Finals Review
Last year’s Test2 during COVID (90 mins)

- Similar in flavor

- Differences
  1. 180 mins vs 90 mins
  2. Review of pre-midterm material (~30-35%)
     a. SQL, Sorting/Hashing, Indexing
  3. Main emphasis (60-70%)
     a. “Amazon recommender style example”
     b. Transactions -- Logging, Locking
     c. Bigschemas - FDs, closures, superkeys

- Style of questions
  - Most are similar to psets, lectures
  - “1” fun question – you’ll need to “apply” 1-2 principles to answer
Course Summary

We’ll learn How To...

- **Query** over small-med-large data sets with **SQL**? [Weeks 1 and 2]
  - On relational engines, and “big data” engines

- **Scale** for “big queries”? On Clusters? [Weeks 3, 4, 5]
  - OLAP/Analytics, 1st principles of scale

- **Scale** for “big writes”? [Weeks 6, 7, 8]
  - Writes, Transactions, Logging, ACID properties

- **Design** “good” databases? [Weeks 9, 10]
  - Big Schemas, design, functional dependencies, query optimizers

**Project:** Query-Visualize-Learn on GB/TB scale data sets on a Cloud [sql + python]

**Industry Talks:** Real world talks from Google, Uber, Coinbase
Takeaway ⇒ How to apply CS concepts at scale to solve big problems?

- How do we scale queries by 10-100x, transactions by 10-100x
- How to work with real data sets? (bigschemas, projects, etc)
Takeaway

IO Hierarchy

⇒ Rest of cs145: Focus on simplified RAM + Disk model (learn tools for other IO models)
How to apply CS concepts at scale to solve big problems?
(How do we scale queries by 10-100x, transactions by 10-100x)
Big Scale Lego Blocks

Roadmap

Primary data structures/algorithms

**Hashing**
- HashTables
  - \( \text{hash}_i(\text{key}) \rightarrow \text{location} \)

**Sorting**
- BucketSort, QuickSort
- MergeSort

**HashFunctions**
- \( \text{hash}_i(\text{key}) \rightarrow \text{location} \)

**MergeSortedFiles**
- MergeSort
- MergeSort
Example Visa DB

Transaction Queue
- 60000 user TXNs/sec
- Monthly 10% Interest TXN

Design#1 VisaDB
For each Transaction in Queue
- For relevant records
  - Use 2 PL to acquire/release locks
  - Process record
  - WAL Logs for updates
- Commit or Abort

Visa DB

- Account Balance ($)
  - 3001 500
  - 6001 150
  - 5001 20
  - 6001 60
  - 3002 80
  - 4002 -200
  - 5002 320
  - 8018 -100
  - 40008 150
  - 50002 20

Commit
WAL Flush to disk
How to scale queries 10-100x on large data sets?
Example: Youtube DB
Relational model (aka tables)

Simple, popular algebra (E.F. Codd et al)

Every relation has a schema

Logical Schema: describes types, names
Physical Schema: describes data layout
Virtual Schema (Views): derived tables

SQL to express queries declaratively

World’s most successful parallel programming language
RDBMS Architecture

How does a SQL engine work?

1. **SQL Query**
   - Declarative query (from user)

2. **Relational Algebra (RA) Plan**
   - Translate to relational algebra expression

3. **Optimized RA Plan**
   - Find logically equivalent - but more efficient - RA expression

4. **Execution**
   - Execute each operator of the optimized plan!
Optimization

Build Query Plans

Roadmap

1. For SFW, Joins queries
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

Cost in I/O, resources?
To query, maintain?
IO Aware algorithms

A class of algorithms which try to minimize IO, and *effectively ignore cost of operations in main memory*
**External Merge Sort Algorithm**

**Goal:** Sort a file that is much bigger than the buffer

**Key idea:**
- **Phase 1:** Split file into smaller chunks (“initial runs”) which can be sorted in memory
- **Phase 2:** Keep merging (do “passes”) using external merge algorithm until one sorted file!
Join Algorithms: Overview

