Final Review

May 9, 2017
SQL
A Basic SQL Query

- **SELECT** \([\text{DISTINCT}]\) **target-list**
  - A list of attributes of relations in **relation-list**
- **FROM** **relation-list**
  - A list of relation names
    - (possibly with a range-variable after each name)
- **WHERE** **condition**
  - Comparisons ('=', '<>', '<', '>', '<=', '>=') and other boolean predicates,
    combined using AND, OR, and NOT
    (a boolean formula)
Integrity Constraints

- Domain Constraints
  - Limitations on valid values of a field.
- Key Constraints
  - A field(s) that must be unique for each row.
- Foreign Key Constraints
  - A field referencing a key of another relation.
  - Can also encode participation/1-many/many-1/1-1.
- Table Constraints
  - More general constraints based on queries.
Algorithms
Memory Conscious Algorithms

- Join
  - NLJ has a small working set (but is slow)
- GB Aggregate
  - Working Set $\sim$ # of Groups
- Sort
  - Working Set $\sim$ Size of Relation
Implementing: Joins

Solution 1 (Nested-Loop)

For Each (a in A) { For Each (b in B) { emit (a, b); }}
Implementing: Joins

Solution 2 (Block-Nested-Loop)

1) Partition into Blocks

2) NLJ on each pair of blocks
Implementing: Joins

Solution 3 (Index-Nested-Loop)

Like nested-loop, but use an index to make the inner loop much faster!
Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value. When you hit two that match, emit, then iterate both

A

1
2
3
5

B

1
4
5
6

Keep iterating on the set with the lowest value. When you hit two that match, emit, then iterate both A and B.
Implementing: Joins

Solution 5 (2-pass Hash)
1) Build a hash table on both relations
2) In-Memory Nested-Loop Join on each hash bucket
Implementing: Joins

Solution 6 (1-pass Hash)

Keep the hash table in memory

(A)

(B)
## Implementing: Joins

### Tradeoffs

<table>
<thead>
<tr>
<th>Method</th>
<th>Pipelined?</th>
<th>Memory Requirements?</th>
<th>Predicate Limitation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested Loop</td>
<td>1/2</td>
<td>1 Table</td>
<td>No</td>
</tr>
<tr>
<td>Block-Nested Loop</td>
<td>No</td>
<td>2 ‘Blocks’</td>
<td>No</td>
</tr>
<tr>
<td>Index-Nested Loop</td>
<td>1/2</td>
<td>1 Tuple (+Index)</td>
<td>Single Comparison</td>
</tr>
<tr>
<td>Sort-Merge</td>
<td>If Data Sorted</td>
<td>Same as reqs. of Sorting Inputs</td>
<td>Equality Only</td>
</tr>
<tr>
<td>2-pass Hash</td>
<td>No</td>
<td>Max of 1 Page per Bucket and All Pages in Any Bucket</td>
<td>Equality Only</td>
</tr>
<tr>
<td>1-pass Hash</td>
<td>1/2</td>
<td>Hash Table</td>
<td>Equality Only</td>
</tr>
</tbody>
</table>
Relational Algebra
RA Equivalencies

Selection
\[ \sigma_{c_1 \land c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R)) \] (Decomposable)
\[ \sigma_{c_1 \lor c_2}(R) \equiv \delta(\sigma_{c_1}(R) \cup \sigma_{c_2}(R)) \] (Decomposable)
\[ \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \] (Commutative)

Projection
\[ \pi_a(R) \equiv \pi_a(\pi_{a \cup b}(R)) \] (Idempotent)

Cross Product (and Join)
\[ R \times (S \times T) \equiv (R \times S) \times T \] (Associative)
\[ (R \times S) \equiv (S \times R) \] (Commutative)
Selection and Projection

\[ \pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R)) \]

Selection **commutes** with Projection
(but only if attribute set \( a \) and condition \( c \) are compatible)

\( a \) must include all columns referenced by \( c \)
Join

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$

Selection combines with Cross Product to form a Join as per the definition of Join (Note: This only helps if we have a join algorithm for conditions like $c$)
Selection and Cross Product

\[ \sigma_c(R \times S) \equiv (\sigma_c(R) \times S) \]

