Quiz 2 Review

December 5, 2022
Logistics

- **What**: Covers lectures 11 to 22 (inclusive)
- **When**: Wednesday during lecture (December 7, 2022 at 2:30 pm)
- **Where**: Lecture classroom (32-155)
- **Length**: 80 minutes
- **How**: Quiz will be on paper

- Open book/notes/laptop, but no Googling please.
- Class website access is permitted.
- Email staff for special accommodation (6.5830-staff@mit.edu)
Topics

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

● **Want:** Run transactions in parallel for performance reasons
● **How to ensure “correctness”?**
● 🪪 Create illusion of transactions running alone, one-by-one, on the database
● **Key property:** serializability, achieved through*
  ○ Two phase locking (2PL)
  ○ Optimistic concurrency control (OCC)

*Conflict serializability
Serializability

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>RA</td>
</tr>
<tr>
<td>RA</td>
<td>WA</td>
</tr>
<tr>
<td>WA</td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td>RB</td>
</tr>
<tr>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>

**RA:** Read A  
**WA:** Write A, may depend on anything read previously  

A/B are “objects” – e.g., records, disk pages, etc  

Assume arbitrary application logic between reads and writes

**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 

\textit{equivalent} to a serial ordering S' iff:

\begin{itemize}
  \item Every value read in S is the same value that was read by the 
  same read in S'.
  
  \item The final write of every object is done by the same transaction 
  T in S and S'
  
  \item 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.
\end{itemize}
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.
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

<table>
<thead>
<tr>
<th></th>
<th>T2</th>
<th>T1</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lock Compatibility Table
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
Deadlocks

• Possible for Ti to hold a lock Tj needs, and vice versa
Complex Deadlocks Are Possible

T1 waits for T2 → RB
   WA

T2
   RB
   WB

RC ← T2 waits for T3
   WC

T3
   RA ← T3 waits for T1
   WA
   RA

Waits-for graph:
Cycle → Deadlock
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

T2 waits for T3
- RC
- WC

Equivalent to T2 - T1

Waits-for graph
Cycle → Deadlock
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).*
Solving Phantoms

- Need a way to lock ranges
- Common approach: next key locking

On insert(val), Xlock ij next pointer if val > max(page i) and < min(page(j))

Only works for ranges with indexes
For unindexed tables, must read the whole table, so just use a table lock
More details next lecture!
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
Serial Validation

validateAndWrite(pastT[], start_tn, my_read_set, my_write_set)
{
    lock();
    int finish_tn = tnc;  //prior transaction
    bool valid = true;
    for(int t = start_tn + 1; t <= finish_tn; t++)
        if(pastT[t].write_set intersects with my_read_set)
            valid = false;
    if (valid) {
        write_phase();
        tnc = tnc+1;
        tn = tnc;
    }
    unlock();
}
What If Serializability Isn’t Needed?

• E.g., application only needs to read committed data
• Databases provide different isolation levels
  – READ UNCOMMITTED
    • Ok to read other transaction’s dirty data
  – READ COMMITTED
    • Only read committed values
  – REPEATABLE READS
    • If R1 read A=x, R2 will read A=x ∀ A

• Many database systems default to READ COMMITTED
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
Topics

- Transactions
- Logging and recovery (ARIES)
- Parallel/distributed databases (analytics and transactions)
- Systems “potpourri”
  - High-performance transactional systems (H-Store / Calvin)
  - Eventual consistency (DynamoDB)
  - Cluster computing (MapReduce / Spark)
- Cardinality estimation
Database State During Query Execution

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.

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
STEAL/NO FORCE ↔ UNDO/REDO

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

<table>
<thead>
<tr>
<th></th>
<th>FORCE</th>
<th>NO FORCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEAL</td>
<td>UNDO</td>
<td>UNDO &amp; REDO</td>
</tr>
<tr>
<td>NO STEAL</td>
<td>? UNDO</td>
<td>REDO</td>
</tr>
</tbody>
</table>

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

**Force:** Force writes to disk on txn commit.

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

In SimpleDB, 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
# ARIES Example

## Diagram

![ARIES Example Diagram](image)

## Table

<table>
<thead>
<tr>
<th>LSN</th>
<th>Type</th>
<th>Tid</th>
<th>PrevLSN</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SOT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>UP</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>UP</td>
<td>1</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>CP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SOT</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>UP</td>
<td>1</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>SOT</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>UP</td>
<td>2</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>9</td>
<td>EOT</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>UP</td>
<td>3</td>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>11</td>
<td>UP</td>
<td>2</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>12</td>
<td>EOT</td>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>UP</td>
<td>3</td>
<td>10</td>
<td>E</td>
</tr>
</tbody>
</table>
ARIES Normal Operation