- NLJ: An example of a non-IO aware join algorithm
- BNLJ: Big gains just by being IO aware & reading in chunks of pages!
- SMJ: Sort R and S, then scan over to join!
- HPJ: Partition R and S into buckets using a hash function, then join the (much smaller) matching buckets

For R \bowtie S on A

Quadratic in P(R), P(S) 
I.e. O(P(R)*P(S))

Given sufficient buffer space, linear in P(R), P(S) 
I.e. ~O(P(R)+P(S))

By only supporting equijoins & taking advantage of this structure!
**Goal:** Execute $R \bowtie S$ on $A$

**Key Idea:** We can sort $R$ and $S$, then just scan over them!

**IO Cost:**
- *Sort phase:* $\text{Sort}(R) + \text{Sort}(S) \approx 2(P(R) + P(S))$
- *Merge/join phase:* $\approx P(R) + P(S) + \text{OUT}$
Hash Join

- **Goal:** Execute $R \bowtie S$ on $A$

- **Key Idea:** We can partition $R$ and $S$ into buckets by hashing the join attribute-then just join the pairs of (small) matching buckets!

- **IO Cost:**
  - Hash Partition phase: $2(P(R) + P(S))$ each pass
  - Partition Join phase: Depends on size of the buckets… can be $\sim P(R) + P(S) + \text{OUT}$ if they are small enough!
  - *Can be worse due to skew!*
Billion products

User searches for “coffee machine”

Product recommendations

Customers who viewed this item also viewed these products

- **Dualit Food XL1500 Processor**
  - $580
  - Add to cart

- **Kenwood kMix Manual Espresso Machine**
  - **4.5 stars**
  - $250
  - Select options

- **Weber One Touch Gold Premium Charcoal Grill-57cm**
  - $225
  - Add to cart

- **NoMU Salt Pepper and Spice Grinders**
  - $3
  - View options
Counting popular product-pairs

**Story:** Amazon/Walmart/Alibaba (AWA) want to sell products

1. AWA wants fast user searches for product
2. AWA shows ‘related products’ for all products so users can explore
   - Using **collaborative filtering** (‘wisdom of crowds’) from historical website logs.
   - Each time a user views a set of products, those products are related (co-occur)

⇒ Goal: compute product pairs and their co-occur count, across all users

**Data input:**
- AWA has **1 billion products**. Each product record is ~1MB (descriptions, images, etc.).
- AWA has **10 billion UserViews** each week, from 1 billion users. Stored in **UserViews**, each row has <userID, productId, viewID, viewTime>.
Data Systems Design

Example:

User search for product
1. Lookup Product Index
2. Lookup Products image
3. Lookup CoOccur Index
4. Capture user browsing info to UserViews

[1] + [2] + [3] < 100 msecs??
Counting popular product-pairs

**Your mission:** Design an efficient system to compute co-occur counts on *Sundays* from weekly logs and produce a **CoOccurCount** table <productID, productID, count>

1. AWA’s data quality magicians recommend
   - (a) keep only **top billion** popular pairs, and (b) drop pairs with co-occur counts less than million.
   - (c) Also, assume users view **ten products on average each week** *(User is interested in ~10 products/week, not 1000s)*.

2. For simplicity, **SortedUserViews** is stored sorted by <userID, productID>.
   - You can sequentially scan the log and produce co-occurring product pairs for each user. In other words, output \((p_i, p_j)\) if a user viewed products \(p_i\) and \(p_j\).
   - This “stream” of tuples (**TempCoOccur**) may then be (a) stored on disk or (b) discarded after updating any data structures.
Plan#1: With 1 machine, use RAM to count (Cost = 25B$ or > 100 million years).

