Pre-Midterm Material

Keys

- Keys are a way to associate tuples in different relations

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
Primary Keys

- **A set of fields is a superkey if:**
  - No two distinct tuples can have same values in all key fields

- **A set of fields is a key for a relation if:**
  - It is a superkey
  - No subset of the fields is a superkey

- **What if >1 key for a relation?**
  - One of the keys is chosen (by DBA) to be the primary key.
  - Other keys are called candidate keys.

Foreign Keys, Referential Integrity

- **Foreign key:** Set of fields in one relation that is used to `refer` to a tuple in another relation.
  - Must correspond to the primary key of the other relation.
  - Like a `logical pointer`.

- **If all foreign key constraints are enforced,** referential integrity is achieved (i.e., no dangling references.)
Basic SQL Query

- **relation-list**: List of relation names
  - possibly with a range variable after each name
- **target-list**: List of attributes of tables in relation-list
- **qualification**: Comparisons combined using AND, OR and NOT.
- **DISTINCT**: optional keyword indicating that the answer should not contain duplicates.

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

Query Semantics

1. FROM: compute cross product of tables.
2. WHERE: Check conditions, discard tuples that fail.
3. SELECT: Delete unwanted fields.
4. DISTINCT (optional): eliminate duplicate rows.

**Note:** Probably the least efficient way to compute a query!
- Query optimizer will find more efficient ways to get the same answer.
Queries With GROUP BY

• The target-list contains
  – (i) list of column names &
  – (ii) terms with aggregate operations (e.g., \( \text{MIN} \) (S.age)).
• List of column names (i) can contain only attributes from the grouping-list.

```
SELECT [DISTINCT] target-list
FROM relation-list
[WHERE qualification]
GROUP BY grouping-list
```

Conceptual Evaluation

• The cross-product of relation-list is computed, tuples that fail qualification are discarded, `unnecessary` fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
• One answer tuple is generated per qualifying group.
• If DISTINCT is specified: drop duplicate answer tuples.

```
SELECT [DISTINCT] target-list
FROM relation-list
[WHERE qualification]
GROUP BY grouping-list
```
Queries With GROUP BY and HAVING

- Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set.

```sql
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

**Formal Relational Query Languages**

Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:

- **Relational Algebra**: More operational, very useful for representing execution plans.
- **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-procedural, *declarative*.)

📌 Understanding Algebra & Calculus is key to understanding SQL, query processing!
Relational Algebra: 5 Basic Operations

- **Selection** (σ) Selects a subset of rows from relation (horizontal).
- **Projection** (π) Retains only wanted columns from relation (vertical).
- **Cross-product** (×) Allows us to combine two relations.
- **Set-difference** (−) Tuples in r1, but not in r2.
- **Union** (∪) Tuples in r1 or in r2.

Since each operation returns a relation, operations can be composed! (Algebra is “closed”.)

Relational Calculus

- **Query** has the form: \( \{ T \mid p(T) \} \)
  - \( p(T) \) is a formula containing \( T \)

- **Answer** = tuples \( T \) for which \( p(T) = true \).
**Formulae**

- **Atomic formulae:**
  - $R \in \text{Relation}$
  - $R.a \ op S.b$
  - $R.a \ op \text{constant}$
  - $\ldots \ op \text{ is one of } <,>,=,\leq,\geq,\neq$
- **A formula can be:**
  - an atomic formula
  - $\neg p, p \land q, p \lor q, p \Rightarrow q$
  - $\exists R(p(R))$
  - $\forall R(p(R))$

**Free and Bound Variables**

- **Quantifiers:** $\exists$ and $\forall$
- **Use of $\exists X$ or $\forall X$ binds $X$.**
  - A variable that is not bound is free.
- **Recall our definition of a query:**
  - $\{ T \mid \rho(\tau) \}$
- **Important restriction:**
  - $T$ must be the only free variable in $\rho(\tau)$.
  - all other variables must be bound using a quantifier.
Indexes

Know how to insert items into B+-trees and how to search them.

Remember how to do prefix-key compression.

Understand fanout’s connection to tree height and search speed.

2-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => the number of passes
  \[ = \lceil \log_2 N \rceil + 1 \]
- So total cost is:
  \[ 2N \left( \lceil \log_2 N \rceil + 1 \right) \]

Idea: Divide and conquer: sort subfiles and merge
External Merge Sort

• To sort a file with \( N \) pages using \( B \) buffer pages:
  – Pass 0: use \( B \) buffer pages. Produce \( \lceil N/B \rceil \) sorted runs of \( B \) pages each.
  – Pass 1, 2, ..., etc.: merge \( B-1 \) runs.

\[
\text{Cost} = 2N \times (\text{# of passes}) = 2N \times \left[ 1 + \log_{B-1} \left( \frac{N}{B} \right) \right]
\]

Iterator Model

• Relational operators are all subclasses of the class `iterator`:

  ```
  class iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[];
    // additional state goes here
  }
  ```

• Note:
  – Edges in the graph are specified by inputs (max 2, usually)
  – Any iterator can be input to any other!
**Simple Nested Loops Join**

\[ R \bowtie S: \text{ foreach tuple } r \text{ in } R \text{ do} \]
\[ \quad \text{ foreach tuple } s \text{ in } S \text{ do} \]
\[ \quad \quad \text{ if } r_i == s_j \text{ then add } <r, s> \text{ to result} \]

Cost = \( (p_R * |R|)*|S| + |R| \)

---

**Page-Oriented Nested Loops Join**

\[ R \bowtie S: \text{ foreach page } b_R \text{ in } R \text{ do} \]
\[ \quad \text{ foreach page } b_S \text{ in } S \text{ do} \]
\[ \quad \text{ foreach tuple } r \text{ in } b_R \text{ do} \]
\[ \quad \quad \text{ foreach tuple } s \text{ in } b_S \text{ do} \]
\[ \quad \quad \quad \text{ if } r_i == s_j \text{ then add } <r, s> \text{ to result} \]

Cost = \( |R|*|S| + |R| \)
Block Nested Loops Join

- Page-oriented NL doesn’t exploit extra buffers :(  
- Idea to use memory efficiently:

Cost: Scan outer + (#outer blocks * scan inner)  
#outer blocks = \lceil \# of pages of outer / blocksize \rceil

Index Nested Loops Join

R \bowtie S:  
foreach tuple r in R do  
    foreach tuple s in S where r_i == s_j do  
        add <r, s> to result

Cost = |R| + (|R| * p_R) * cost to find matching S tuples

- If index uses Alt. 1, cost = cost to traverse tree from root to leaf.  
- For Alt. 2 or 3:  
  1. Cost to lookup RID(s); typically 2-4 IO’s for B+Tree.  
  2. Cost to retrieve records from RID(s); depends on clustering.  
     - Clustered index: 1 I/O per page of matching S tuples.  
     - Unclustered: up to 1 I/O per matching S tuple.
**Sort-Merge Join**

1. Sort R on join attr(s)
2. Sort S on join attr(s)
3. Scan sorted-R and sorted-S in tandem, to find matches

- **Cost:** \( \text{Sort R} + \text{Sort S} + (|R| + |S|) \)
  - But in worst case, last term could be \(|R| \times |S|\)

You could do the join during the final merging pass of sort!

1. Read R and write out sorted runs
2. Read S and write out sorted runs
3. Merge R-runs and S-runs, and find \(R \times S\) matches

\[ \text{Cost} = 3 \times |R| + 3 \times |S| \]

**Hash Join**

![Diagram of Hash Join](image)

- Original Relation
- Hash table for partition \(R_i\) (B-2 pages)
- Join Result
- Hash function \(h\)
- Input buffer for \(S_i\)
- Output buffer

B main memory buffers

Disk

Partitions

1 2 B-1
Cost of Hash Join

- **Partitioning phase:** read+write both relations
  \[ \Rightarrow 2(|R|+|S|) \text{ I/Os} \]
- **Matching phase:** read+write both relations
  \[ \Rightarrow |R|+|S| \text{ I/Os} \]
- **Total cost of 2-pass hash join** = \( 3(|R|+|S|) \)

Hybrid Hashing

- **Idea:** keep one of the hash buckets in memory!

Q: how do we choose the value of \( k \)?
Summary: Hashing vs. Sorting

- **Sorting pros:**
  - Good if input already sorted, or need output sorted
  - Not sensitive to data skew or bad hash functions

- **Hashing pros:**
  - Often cheaper due to hybrid hashing
  - For join: # passes depends on size of smaller relation
  - Highly parallelizable

Cost-based Query Sub-System

<table>
<thead>
<tr>
<th>Queries</th>
<th>Usually there is a heuristics-based rewriting step before the cost-based steps.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select * From Blah B Where B.blah = blah</td>
<td></td>
</tr>
</tbody>
</table>

- Query Parser
- Query Optimizer
- Plan Generator
- Plan Cost Estimator
- Catalog Manager
- Schema
- Statistics
- Query Executor

Sometimes there is a heuristics-based rewriting step before the cost-based steps.
Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of **query blocks**, and these are optimized one block at a time.

- For each block, the plans considered are:
  - All available access methods, for each relation in `FROM` clause.
  - All left-deep join trees (i.e., right branch always a base table, consider all join orders and join methods.)

### Cost Estimation

- **For each plan considered, must estimate total cost:**
  - Must estimate **cost** of each operation in plan tree.
    - Depends on input cardinalities.
    - We’ve already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must estimate **size of result** for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
  - In System R, cost is boiled down to a single number consisting of \#I/O + \textit{factor} * \#CPU instructions
Size Estimation and Reduction Factors

- Consider a query block:
- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size. Result cardinality = Max # tuples * product of all RF’s.
- RF usually called "selectivity"

Enumerating Plans

- Different plans are generated by:
  - Varying the join orders
  - Moving selections/projections down the tree
  - Varying the different algorithms for each operator

- At each level of the tree...
  - Find the cheapest way to compute the subtree
  - Also consider any “interesting orders”