• Two key data structures:
  – *Transaction table* -- list of active transactions
  – *Dirty page table* -- List of pages that have been modified and not yet written to disk

Data Structures (which stay in memory) are updated as the system runs:
  Buffer pool pages are flushed *asynchronously*, a few at a time, to disk
  Writes occur on the page, then force log right before flush
 Flushes are not logged
Log entries are forced on COMMIT
Transaction Table

<table>
<thead>
<tr>
<th>lastLSN</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>3</td>
</tr>
</tbody>
</table>

- All active transactions in table
- `lastLSN`: most recent log record written by that transaction
Dirty Page Table

<table>
<thead>
<tr>
<th>pgNo</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>A</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
</tr>
</tbody>
</table>

Dirty pages are periodically flushed to disk by a background process (flushes are not logged)
On flush, remove from dirtyPageTable

- One entry for each page that has been modified but not flushed to disk
- recLSN: log record that first dirtied the page
Checkpoints

• Taken periodically
• Record the state of the dirty page table and transaction table
  - Doesn’t require pages to be flushed to disk during checkpoint
• Allow us to limit amount of log we have to keep and replay during crash
ARIES Approach: 3 Log Passes

FW = Forward Pass; BW = Backward Pass

• Analysis, to see what needs to be done (FW)
• Redo, to ensure DB reflects updates that are in the log but not in tables (FW)
  – Including those that belong to txns that will eventually be rolled back!
  – Why? Ensures “action consistent” state -- which will allow logical undo.
  – “Repeating History”
• Undo, to rollback losers (BW)
Analysis Pass

• Goal: reconstruct the state of the transaction table and the dirty page table at the time the crash occurred.

• Play log forward
  – Add and remove xactions to/from the transaction table on SOT and COMMIT/ABORT
  – Update the lastLSN on writes
  – Update the dirty page table as writes happen
Redo

- Where to begin?
  - Checkpoint?
  - Min(recLSN)! – earliest unflushed update

- What to REDO
  - Everything?
    - Slow
    - Problematic if using operational (escrow) logging
  - Redo an update UNLESS:
    - Page is not in dirtyPgTable
      - Page flushed prior to checkpoint, didn’t redirty
    - LSN < recLSN
      - Page flushed & redirtied prior to checkpoint
    - LSN <= pageLSN
      - Page flushed after checkpoint

*Only step that requires going to disk*
REDO Conditions Example


Min(recLSN)  Flush  Checkpoint  Flush  dirtyPgTable @ CP

Redo an update UNLESS:
- Page is not in dirtyPgTable
- Page flushed prior to checkpoint, didn’t redirty
- LSN < recLSN
- Page flushed & redirtied prior to checkpoint
- LSN <= pageLSN
  Page flushed after checkpoint

A/LSN 3
B/LSN 2
C/LSN 6

Disk

<table>
<thead>
<tr>
<th>Page</th>
<th>pageLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pgNo</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>
Compensation Log Records (CLRs)

- CLR record written after each UNDO
- Avoid repeating UNDO work
- Why?
  - Because UNDO is logical, and we don't check if records have already been UNDONE. Could get into trouble if re-undid some logical operation.
## UNDO with CLR

<table>
<thead>
<tr>
<th>LSN</th>
<th>Type</th>
<th>Tid</th>
<th>PrevLSN</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>SOT</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>UP</td>
<td>1</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>SOT</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>UP</td>
<td>2</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>9</td>
<td>EOT</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>UP</td>
<td>3</td>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>11</td>
<td>UP</td>
<td>2</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>12</td>
<td>EOT</td>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>UP</td>
<td>3</td>
<td>10</td>
<td>E</td>
</tr>
<tr>
<td>14</td>
<td>CLR</td>
<td>3</td>
<td>13</td>
<td>E, 10</td>
</tr>
<tr>
<td>15</td>
<td>CLR</td>
<td>3</td>
<td>14</td>
<td>B, 5</td>
</tr>
<tr>
<td>16</td>
<td>EOT</td>
<td>3</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
REDO with CLR

- REDO CLRs on crash recovery
  - Use REDO rules to check if updates in CLRs have already been done
    - Avoids repeating operational (escrow) operations

- After processing CLR, update lastLSN field in dirtyPgTable
  - Allows UNDO to start from the right place, should we checkpoint while UNDOing
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)
  - Eventual consistency (DynamoDB)
  - Cluster computing (MapReduce / Spark)
- Cardinality estimation
Distributed and Parallel Databases

- Same semantics as a single-node ACID SQL database, but on multiple cores/machines

- Parallel databases are more about performance

- Distributed databases must deal with node failures
Implementation

Architectures:

- Shared-everything
- Shared-disk
- Shared-nothing
- Shared-nothing w/ distributed file system

Parallelism:

- Pipeline
- Partitioning
- Replication
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 \( \frac{|T|}{n} / n \times (n - 1) \) bytes per repartitioned table
  - **Replicate table across all nodes**
    - Each node transmits and receives \( \frac{|T|}{n} \times (n - 1) \) 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 – High-Level Sketch

A. Log start of transaction
B. Execute transaction on worker nodes
C. Vote YES if can commit
D. Log transaction commit if all YES
E. Tell each worker to commit
F. Log transaction complete
Topics

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

- Running old code on new hardware ≠ speed-up
- New performance bottlenecks
- 2PC is slow
- Consider new architectures!

