crash (physical logging)



LSN	Type	Tid	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E

Simplistic protocol: Problem #1

- We don't want to REDO things that are already reflected on disk (i.e. do an operation twice). That's a problem for escrows (e.g. record += 1).
 - Easy: Just keep a pageLSN field in the page header that tells you the last LSN that modified the page (at the time the page was flushed).

LSN	Type	Tid	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
<hr/>				
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E



1. REDO

Disk	
Page	pageLSN
A	2
B	3
C	6
D	0
E	0



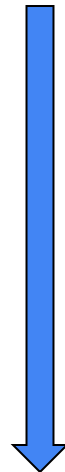
Skip if LSN \leq pageLSN



Simplistic protocol: Problem #2

- We want to recover to a state from which the user can resume normal operation (no pending transactions that were uncommitted at crash).
 - Easy: Just keep track of which transactions were not committed at crash and UNDO them. Can undo them logically (makes our life easier).

LSN	Type	Tid	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
<hr/>				
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E




1. REDO

lastLSN	TID
13	3
Losers	

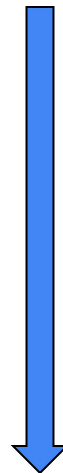


2. UNDO 

Simplistic protocol: Problem #3 (last one!)

- This is super slow! Imagine we need to replay a log containing months of transactions.
-  Only the last couple of log entries really need to be REDOne and UNDOne (with time pages get flushed & transactions commit)

LSN	Type	Tid	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
<hr/>				
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E



1. REDO


Disk	
Page	pageLSN
A	2
B	3
C	6
D	0
E	0



Skip if LSN \leq pageLSN



Simplistic protocol: Problem #3 (last one!)

- This is super slow! Imagine we need to replay a log containing months of transactions.
- Only the last couple of log entries really need to be REDOne and UNDOne
-  For each dirty page in the buffer pool, we keep track which was the first LSN that dirtied it. → For this page, we only need to REDO the log from there.


LSN	Type	Tid	PrevLSN	Data
1	SOT	1		
2	UP	1	1	A
3	UP	1	2	B
<hr/>				
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E

dirtyPgTable

pgNo	recLSN
A	2
B	3
C	6
D	8

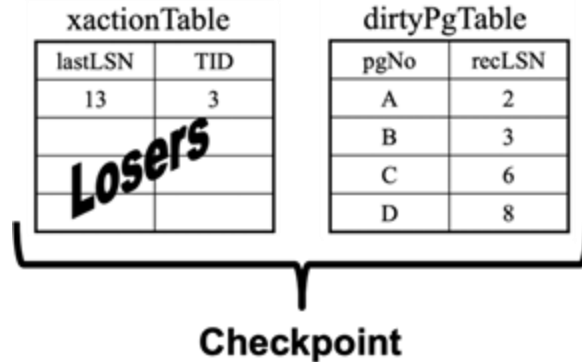
Simplistic protocol: Problem #3 (last one!)

- For each dirty page in the buffer pool, we keep track which was the first LSN that dirtied it. → For this page, we only need to REDO the log from there.
 - But where to keep this “dirty page table”?
 - Persist: Lots of logging and we need to force these writes to disk.
 - Memory: It will be lost at crash, so we need to start scanning the log from beginning again to build it.

-  Solution: Checkpoints: Keep in memory but periodically write to disk.
- You will need to scan some of the log to rebuild the dirty pages table, but not all of it!
- At the same time you avoid doing a lot of forced writes!

Simplistic protocol: Problem #3 (last one!)

- What do we need to checkpoint to only re-scan some of the log?
 - Dirty page table
 - Transaction table



Simplistic protocol: Problem #3 (last one!)

- What do we need to checkpoint to only re-scan some of the log?
 - Dirty page table
 - Transaction table



Summary / ARIES

- Normal operation:
 - Physical write-ahead-logging
 - Include LSN of last update in page headers (pageLSN)
 - Keep track of active transactions (for UNDO) and LSNs that first dirtied a page (for REDO) → checkpoint that periodically.
- Recovery:
 - **Analysis phase:** Start from last checkpoint in log and reconstruct transaction table and dirty page table at time of crash.
 - **REDO phase:** Physically REDO log records that haven't been flushed before crash. After that, your system will be in the state at the crash.
 - **UNDO phase:** UNDO transactions that weren't committed at the time of the crash ("losers").