Plan#2: With 1 machine

Plan 2
1. Scan SortedUserViews. For each user, append \(<p_i, p_j>\) to a file TempCoOccurLog if the user has viewed \(p_i\) and \(p_j\). (i.e., produce per-user co-occur product pair. Append to log ⇒ No seek...)
2. Externally sort TempCoOccurLog on disk, so identical product pairs are adjacent to each other in the sorted file
3. Scan sorted TempCoOccurLog. With a single pass, you can count co-occur pairs. Drop co-occur pairs with < 1 million.

<table>
<thead>
<tr>
<th>Nespresso</th>
<th>Iphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Nespresso</td>
<td>Iphone</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Nespresso</td>
<td>Iphone</td>
</tr>
</tbody>
</table>

TempCoOccurLog (After Step 1)

<table>
<thead>
<tr>
<th>.....</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nespresso</td>
</tr>
</tbody>
</table>

Sorted TempCoOccurLog (After Step 2)

<table>
<thead>
<tr>
<th>Count sorted TempCoOccurLog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nespresso</td>
</tr>
<tr>
<td>Nespresso</td>
</tr>
<tr>
<td>Nespresso</td>
</tr>
<tr>
<td>Nespresso</td>
</tr>
<tr>
<td>.....</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>ProductId</td>
</tr>
<tr>
<td>UserID</td>
</tr>
<tr>
<td>ViewID</td>
</tr>
<tr>
<td>Product</td>
</tr>
<tr>
<td>SortedUserViews</td>
</tr>
<tr>
<td>CoOccur (for top 1 Billion)</td>
</tr>
<tr>
<td>TempCoOccurLog (assume: ~10 product views/user)</td>
</tr>
</tbody>
</table>
### Systems Design Example:

**Product CoOccur Plan #2**

<table>
<thead>
<tr>
<th>Steps</th>
<th>Cost (IO)</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan SortedUserViews</td>
<td>4 M</td>
<td>240GB (4 M pages)</td>
</tr>
<tr>
<td>Append (&lt;p_i, p_j&gt;) to TempCoOccurLog</td>
<td>12.5M</td>
<td>800 GB (12.5M pages)</td>
</tr>
<tr>
<td>Externally sort TempCoOccurLog on disk (Assume sort cost is ~2N, where N is number of pages for table and B is number of buffers, and B (\sim N))</td>
<td>25M</td>
<td>IO cost is (appx) (= 2^*N = 2^*12.5M)</td>
</tr>
<tr>
<td>Scan TempCoOccurLog (sorted) and keep counts in CoOccur</td>
<td>12.5M</td>
<td>800 GB</td>
</tr>
</tbody>
</table>

Total IO cost = \((4M + 12.5M + 25M + 12.5M) = 54M\)

Recall: Scan at 100 MBps, then time (secs) [assume, files are stored sequentially]

\[= (54M * 64 KB) / 100 MBps = \sim 34.5K \text{ secs}\]
How to scale transactions by 10-100x for large data?
Transactions in SQL

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

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

```sql
START TRANSACTION
  UPDATE Bank SET amount = amount - 100
  WHERE name = 'Bob'
  UPDATE Bank SET amount = amount + 100
  WHERE name = 'Joe'
COMMIT
```
Example Visa DB

Transaction Queue
- 60000 user TXNs/sec
- Monthly 10% Interest TXN

Design#1 VisaDB
For each Transaction in Queue
- For relevant records
  - Use 2 PL to acquire/release locks
  - Process record
  - WAL Logs for updates
- Commit or Abort

Commit
WAL Flush to disk

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>150</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-20</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30108</td>
<td>-100</td>
</tr>
<tr>
<td>40008</td>
<td>150</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>
Example Problem 1

Monthly bank interest transaction

With crash

'T-Monthly-423'

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>500</td>
</tr>
<tr>
<td>4001</td>
<td>100</td>
</tr>
<tr>
<td>5001</td>
<td>20</td>
</tr>
<tr>
<td>6001</td>
<td>60</td>
</tr>
<tr>
<td>3002</td>
<td>80</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>30108</td>
<td>-100</td>
</tr>
<tr>
<td>40008</td>
<td>100</td>
</tr>
<tr>
<td>50002</td>
<td>20</td>
</tr>
</tbody>
</table>