Selection commutes with Cross Product
(but only if condition c references attributes of R exclusively)
Projection and Cross Product

\[ \pi_a(R \times S) \equiv (\pi_{a_1}(R)) \times (\pi_{a_2}(S)) \]

Projection commutes (distributes) over Cross Product (where \( a_1 \) and \( a_2 \) are the attributes in \( a \) from \( R \) and \( S \) respectively)
RA Equivalencies

Union and Intersections are **Commutative** and **Associative**

Selection and Projection both commute with both Union and Intersection
## Relational Algebra

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$\sigma$</td>
<td>Select a subset of the input rows</td>
</tr>
<tr>
<td>Projection</td>
<td>$\pi$</td>
<td>Delete unwanted columns</td>
</tr>
<tr>
<td>Cross-product</td>
<td>$\times$</td>
<td>Combine two relations</td>
</tr>
<tr>
<td>Set-difference</td>
<td>$-$</td>
<td>Tuples in Rel 1, but not Rel 2</td>
</tr>
<tr>
<td>Union</td>
<td>$U$</td>
<td>Tuples either in Rel 1 or in Rel 2</td>
</tr>
</tbody>
</table>

**Also:** Intersection, **Join**, Division, Renaming (Not essential, but very useful)
SQL to RA

\[
\begin{align*}
\text{SELECT} & \quad \text{[DISTINCT]} \\
& \quad \text{target} \\
\text{FROM} & \quad \text{source} \\
\text{WHERE} & \quad \text{cond1} \\
\text{GROUP BY} & \quad \ldots \\
\text{HAVING} & \quad \text{cond2} \\
\text{ORDER BY} & \quad \text{order} \\
\text{LIMIT} & \quad \text{lim} \\
\text{UNION} & \quad \text{nextselect}
\end{align*}
\]
Transactions
What does it mean for a database operation to be correct?
What could go wrong?

Reading uncommitted data  
(write-read/WR conflicts; aka “Dirty Reads”)

T1: R(A), W(A),  
T2: R(A), W(A), CMT,  
T1: R(B), W(B), ABRT 
T2: R(A), W(A), CMT,

Unrepeatable Reads  
(read-write/RW conflicts)

T1: R(A),  
T2: R(A), W(A), CMT 
T2: R(A), W(A), CMT,
What could go wrong?

Overwriting Uncommitted Data
(write-write/WW conflicts)

T1: $W(A), W(B), CMT$
T2: $W(A), W(B), CMT$
Schedule
An ordering of read and write operations.

Serial Schedule
No interleaving between transactions at all

Serializable Schedule
Guaranteed to produce equivalent output to a serial schedule
Conflict Equivalence

Possible Solution: Look at read/write, etc… conflicts!

Allow operations to be reordered as long as conflicts are ordered the same way

Conflict Equivalence: Can reorder one schedule into another without reordering conflicts.

Conflict Serializability: Conflict Equivalent to a serial schedule.
Conflict Serializability

• **Step 1:** Serial Schedules are Always Correct

• **Step 2:** Schedules with the same operations and the same conflict ordering are conflict-equivalent.

• **Step 3:** Schedules conflict-equivalent to an always correct schedule are also correct.

• … or conflict serializable
View Serializability

Possible Solution: Look at data flow!

View Equivalence: All reads read from the same writer
Final write in a batch comes from the same writer

View Serializability: Conflict Equivalent to a serial schedule.
Information Flow

Multiple Transactions

R(…)

R(…)

R(…)

[Image of a diagram showing information flow through multiple transactions]
View Serializability

• **Step 1:** Serial Schedules are *Always Correct*

• **Step 2:** Schedules with the same information flow are *view-equivalent*.

• **Step 3:** Schedules *view-equivalent* to an always correct schedule are also correct.

• … or *view serializable*
Enforcing Serializability

• Conflict Serializability:
  • Does locking enforce conflict serializability?

• View Serializability
  • Is view serializability stronger, weaker, or incomparable to conflict serializability?

• What do we need to enforce either fully?
How to detect conflict serializable schedule?

It is not conflict serializable because the precedence graph has a cycle. It cannot be strict 2PL because T₂ will have to unlock(B) at the very end and hence it will be impossible for T₁ to update(B).

Every non-serializable schedule cannot be 2PL or strict 2PL.