Redo

- Where to begin?
 - $\text{Min}(\text{recLSN})!$ – earliest unflushed update
- Redo an update UNLESS:
 - Page is not in dirtyPgTable
 - Page flushed prior to checkpoint, didn't re-dirty
 - $\text{LSN} < \text{recLSN}$
 - Page flushed & re-dirtied prior to checkpoint
 - $\text{LSN} \leq \text{pageLSN}$
 - Page flushed after checkpoint
 - *Only step that requires going to disk*


dirtyPgTable

pgNo	recLSN
A	2
B	3
C	6
D	8
E	13


Disk

Page	pageLSN
A	2
B	3
C	6
D	0
E	0

UNDO

-  We need to log UNDOs
 - If we crash during recovery
 - If we crash while rolling back an aborted transaction
 - These log records are called Compensation Log Records (CLRs)
- Check where to start UNDO (lastLSN in Transaction Table) and UNDO each update going backwards using the prevLSN field in log

LSN	Type	Tid	PrevLSN	Data
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E




TransactionTable

lastLSN	TID
13	3

Losers

UNDO

-  We need to log UNDOs
 - If we crash during recovery
 - If we crash while rolling back an aborted transaction
 - These log records are called Compensation Log Records (CLRs)
- Check where to start UNDO (lastLSN in Transaction Table) and UNDO each update going backwards using the prevLSN field in log

LSN	Type	Tid	PrevLSN	Data
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E
14	CLR	3	13	E, 10



TransactionTable

lastLSN	TID
10	3

Losers

V ARIES


10. [10 points]: Which of the following statements about ARIES recovery are true?

- A. True / False** If a CLR (Compensation Log Record) is found in the log, the system must have crashed during the REDO phase of the ARIES Algorithm.
- B. True / False** In theory, if the recovery algorithm keeps crashing during recovery forever, then due to the CLR logs being added the size of the log can keep on increasing forever.
- C. True / False** Dirty pages are flushed to the disk at checkpoints.
- D. True / False** We can always get rid of the log before the second last checkpoint.
- E. True / False** PrevLSN is used to determine where to start the REDO phase from.

Topics

- Transactions
- Logging and recovery (ARIES)
- **Parallel/distributed databases (analytics and transactions)**
- Systems “potpourri”
 - High-performance transactional systems (H-Store / Calvin / Aurora)
 - Eventual consistency (DynamoDB)
 - Cluster computing (Spark)
 - Cloud analytics (Snowflake)
- Cardinality estimation

Distributed and Parallel Databases

-  Same semantics as a single-node ACID SQL database, but on multiple cores/machines
- Distributed databases must deal with node failures

Ways to Partition the Data

- **Round-robin**

- Perfect load-balancing (data-wise)
- Often all nodes need to participate in a query

- **Hash**

- Pretty good load balancing (unless many duplicates)
- Bad at range analytical queries (cannot easily skip partitions)


- **Range**

- Good at range / localized analytical queries
- Can be bad at load-balancing (data skew)

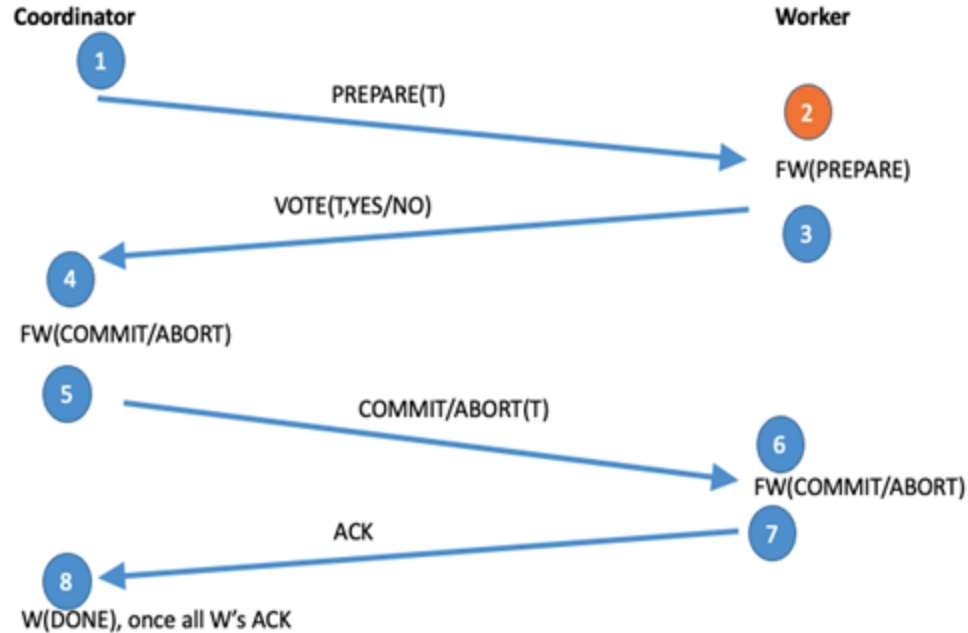
Parallel Joins (Hash Partitioning and Equijoins)