Money (@10:45 am)

<table>
<thead>
<tr>
<th>Account</th>
<th>Balance ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3001</td>
<td>550</td>
</tr>
<tr>
<td>4001</td>
<td>110</td>
</tr>
<tr>
<td>5001</td>
<td>22</td>
</tr>
<tr>
<td>6001</td>
<td>66</td>
</tr>
<tr>
<td>3002</td>
<td>88</td>
</tr>
<tr>
<td>4002</td>
<td>-200</td>
</tr>
<tr>
<td>5002</td>
<td>320</td>
</tr>
<tr>
<td>30108</td>
<td>-110</td>
</tr>
<tr>
<td>40008</td>
<td>110</td>
</tr>
<tr>
<td>50002</td>
<td>22</td>
</tr>
</tbody>
</table>

Did T-Monthly-423 complete?
Which tuples are bad?

Case 1: T-Monthly-423 was crashed
Case 2: T-Monthly-423 completed. 4002 deposited 20$ at 10:45 am
Why study Transactions?
Good programming model for parallel applications on shared data!
- Atomic
- Consistent
- Isolation
- Durable

Design choices?
- Write update Logs (e.g., WAL logs)
- Serial? Parallel, interleaved and serializable?

"Need to Master Extreme Transactions" (Forbes Insights)
Transaction Properties: ACID

- **Atomic**
  - State shows either all the effects of txn, or none of them

- **Consistent**
  - Txn moves from a state where integrity holds, to another where integrity holds

- **Isolated**
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)

- **Durable**
  - Once a txn has committed, its effects remain in the database
Write-Ahead Logging (WAL)

Algorithm: WAL

For each tuple update, write Update Record into LOG-RAM

Follow two Flush rules for LOG
- Rule1: Flush Update Record into LOG-Disk before corresponding data page goes to storage
- Rule2: Before TXN commits, - Flush all Update Records to LOG-Disk - Flush COMMIT Record to LOG-Disk

→ Durability
→ Atomicity

Transaction is committed once COMMIT record is on stable storage
Example WAL (+S2PL) scenarios

| TXN commit before COMMIT Record on disk? | No |
| Data page flushed before its Update Record flushed? | No |
| TXN commit before modified data page flushed? | Yes, often. Especially for large transactions. For TXN Commit, should have Flushed...  
- All Update Records to Log  
- COMMIT record to Log |
| TXN updated “Bob” and committed. TXN2 needs committed data record for ‘Bob’; record still in RAM, not flushed to disk | TXN2 requests/gets LOCK to get latest from memory. |
| TXN3 updated “John” but not yet committed. TXN4 needs updated record for “Tom” | TXN4 requests LOCK. Waits for TXN3. |

### Rules

**Rule1:** For each tuple update
- Flush Update Record to LOG-Disk

**Rule2:** Before TXN commits
- Flush COMMIT Record to LOG-Disk
Example

Monthly bank interest transaction

### ‘T-Monthly-423’
- Monthly Interest 10%
- 4:28 am Starts run on 10M bank accounts
- Takes 24 hours to run

```sql
START TRANSACTION
UPDATE Money
SET Balance = Balance * 1.1
COMMIT
```
Example

Monthly bank interest transaction

With crash

‘T-Monthly-423’

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

Network outage at 10:29 am,
System access at 10:45 am

Did T-Monthly-423 complete?
Which tuples are bad?