It is serializable because it has an acyclic graph and 2PL because locks can be assigned as follows (many similar solutions are possible):

- T₁: write(A), read(D), write(D)
- T₂: read(B)
- T₃: read(D), write(D)

It cannot be strict 2PL for the same reasons with the first schedule.
Not conflict serializable but view serializable

Satisfies 3 conditions of view serializability

Every view serializable schedule which is not conflict serializable has blind writes.
Two-Phase Locking

- Phase 1: Acquire (do not release) locks.
  - Typically happens as objects are needed.

- Phase 2: Release (do not acquire) locks.
  - Typically happens as part of commit.
Reader/Writer (S/X)

• When accessing a DB Entity…
  • Table, Row, Column, Cell, etc…

• Before reading: Acquire a Shared (S) lock.
  • Any number of transactions can hold S.

• Before writing: Acquire an Exclusive (X) lock.
  • If a transaction holds an X, no other transaction can hold an S or X.
New Lock Modes

Even within the same application, there may be a need for locks at multiple levels of granularity.

Database elements are organized in a hierarchy:

- relations
- blocks
- tuples

```
R1
  └── B1
      ├── t1
      └── t2
  │     └── B2
  │         └── t3
  └── B3
      └── B4
          └── t4
              └── t5
```

contained in
Hierarchical Locks

• Lock Objects Top-Down

• Before acquiring a lock on an object, an xact must have at least an intention lock on its parent!

• For example:

• To acquire a S on an object, an xact must have an IS, IX on the object’s parent (why not S, SIX, or X?)

• To acquire an X (or SIX) on an object, an xact must have a SIX, or IX on the object’s parent.
# New Lock Modes

<table>
<thead>
<tr>
<th>Lock Mode Desired</th>
<th>None</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>IS</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
</tr>
<tr>
<td>IX</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>S</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
<td>valid</td>
<td>fail</td>
</tr>
<tr>
<td>X</td>
<td>valid</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
</tr>
</tbody>
</table>
Serializability

- **Read**
- **Read**
- **Read**
- **Write**
- **Write**
- **Write**
- **Abort**
- **Commit**

Time
Optimistic CC

• **Read Phase**: Transaction executes on a private copy of all accessed objects.

• **Validate Phase**: Check for conflicts.

• **Write Phase**: Make the transaction’s changes to updated objects public.
(1) Transaction executes on a **private copy** of the DB (writes are buffered)

(2) Transaction checks for conflicts

(3) Buffered writes written to main Database

**COMMIT Called** (user ok with commit)

**COMMIT Returns** (Commit complete)
Read Phase

Read Set

Write Set

Read
Read
Read

Write
Write

Abort
Commit

Time
Read Phase

ReadSet(T<sub>i</sub>): Set of objects read by T<sub>i</sub>.

WriteSet(T<sub>i</sub>): Set of objects written by T<sub>i</sub>.
Validation Phase

Pick a serial order for the transactions
(e.g., assign id #s or timestamps)

When should we assign Transaction IDs? (Why?)
Validation Phase

What tests are needed?
Simple Test

For all $i$ and $k$ for which $i < k$, check that $T_i$ completes before $T_k$ begins.

Is this sufficient?  Is this efficient?
Test 2

For all i and k for which $i < k$, check that $T_i$ completes before $T_k$ begins its write phase AND $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_k)$ is empty

How do these two conditions help?
Test 3
For all $i$ and $k$ for which $i < k$, check that $T_i$ completes its read phase first
AND $\text{WriteSet}(T_i) \cap \text{ReadSet}(T_k)$ is empty
AND $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_k)$ is empty

How do these three conditions help?
• Give each object a read timestamp (RTS) and a write timestamp (WTS)

• Give each transaction a timestamp (TS) at the start.

• Use RTS/WTS to track previous operations on the object.

• Compare with TS to ensure ordering is preserved.
Timestamp CC

- When $T_i$ reads from object $O$:
  - If $WTS(O) > TS(T_i)$, $T_i$ is reading from a ‘later’ version.
    - Abort $T_i$ and restart with a new timestamp.
  - If $WTS(O) < TS(T_i)$, $T_i$’s read is safe.
    - Set $RTS(O)$ to $\text{MAX}( RTS(O), TS(T_i) )$
When $T_i$ writes to object $O$:

- If $RTS(O) > TS(T_i)$, $T_i$ would cause a dirty read.
  - Abort $T_i$ and restart it.
- If $WTS(O) > TS(T_i)$, $T_i$ would overwrite a ‘later’ value.
  - Don’t need to restart, just ignore the write.
- Otherwise, allow the write and update $WTS(O)$. 
Logging
Write-Ahead Logging

Before writing to the database, first write what you plan to write to a log file...