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. SIGMOD 2008
H-Store

- **Distributed in-memory DBMS**
  - Often enough memory to store the entire dataset

- **Partition the data; single-thread per partition**
  - Eliminate coordination overhead within a partition

- **Stored-procedure transactions**
  - Optimize partitioning for the workload
  - Avoid waiting for the client

- **Weaknesses**
  - Stored-procedure assumption
  - Multi-partition transactions?

Diagrams courtesy of Prof. Andy Pavlo
Calvin

- Distributed DBMS

- 🖼️ Deterministic execution
  - Avoid distributed coordination (2PC) during transaction execution by agreeing on commit order up front!
  - Ordering performed in batches

- Grant locks to transactions in agreed-upon order

- Weaknesses
  - Read/write sets needed
  - Stored-procedure assumption
  - Potentially increased transaction latency
Topics

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

- In general, trade-off between availability and consistency

- 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.

- Clock at coordinator i
  - before: V[1], ... V[i] ..., V[N]
  - after: V[1], ... V[i] + 1 ... V[N]

- Send V to write quorum
- Receiver will update V to max of what it gets
- $V = \text{max} \ (V, V_{\text{recv}})$
Vector Clock

- Read - Read from the quorums
- 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)

V1 =< R1 : 0, R2 : 3, R3 : 2 >
V2 =< R1 : 1, R2 : 3, R3 : 2 >
V3 =< R1 : 0, R2 : 0, R3 : 3 >

- The writer that produced V1 observed V2
Dynamo Question (2015)

$V_1 = \langle R_1 : 0, R_2 : 3, R_3 : 2 \rangle$

$V_2 = \langle R_1 : 1, R_2 : 3, R_3 : 2 \rangle$

$V_3 = \langle R_1 : 0, R_2 : 0, R_3 : 3 \rangle$

- The writer that produced $V_1$ observed $V_2$
- The writer that produced $V_2$ observed $V_1$
Dynamo Question (2015)

V1 =< R1 : 0, R2 : 3, R3 : 2 >
V2 =< R1 : 1, R2 : 3, R3 : 2 >
V3 =< R1 : 0, R2 : 0, R3 : 3 >

- The writer that produced V1 observed V2
- The writer that produced V2 observed V1
- V2 and V3 are concurrent writes
MapReduce

- Paradigm for distributed programs
- Map and Reduce functions
Example: Word Count

map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
Example: Word Count

map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += parseInt(v);
    Emit(AsString(result));
MapReduce: Pros and Cons

Pros
- Fault Tolerant
  - Worker dies / Straggler ---> just use another worker
- Hides distributed system complexity

Cons
- Large overhead for reusing data in iterative or interactive tasks (I/O, replication, serialization)
- Map <-> Reduce framework isn’t fully general
Spark

- Distributed “dataflow” language
- Programs operate on partitions of data in parallel
- More general than MapReduce
  - allows joins and other usual DB operators
Spark: Memory use

- Several in-memory storage options for intermediate results that are reused
- LRU caches / or user specified cache priorities

Very useful for interactive or iterative workloads, e.g., ML tasks that train over same data periodically
Spark: Lineage

- Limit operations to coarse-grained transformations and only log the transformations instead of replicating data for recovering —> **Lineages**

Use cases to recover from failure:

- Short lineage chain? 🔍
  - Just recompute from lineage
- Long lineage chain with narrow dependencies?
  - Fast to recompute from lineage using pipelined execution
- Long lineage chain with wide dependencies?
  - Checkpoint intermediate results to stable storage!
Topics

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

Equal width vs Equal depth histograms

Source of error: Within this large bucket, assume uniformity
Cardinality Estimation for one column

Equal width vs Equal depth histograms

- **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)
  - Requires Most Common Values for outliers

Source of error: Within this large bucket, assume uniformity
Cardinality Estimation for 2 columns

- Take selectivity estimates for single columns, and assume they are independent. Multiply them together.

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

- Cons
  - If columns were positively correlated, we under-estimate
  - over-estimate if negatively correlated
  - 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