Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- Other operators
- Quiz 8
- Query *optimization*....

Quiz 8
**Index-ANDing**

**Q5**
3 Points

Consider a query with a conjunctive predicate:

```sql
select * from R where a = 10 and b = 20.
```

- R occupies 1 million blocks on disk, and
- there are secondary indexes of height 4 on both R.a and R.b.
- Assume number of tuples in R with R.a = 10 is 1000, with R.b = 20 is 3000, and with both R.a = 10 & R.b = 20 is 200.

For all the indexes, assume the number of pointers in each leaf (to the actual records) is 500, and number of records of R per block is 100.

**Q5.1**
1 Point

How many blocks are transferred when using the index on R.a to fetch tuples matching R.a = 10, and then checking the condition in memory.

- **Explaination**: 4 for tree (including leaf), 1 for extra leaf (1000 ptrs needed, 500 per leaf), 1000 for tuple blocks

**Q5.2**
1 Point

The same as the above but using the index on R.b:

- **Explaination**: 4 for tree, 5 for extra leaves, 3000 for tuple blocks
**Index-ANDing**

**Q5**  
3 Points  
Consider a query with a conjunctive predicate:

```
select * from R where a = 10 and b = 20.
```

- R occupies 1 million blocks on disk, and
- there are secondary indexes of height 4 on both R.a and R.b.
- Assume number of tuples in R with R.a = 10 is 1000, with R.b = 20 is 3000, and with both R.a = 10 & R.b = 20 is 200.

For all the indexes, assume the number of pointers in each leaf (to the actual records) is 500, and number of records of R per block is 100.

**Q5.3**  
1 Point  
Same as above, but instead using “index-anding” to identify tuples who reading them, and then reading only those that match the entire predicate:

EXPLANATION
4 for tree, 1 extra leaf for a, 4 for tree, 5 extra leaves for b, 200 tuple blocks

**Index-ORing?**

**Q6**  
0.2 Points  
Consider a query with a disjunctive predicate:

```
select * from R where a = 10 or b = 20
```

- R occupies 1 million blocks on disk
- secondary indexes of height 4 on both R.a and R.b
- 1000 tuples match R.a = 10, 1500 match R.b = 20, 2000 tuples match the entire predicate.

What are the total number of disk I/Os (i.e., # blocks transferred) required to identify all matching tuples?

EXPLANATION
4 for tree and 1 extra leaf for a, 4 for tree and 2 extra leaves for b = 11

For all the indexes, assume the number of pointers on the leaf level (to the actual records) is 500 per block, and number of records of R per block is 100.
Index-ORing?

Recap: Query Processing

- Many, many ways to implement the relational operations
  - Numerous more used in practice
  - Especially in data warehouses which handles TBs (even PBs) of data
- However, SQL is complex, and you can do much with it
  - Compared to that, this isn’t much
- Most of it is very nicely modular
  - Especially through use of the iterator() interface
  - Can plug in new operators quite easily
  - PostgreSQL query processing codebase very easy to read and modify
- Having many operators does complicate the query optimizer
  - But needed for performance
Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies

Query Optimization

- Overview
- Statistics Estimation
- Transformation of Relational Expressions
- Optimization Algorithms
Query Optimization

- Why?
  - Many different ways of executing a given query
  - Huge differences in cost
- Example:
  - `select * from person where ssn = "123"`
  - Size of person = 1GB
  - Sequential Scan:
    - Takes 1GB / (20MB/s) = 50s
  - Use an index on SSN (assuming one exists):
    - Approx 4 Random I/Os = 16ms

Query Optimization

- Many choices
  - Using indexes or not, which join method (NL, hash, merge...)
  - What join order?
    - Given a join query on R, S, T, should I join R with S first, or S with T first?
- This is an optimization problem
  - Similar to say traveling salesman problem
  - Number of different choices is very very large
  - Step 1: Figuring out the solution space
  - Step 2: Finding algorithms/heuristics to search through the solution space
Query Optimization

- Equivalent relational expressions
  - Drawn as a tree
  - List the operations and the order
    - note that the select operator has moved

(a) Initial expression tree    (b) Transformed expression tree

Query Optimization

- Execution plans
  - Evaluation expressions annotated with the methods used
Query Optimization

- **Steps:**
  - Generate all possible execution plans for the query
  - Figure out the cost for each of them
  - Choose the best

- **Not done exactly as listed above**
  - Too many different execution plans for that
  - Typically interleave all of these into a single efficient search algorithm

---

Query Optimization

- **Steps (detail):**
  - Generate all possible execution plans for the query
    - First generate all equivalent expressions
    - Then consider all annotations for the operations
  - Figure out the cost for each of them
    - Compute cost for each operation
      - Using the formulas discussed before
      - One problem: How do we know the number of result tuples for, say,
    - Count them! Better yet, estimate…
  - Choose the best
Query Optimization

- Introduction
- Transformation of Relational Expressions
- Statistics Estimation
- Optimization Algorithms