Log

\[ W(A:10) \]
Write-Ahead Logging

Once the log is safely on disk you can write the database

\[ \text{Log} \]
\[ W(A:10) \]
Write-Ahead Logging

Log is append-only, so writes are always efficient

Log:
\[ W(A:10) \]
\[ W(C:8) \]
\[ W(E:9) \]
Write-Ahead Logging

...allowing random writes to be safely batched

Log
W(A:10)
W(C:8)
W(E:9)
UNDO Logging

Store both the “old” and the “new” values of the record being replaced.

Log

\[ W(A:8\rightarrow10) \]
\[ W(C:5\rightarrow8) \]
\[ W(E:16\rightarrow9) \]
UNDO Logging

Active Xacts
Xact:1, Log: 45
Xact:2, Log: 32

Log
43: w(A: 8→10)
44: w(C: 5→8)
45: w(E: 16→9)

Image copyright: OpenClipart (rg1024)
UNDO Logging

Active Xacts
- Xact 1, Log: 45
- Xact 2, Log: 32

Log
- 43: W(A: 8\rightarrow 10)
- 44: W(C: 5\rightarrow 8)
- 45: W(E: 16\rightarrow 9)

Xact: ABORT
UNDO Logging

Active Xacts

Xact 1, Log: 45

Xact 2, Log: 32

Log

43: W(A: 8 → 10)
44: W(C: 5 → 8)
45: W(E: 16 → 9)
UNDO Logging

Active Xacts
- Xact 1, Log: 45
- Xact 2, Log: 32

Log
- 43: W(A: 8→10)
- 44: W(C: 5→8)
- 45: W(E: 16→9)

Image copyright: OpenClipart (rg1024)
UNDO Logging

Active Xacts

Xact:1, Log: 45
Xact:2, Log: 32

Log

43: \text{W}(A:8\rightarrow10)
44: \text{W}(C:5\rightarrow8)
45: \text{W}(E:16\rightarrow9)

Image copyright: OpenClipart (rg1024)
• **Isolation**: Already addressed.
• **Atomicity**: Need writes to get *flushed* in a single step.
  • IOs are only atomic at the page level.

• **Durability**: Need to *buffer* some writes until commit.
  • May need to free up memory for another xact.

• **Consistency**: Need to roll back incomplete xacts.
  • May have already paged back to disk.
Atomicity

- **Problem**: IOs are only atomic for 1 page.
  - What if we crash in between writes?
- **Solution**: Logging (e.g., Journaling Filesystem)
  - Log everything first before you do it.

Time

- Append changes to log
- Overwrite file blocks
Durability / Consistency

• **Problem**: Buffer memory is limited
  • What if we need to ‘page out’ some data?

• **Solution**: Use log (or similar) to recover buffer
  • *Problem*: Commits more expensive

• **Solution**: Modify DB in place, use log to ‘undo’ on abort
  • *Problem*: Aborts more expensive
Anatomy of a log entry

Last entry for this Xact (forms a Linked List)

What was written, where, prior value, etc…

Which Xact Triggered This Entry

Write, Commit, etc…

Entry Metadata
## Transaction Table

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Status</th>
<th>Last Log Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction 24</td>
<td>VALIDATING</td>
<td>99</td>
</tr>
<tr>
<td>Transaction 38</td>
<td>COMMITTING</td>
<td>85</td>
</tr>
<tr>
<td>Transaction 42</td>
<td>ABORTING</td>
<td>87</td>
</tr>
<tr>
<td>Transaction 56</td>
<td>ACTIVE</td>
<td>100</td>
</tr>
</tbody>
</table>
## Buffer Manager