Case1: T-Monthly-423 was crashed
Case2: T-Monthly-423 completed. 4002 deposited 20$ at 10:45 am
Example

Monthly bank interest transaction

Recovery

System recovery (after 10:45 am)

1 Rollback uncommitted transactions
   - Restore old values from WALlog (if any)
   - Notify developers about aborted txn
1.1 Redo Recent committed transactions (w/ new values)
2 Back in business
3 Redo (any pending) transactions

(Sometimes swap 2 and 3, as a tradeoff)
**Example**

**Monthly bank interest transaction**

**Performance**

---

**Cost to update all data**

100M bank accounts → 100M seeks? (worst case)

(@10 msec/seek, that’s 1 Million secs)

---

**Cost to Append to log**

+ 1 seek to get ‘end of log’
+ write 100M log entries sequentially (fast!!! < 10 sec)

[Lazily update data on disk later, when convenient.]

---

**Speedup for TXN Commit**

1 Million secs vs 10 sec!!!
Example- consider two TXNs:

<table>
<thead>
<tr>
<th>T1: START TRANSACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPDATE</strong> Accounts</td>
</tr>
<tr>
<td><strong>SET</strong> Amt = Amt + 100</td>
</tr>
<tr>
<td><strong>WHERE</strong> Name = 'A'</td>
</tr>
<tr>
<td><strong>UPDATE</strong> Accounts</td>
</tr>
<tr>
<td><strong>SET</strong> Amt = Amt - 100</td>
</tr>
<tr>
<td><strong>WHERE</strong> Name = 'B'</td>
</tr>
<tr>
<td>COMMIT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>T2: START TRANSACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPDATE</strong> Accounts</td>
</tr>
<tr>
<td><strong>SET</strong> Amt = Amt * 1.06</td>
</tr>
<tr>
<td>COMMIT</td>
</tr>
</tbody>
</table>

T2 credits both accounts with a 6% interest payment

**Note:**
1. DB does not care if T1 —> T2 or T2 —> T1 (which TXN executes first)
2. If developer does, what can they do? (Put T1 and T2 inside 1 TXN)
Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions.

- A and B are **equivalent schedules** if, for any database state, the effect on DB of executing A is identical to the effect of executing B.

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

The word “some” makes this def powerful and tricky!
### Serial Schedules

- **S1**: T1: A += 100, B -= 100; T2: A *= 1.06, B *= 1.06
- **S2**: T1: A *= 1.06, B *= 1.06; T2: A += 100, B -= 100

### Interleaved Schedules

- **S3**: T1: A += 100; T2: A *= 1.06, B *= 1.06
- **S4**: T1: A += 100; T2: A *= 1.06, B *= 1.06
- **S5**: T1: A += 100; T2: A *= 1.06, B *= 1.06
- **S6**: T1: A += 100; T2: A *= 1.06, B *= 1.06

### Serially Equivalent Schedules

- S1 and S2 are equivalent.

### Serializable Schedules

- S3 and S4 (and S1 and S2).

### Equivalent Schedules

- <S1, S3>
- <S2, S4>

### Non-serializable (Bad) Schedules

- S5 and S6
General DBMS model: Concurrency as Interleaving TXNs

**Serial Schedule**

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A)</th>
<th>W(A)</th>
<th>R(B)</th>
<th>W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Interleaved Schedule**

<table>
<thead>
<tr>
<th>T1</th>
<th>R(A)</th>
<th>W(A)</th>
<th>R(B)</th>
<th>W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>R(A)</td>
<td>W(A)</td>
<td>R(B)</td>
<td>W(B)</td>
</tr>
</tbody>
</table>

Each action in the TXNs reads a value from global memory and then writes one back to it.

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

We call the particular order of interleaving a **schedule**.
Conflict Types

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

Thus, there are three types of conflicts:
- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

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

Two schedules are **conflict equivalent** if:

- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions of two TXNs are ordered in the same way*

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

**Conflict serializable $\Rightarrow$ serializable**
So if we have conflict serializable, we have consistency & isolation!
The Conflict Graph

• Let’s now consider looking at conflicts at the TXN level

• Consider a graph where the nodes are TXNs, and there is an edge from $T_i \rightarrow T_j$ if any actions in $T_i$ precede and conflict with any actions in $T_j$
What can we say about “good” vs. “bad” conflict graphs?