Equivalence of Expressions

- Two relational expressions equivalent iff:
  - Their result is identical on all legal databases
- Equivalence rules (Section 13.2.1):
  - Allow replacing one expression with another
- Examples:

1. \( \sigma_{\theta_1 \land \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E)) \)

2. Selections are commutative

\[ \sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E)) \]
Equivalence Rules

- Examples:

3. \( \Pi_{L_1}(\Pi_{L_2}(E \join \Pi_{L_3}(E))) = \Pi_{L_1}(E) \)

5. \( E_1 \theta E_2 = E_2 \theta E_1 \)

7(a). If \( \theta_0 \) only involves attributes from \( E_1 \):

\[ \sigma_{\theta_0}(E_1 \theta E_2) = (\sigma_{\theta_0}(E_1)) \theta E_2 \]

- And so on…
  - Many rules of this type

Pictorial Depiction

Assuming projection on output

Natural joins associative

If \( \theta \) only has attributes from \( E_1 \)
Example

- Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.

\[ \Pi_{\text{customer\_name}} \left( \sigma_{\text{branch\_city} = \text{"Brooklyn" \land balance > 1000}} \right) \]

\( (\text{branch} \bowtie (\text{account} \bowtie \text{depositor})) \)

- Apply the rules one by one

\[ \Pi_{\text{customer\_name}} \left( \left( \sigma_{\text{branch\_city} = \text{"Brooklyn" \land balance > 1000}} \right) \right) \]

\( (\text{branch} \bowtie \text{account}) \bowtie \text{depositor} \)

inner joins associative

\[ \Pi_{\text{customer\_name}} \left( \left( \left( \sigma_{\text{branch\_city} = \text{"Brooklyn"}} \right) \bowtie \left( \sigma_{\text{balance > 1000}} \right) \right) \right) \]

account)

first predicate on branch

second predicate on account

Example
Equivalence of Expressions

- The rules give us a way to enumerate all equivalent expressions
  - Note that the expressions don't contain physical access methods, join methods etc…
- Simple Algorithm:
  - Start with the original expression
  - Apply all possible applicable rules to get a new set of expressions
  - Repeat with this new set of expressions
  - Till no new expressions are generated

Equivalence of Expressions

- Works, but is not feasible
- Consider a simple case:
  - \( R1 \ \otimes \ (R2 \ \otimes \ (R3 \ \otimes \ (\ldots \ \otimes Rn))\ldots) \)

- Just join commutativity and associativity will give us:
  - At least:
    - \( n^2 \ast 2^n \)
  - At worst:
    - \( n! \ast 2^n \)
  - Typically enumeration combined with a search process
Evaluation Plans

- We still need to choose the join methods etc..
  - Option 1: Choose for each operation separately
    - Usually okay, but sometimes the operators interact
    - Consider joining three relations on the same attribute:
      - \( R_1 \Join_a (R_2 \Join_a R_3) \)
    - Best option for \( R_2 \) join \( R_3 \) might be hash-join
      - But if \( R_1 \) is sorted on \( a \), then sort-merge join is preferable
      - Because it produces the result in sorted order by \( a \)
- Also, pipelining or materialization
- Such issues typically arise when doing the optimization

Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation
Optimization Algorithms

- Two types:
  - Exhaustive: attempt to find the best plan
  - Heuristic: simpler, but not guaranteed to find the optimal plan

- Consider a simple case
  - Join of the relations $R1, \ldots, Rn$
  - No selections, no projections
  - Still very large plan space

Searching for the best plan

- Option 1:
  - Enumerate all equivalent expressions for the original query
    - Using the rules outlined earlier
  - Estimate cost for each and choose the lowest

- Too expensive!
  - Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
  - There are $(2(n – 1))!/(n – 1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
Searching for the best plan

- **Option 2:**
  - Dynamic programming
    - There is much commonality between the plans
    - Costs are additive
      - Caveat: Sort orders (also called “interesting orders”)
  - Reduces costs to $O(n^3)$ or $O(n^2)$ in most cases
    - Interesting orders increase this a little bit
  - Considered acceptable
    - Typically $n < 10$.
  - Switch to heuristic if not acceptable
Left Deep Join Trees

- In left-deep join trees, the right-hand-side input for each join is a relation, not the result of an intermediate join.
- Early systems only considered these types of plans:
  - Easier to pipeline.

Heuristic Optimization

- Dynamic programming is expensive.
- Use heuristics to reduce the number of choices.
- Typically rule-based:
  - Perform selection early (reduces the number of tuples).
  - Perform projection early (reduces the number of attributes).
  - Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.
**Steps in Typical Heuristic Optimization**

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining).