- **Partitioned on join attributes?** Run join locally on each partition.
- Otherwise, two options (non-exhaustive):
- **Re-partition (one or both tables): “shuffle join”**
 - Each node transmits and receives $(|T| / n) / n * (n - 1)$ bytes per repartitioned table
- **Replicate table across all nodes**
 - Each node transmits and receives $(|T| / n) * (n - 1)$ bytes

Distributed Transactions: Two Phase Commit

-  Distributed algorithm used to make a commit/abort decision **for multiple “sites”**
 - “Commit only if **all** participants agree to commit”
- Requires a coordinator
- Often considered a performance bottleneck

Two Phase Commit




Topics

- Transactions
- Logging and recovery (ARIES)
- Parallel/distributed databases (analytics and transactions)
- **Systems “potpourri”**
 - High-performance transactional systems (H-Store / Calvin / Aurora)
 - Eventual consistency (DynamoDB)
 - Cluster computing (Spark)
 - Cloud analytics (Snowflake)
- Cardinality estimation


Topics

- Transactions
- Logging and recovery (ARIES)
- Parallel/distributed databases (analytics and transactions)
- **Systems “potpourri”**
 - High-performance transactional systems (H-Store / Calvin / Aurora)
 - Eventual consistency (DynamoDB)
 - Cluster computing (Spark)
 - Cloud analytics (Snowflake)
- Cardinality estimation

CAP theorem

- Consistency, availability, partition-tolerance: You can have 2, not all 3 
- ACID has strong consistency but will appear down if machines go down or network becomes partitioned
- Many systems choose availability over consistency (e.g. NoSQL)


Dynamo

- Availability
- Partitioning
 - for scaling
 - consistent hashing 
- Replication
 - for fault tolerance and performance
 - 'N' successors in the ring stores the key
- Vector clocks for detecting conflicting writes

Vector Clock Updates

- Each coordinator maintains a version counter for **each data item** that increments for every write it coordinates
 - If a node stores m objects, it stores m vector clocks along with them
 - Each vector clock has n entries, which denote the number of writes done by each of n coordinators
- 🔑 ● Clock for one data item A at coordinator i
 - before: $V_A[1], \dots, V_A[i], \dots, V_A[n]$
 - after: $V_A[1], \dots, V_A[i]+1, \dots, V_A[n]$

Vector Clock

- Read - Read from the quorums
- E.g.: Read V1, V2, V3 - If one of these, say V1, is greater than the others for every component, V1 is the latest value and we can reconcile based on vector clocks
- What if they are incomparable? ---> i.e., can't decide which is the latest version of the data
 - $V1 = [1, 1]$, $V2 = [2, 0]$
 - Return both data versions, and use application-specific reconciliation 

Dynamo Question (2015)

Which of the following statements are true?

V1 = < R1 : 0, R2 : 3, R3 : 2 >

V2 = < R1 : **1**, R2 : 3, R3 : **2** >

V3 = < R1 : **0**, R2 : 0, R3 : **3** >

A) The writer that produced V1 observed V2

B) The writer that produced V2 observed V1

C) V2 and V3 are : "concurrent writes" (cannot be reconciled)