<table>
<thead>
<tr>
<th>Page</th>
<th>Status</th>
<th>Last Log Entry</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>DIRTY</td>
<td>47</td>
<td>01011010…</td>
</tr>
<tr>
<td>30</td>
<td>CLEAN</td>
<td>n/a</td>
<td>11001101…</td>
</tr>
<tr>
<td>52</td>
<td>DIRTY</td>
<td>107</td>
<td>10100010…</td>
</tr>
<tr>
<td>57</td>
<td>DIRTY</td>
<td>87</td>
<td>01001101…</td>
</tr>
<tr>
<td>66</td>
<td>CLEAN</td>
<td>n/a</td>
<td>01001011…</td>
</tr>
</tbody>
</table>
Transaction Table

**Step 1:** Recover Xact State

- **Problem:** We might need to scan to the very beginning of the log to recover the full state of the Xact table (& Buffer Manager)

- **Solution:** Periodically save (checkpoint) the Xact table to the log.
  
  - Only need to scan the log up to the last (successful) checkpoint.
Checkpointing

• `begin_checkpoint` record indicates when the checkpoint began.

• Checkpoint covers all log entries before this entry.

• `end_checkpoint` record contains the current transaction table and the dirty page table.

• Signifies that the checkpoint is now stable.
Buffer Manager

**Step 2:** Recover Buffered Data

• Where do we get the buffered data from?
Consistency

**Step 3**: Undo incomplete xacts

- Record *previous values* with log entries
- Replay log in reverse (linked list of entries)
  - Which Xacts do we undo?
  - Which log entries do we undo?
  - How far in the log do we need to go?
Compensation Log Records

- **Problem**: Step 3 is expensive!
  - What if we crash during step 3?
- **Optimization**: Log undos as writes as they are performed (CLR).s.
  - Less repeat computation if we crash during recovery
  - Shifts effort to step 2 (replay)
  - CLR.s don’t need to be undone!
ARIES Crash Recovery

- Start from checkpoint stored in master record.

- **Analysis**: Rebuild the Xact Table

- **Redo**: Replay operations from all live Xacts (even uncommitted ones).

- **Undo**: Revert operations from all uncommitted/aborted Xacts.

- Oldest log record of transaction active at crash

- Smallest recLSN in dirty page table after Analysis

- Last Checkpoint

- CRASH

- A R U
Materialized Views
Materialized Views

When the base data changes, the view needs to be updated
View Maintenance

\[ \text{VIEW} \leftarrow Q(D) \]
View Maintenance

\[
\text{WHEN } D \leftarrow D + \Delta D \text{ DO:}
\]

\[
\text{VIEW } \leftarrow Q(D + \Delta D)
\]

Re-evaluating the query from scratch is expensive!
View Maintenance

WHEN D ← D+ΔD DO:
VIEW ← VIEW+ΔQ(D,ΔD)

(ideally) Smaller & Faster Query

(ideally) Fast “merge” operation.
Delta Queries

$$\Delta(\sigma(R))$$

$$\sigma$$

$$\sigma$$

R

R

\(\Delta R\)

Original R

Inserted Tuples of R

Does this work for deleted tuples?
Delta Queries

$$\Delta(\pi(R)) = \pi(\Delta R)$$

Does this work (completely) under set semantics?
Delta Queries

\[ \Delta(R_1 \cup R_2) \]
Delta Queries

\[ X \quad R \quad S \quad R \quad \Delta R \quad S \]
Delta Queries

\[ R : \{ 1, 2, 3 \} \quad S : \{ 5, 6 \} \]

\[ R \times S = \{ <1,5>, <1, 6>, <2,5>, <2,6>, <3,5>, <3,6> \} \]

\[ \Delta R_{\text{inserted}} = \{ 4 \} \]

\[ \Delta R_{\text{deleted}} = \{ 3, 2 \} \]

\[ (R+\Delta R) \times S = \{ <1,5>, <1, 6>, <4,5>, <4,6> \} \]

\[ \Delta_{\text{inserted}} (R \times S) = \Delta R_{\text{inserted}} \times S \]

\[ \Delta_{\text{deleted}} (R \times S) = \Delta R_{\text{deleted}} \times S \]

What if \( R \) and \( S \) both change?
Delta Queries

$$(R_1 \cup \Delta R_1) \times (R_2 \cup \Delta R_2)$$

$$(R_1 \times R_2) \cup (R_1 \times \Delta R_2) \cup (\Delta R_1 \times R_2) \cup (\Delta R_1 \times \Delta R_2)$$

The original query

The delta query