**Serial Schedule:**

\[ T_1 \rightarrow T_2 \]

**Interleaved Schedules:**

\[ T_1 \rightarrow T_2 \]

\[ T_2 \rightarrow T_1 \]

**Theorem:** Schedule is conflict serializable if and only if its conflict graph is acyclic.
Good or Bad schedule? Conflict serializable?

Step 1
Find conflicts (RW, WW, WR)

Step 2
Build Conflict graph
Acyclic? Topo Sort

Step 3
Example serial schedules
Conflict Equiv to S1

---

### Schedule S1

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1(A)</td>
<td>r2(A)</td>
<td>w1(B)</td>
<td>w3(C)</td>
<td>r2(C)</td>
<td>r4(B)</td>
</tr>
</tbody>
</table>

### Example with 5 Transactions

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1(A)</td>
<td>r2(A)</td>
<td>w1(B)</td>
<td>w3(C)</td>
<td>r2(C)</td>
<td>w2(D)</td>
</tr>
<tr>
<td>r4(B)</td>
<td>w4(E)</td>
<td>r5(D)</td>
<td>w5(E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Examples:

- **SerialSched (SS1)**
  - `w1(A)`
  - `w1(B)`
  - `w3(C)`
  - `r4(B)`
  - `w2(D)`
  - `r5(D)`
  - `w5(E)`

- **SerialSched (SS2)**
  - `w4(E)`
  - `r5(D)`
  - `w5(E)`

---

**Acyclic** ⇒ Conflict serializable! ⇒ Serializable
Strict 2-Phase Locking (S2PL)

2-Phase Locking: A transaction cannot request additional locks once it releases any locks. [Phase1: “growing phase” to get more locks. Phase2: “shrinking phase”]

Strict 2-PL: Release locks only at COMMIT (COMMIT Record flushed) or ABORT
Example with 5 Transactions (2PL)

Schedule S1

Execute with S2PL

T1 T2 T3 T4 T5

Waits-For Graph
Example with 5 Transactions (2PL)

Schedule S1

Execute with S2PL

Step 0
X (A)
w1(A)

Step 1
X (B)
w1(B)
Req S(A)

Step 2
X (B)
w1(B)
Unl B, A

Step 3
Get S(A)
S(A)
r2(A)
X (C)
w3(C)
Unl B, A

Step 4
S(C)
r2(C)
X (C)
w3(C)
Unl C

Step 5
S(B)
r2(C)
X (D)
w2(D)
Unl A, C, D

Step 6
S(B)
r4(B)
X (E)
w4(E)
Unl B, E

Step 7
S(D)
r5(D)
X (E)
w5(E)
Unl D, E

Step 8

Step 9

Step 10
Transaction Queue

- 60000 TXNs/sec
- Monthly Interest TXN

Example Visa DB -- Need Higher Performance?

`T-Monthly-423`
Monthly Interest 10%
4:28 am Starts run on 10M visa accounts
Takes 24 hours to run

Design#2 VisaDB
For each Transaction in Queue
- For relevant records
  - Use 2-PL to acquire/release locks
  - Process record
  - WAL Logs for updates
- Commit or Abort

Replace with more sophisticated algorithms (cs245/cs345)
Design Theory

• Design theory is about how to represent your data to avoid anomalies.

• Simple algorithms for “best practices”
Relational Schema Design

High-level idea

1. Start with some relational schema
2. Find out its functional dependencies (FDs)
3. Use these to design a better schema
   One which minimizes the possibility of anomalies
A Picture Of FDs

Defn (again): Given attribute sets $A = \{A_1, \ldots, A_m\}$ and $B = \{B_1, \ldots, B_n\}$ in $R$, the functional dependency $A \rightarrow B$ on $R$ holds if for any $t_i, t_j$ in $R$:

- If $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2] = t_j[A_2]$ AND $\ldots$ AND $t_i[A_m] = t_j[A_m]$ then $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2] = t_j[B_2]$ AND $\ldots$ AND $t_i[B_n] = t_j[B_n]$.