Topics

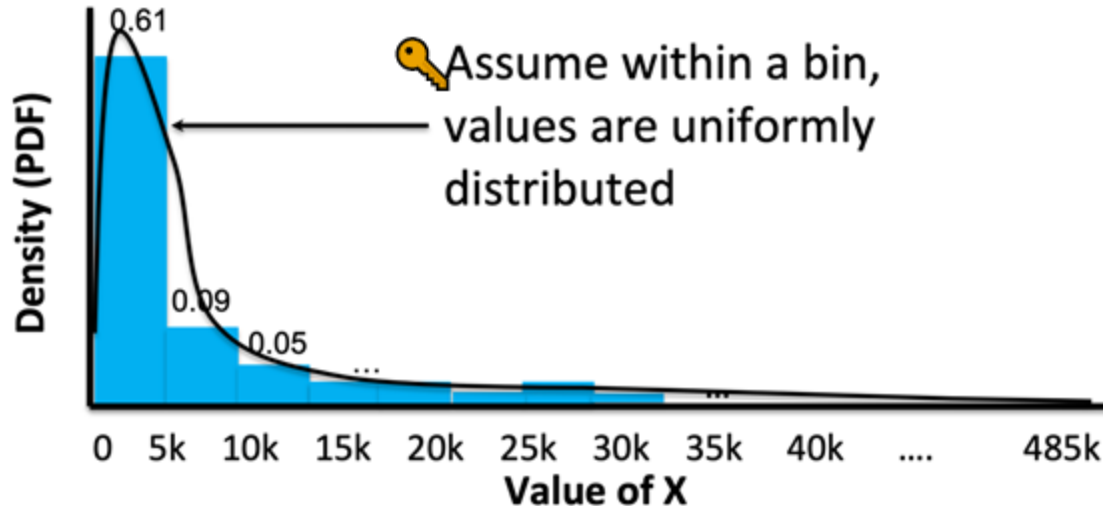
- Transactions
- Logging and recovery (ARIES)
- Parallel/distributed databases (analytics and transactions)
- **Systems “potpourri”**
 - High-performance transactional systems (H-Store / Calvin / Aurora)
 - Eventual consistency (DynamoDB)
 - **Cluster computing (Spark)**
 - Cloud analytics (Snowflake)
- Cardinality estimation

Topics

- Transactions
- Logging and recovery (ARIES)
- Parallel/distributed databases (analytics and transactions)
- Systems “potpourri”
 - High-performance transactional systems (H-Store / Calvin / Aurora)
 - Eventual consistency (DynamoDB)
 - Cluster computing (Spark)
 - Cloud analytics (Snowflake)
- Cardinality estimation

Equal-width histograms

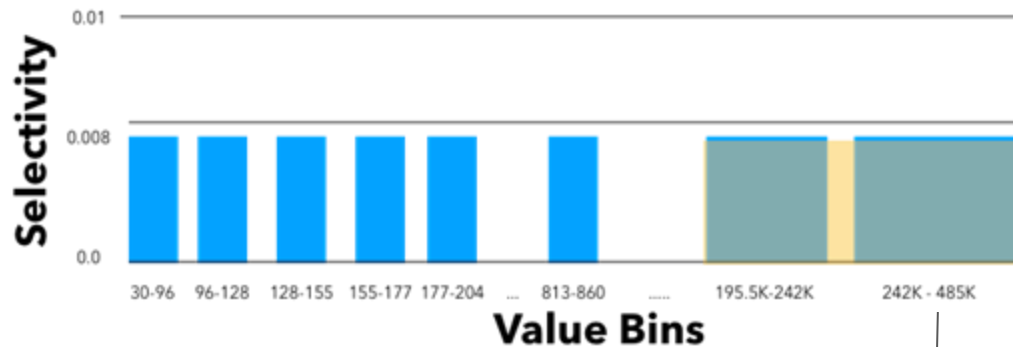
- Histograms can approximate any distribution (pdf) for a single attribute.
- Easy to build (ANALYZE): scan (sample of) one table.



Cardinality Estimation for one column

Equal width vs **Equal depth histograms**

100 bins with ~similar #values



Source of error: Within this large bucket, assume **uniformity**

Pros




- More detail where there is more data
---> uniformity assumption more accurate
- Fast to compute

Cons

- Less detail in other regions (e.g., in the large bins)

Cardinality Estimation for 2 columns

- Take selectivity estimates for single columns, and assume they are  **independent**, i.e. multiply selectivities.

Pros

- Fast to compute
- Don't need to store 2d distributions etc.

Cons

- Columns are often correlated (might severely misestimate then)
- Errors will accumulate as more columns / joins added

Main Assumptions

- **Uniformity**
 - Within a bin of histogram; (or when computing joins)
- **Independence**
 - When combining selectivities for multiple columns