<table>
<thead>
<tr>
<th></th>
<th>$A_1$</th>
<th>$\ldots$</th>
<th>$A_m$</th>
<th>$B_1$</th>
<th>$\ldots$</th>
<th>$B_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If $t_1, t_2$ agree here.. ...they also agree here!
Finding Functional Dependencies

Given a set of FDs, $F = \{f_1, \ldots f_n\}$, does an FD $g$ hold?

**Inference problem:** How do we decide?

Answer: Three simple rules called **Armstrong’s Rules.**

1. **Split/Combine**
2. **Reduction**
3. **Transitivity**
Finding Functional Dependencies

Example:

<table>
<thead>
<tr>
<th>Products</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Color</td>
<td>Category</td>
<td>Dep</td>
<td>Price</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Green</td>
<td>Gadget</td>
<td>Toys</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Widget</td>
<td>Black</td>
<td>Gadget</td>
<td>Toys</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Green</td>
<td>Whatsit</td>
<td>Garden</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

Provided FDs:
1. \{Name\} → \{Color\}
2. \{Category\} → \{Department\}
3. \{Color, Category\} → \{Price\}

Which / how many other FDs hold?
Finding Functional Dependencies

What's an algorithmic way to do this?

Example:

<table>
<thead>
<tr>
<th>Inferred FD</th>
<th>Rule used</th>
<th>Provided FDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. {Name, Category} -&gt; {Name}</td>
<td>Trivial</td>
<td></td>
</tr>
<tr>
<td>5. {Name, Category} -&gt; {Color}</td>
<td>Transitive (4 -&gt; 1)</td>
<td></td>
</tr>
<tr>
<td>6. {Name, Category} -&gt; {Category}</td>
<td>Trivial</td>
<td></td>
</tr>
<tr>
<td>7. {Name, Category} -&gt; {Color, Category}</td>
<td>Split/Combine (5 + 6)</td>
<td></td>
</tr>
<tr>
<td>8. {Name, Category} -&gt; {Price}</td>
<td>Transitive (7 -&gt; 3)</td>
<td></td>
</tr>
</tbody>
</table>

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}
Keys and Superkeys

A **superkey** is a set of attributes $A_1, \ldots, A_n$ s.t. for any other attribute $B$ in $R$, we have $\{A_1, \ldots, A_n\} \rightarrow B$

A **key** is a *minimal* superkey

I.e. all attributes are *functionally determined* by a superkey

Meaning that no subset of a key is also a superkey

**Superkey Algorithm:**
For each set of attributes $X$

1. Compute $X^+$
2. If $X^+ = \text{set of all attributes}$ then $X$ is a **superkey**
3. If $X$ is minimal, then it is a **key**
Conceptual Design

For a “mega” table

- Search for “bad” dependencies
- If any, keep decomposing (lossless) the table into sub-tables until no more bad dependencies
- When done, the database schema is normalized

Recall: there are several normal forms…
Course Summary

We’ll learn How To…

- **Query** over small-med-large data sets with **SQL**? [Weeks 1 and 2]
  - On relational engines, and “big data” engines

- **Scale** for “big queries”? On Clusters? [Weeks 3, 4, 5]
  - OLAP/Analytics, 1st principles of scale

- **Scale** for “big writes”? [Weeks 6, 7, 8]
  - Writes, Transactions, Logging, ACID properties

- **Design** “good” databases? [Weeks 9, 10]
  - Big Schemas, design, functional dependencies, query optimizers

**Project:** Query-Visualize-Learn on GB/TB scale data sets on a Cloud [sql + python]
Course Summary

CS 145
Final lecture: Review

CS 245/345
Systems scale
Serializability Txns
SQL/DFs on Spark

Self study
Play with real data
Pick a vertical to study data
THANK YOU!