**Query Optimization**

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation
Cost estimation

- Computing operator costs requires information like:
  - Primary key?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks?
  - RAID?? Which one?
    - Read/write costs are quite different
  - How many tuples match a predicate like “age > 40”?  
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
    - E.g. (R JOIN S) is input to another join operation – need to know if it fits in memory
  - And so on…

Cost estimation

- Some info is static and maintained in the metadata
  - Primary key?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks?
  - RAID?? Which one?
    - Read/write costs are quite different

- Typically kept in some tables in the database
  - “all_tab_columns” in Oracle
- Most systems have commands for updating them
Cost estimation

- Others need to be estimated:
  - How many tuples match a predicate like “age > 40”? 
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
- The problem variously called:
  - “intermediate result size estimation”
  - “selectivity estimation”

- Very important to estimate reasonably well
  - E.g. consider “SELECT * FROM R WHERE zipcode = 20742”
  - We estimate that there are 10 matches, and choose to use a secondary index
    (remember: random I/Os)
  - Turns out there are 10,000 matches
  - Using a secondary index very bad idea
  - Optimizer also often choose Nested-loop joins if one relation very small…
    underestimation can be very bad

Selectivity Estimation

- Basic idea:
  - Maintain some information about the tables
    - More information \( \rightarrow \) more accurate estimation
    - More information \( \rightarrow \) higher storage cost, higher update cost
    - Make uniformity and randomness assumptions to fill in the gaps

- Example:
  - For a relation “people”, we keep:
    - Total number of tuples = 100,000
    - Distinct “zipcode” values that appear in it = 100
  - Given a query: “zipcode = 20742”
    - We estimated the number of matching tuples as: 100,000/100 = 1000
  - What if I wanted more accurate information?
    - Keep histograms...
Histograms

- A condensed, approximate version of the “frequency distribution”
  - Divide the range of the attribute value in “buckets”
  - For each bucket, keep the total count
  - Assume uniformity within a bucket

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000-20199</td>
<td>45000</td>
</tr>
<tr>
<td>20200-20399</td>
<td></td>
</tr>
<tr>
<td>20400-20599</td>
<td></td>
</tr>
<tr>
<td>20600-20799</td>
<td>45000</td>
</tr>
<tr>
<td>20800-20999</td>
<td></td>
</tr>
</tbody>
</table>

- Given a query: zipcode = “20742”
  - Find the bucket (Number 3)
  - Say the associated count = 45000
  - Assume uniform distribution within the bucket: 45,000/200 = 225
Histograms

- What if the ranges are typically not full?
  - i.e., only a few of the zipcodes are actually in use?
- With each bucket, also keep the number of zipcodes that are valid
- Now the estimate would be: 45,000/80 = 562.50
- More Information ➔ Better estimation

![Histogram Chart]

Historegrams

- Very widely used in practice
  - One-dimensional histograms kept on almost all columns of interest
    - i.e., the columns that are commonly referenced in queries
  - Sometimes: multi-dimensional histograms also make sense
    - Less commonly used as of now
- Two common types of histograms:
  - Equi-depth
    - The attribute value range partitioned such that each bucket contains about the same number of values
  - Equi-width
    - The attribute value range partitioned in equal-sized buckets
  - others…
Next…

- Estimating sizes of the results of various operations
- Guiding principle:
  - Use all the information available
  - Make uniformity and randomness assumptions otherwise
  - Many formulas, but not very complicated…
    - In most cases, the first thing you think of!

Basic statistics

- Basic information stored for all relations
  - \( n_r \): number of tuples in a relation \( r \).
  - \( b_r \): number of blocks containing tuples of \( r \).
  - \( l_r \): size of a tuple of \( r \).
  - \( f_r \): blocking factor of \( r \) — i.e., the number of tuples of \( r \) that fit into one block.
  - \( V(A, r) \): number of distinct values that appear in \( r \) for attribute \( A \); same as the size of \( \prod_A(r) \).
  - \( MAX(A, r) \): th maximum value of \( A \) that appears in \( r \)
  - \( MIN(A, r) \)
  - If tuples of \( r \) are stored together physically in a file, then:

\[
    b_r = \left\lceil \frac{n_r}{f_r} \right\rceil
\]
Selection Size Estimation

- $\sigma_{A=X}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - equality condition on a key attribute: size estimate = 1

- $\sigma_{A\geq v}(r)$ (case of $\sigma_{A\geq v}(r)$ is symmetric)
  - Let $c$ denote the estimated number of tuples satisfying the condition.
  - If $\min(A,r)$ and $\max(A,r)$ are available in catalog
    - $c = 0$ if $v < \min(A,r)$
    - $c = n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$ if $\min(A,r) \leq v \leq \max(A,r)$
    - $c = n_r$ otherwise
  - If histograms available, can refine above estimate
  - In absence of statistical information $c$ is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- selectivity($\theta_i$) = the probability that a particular tuple in $r$ satisfies $\theta_i$.
  - If $s_i$ is the number of satisfying tuples in $r$, then selectivity ($\theta_i$) = $s_i / n_r$.

- conjunction: $\sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r)$. Assuming independence, estimate of tuples in the result is:
  $n_r \cdot \frac{S_1 \cdot S_2 \cdot \ldots \cdot S_n}{n_r^n}$

- disjunction: $\sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r)$. Estimated number of tuples:
  $n_r \cdot \left(1 - \left(1 - \frac{S_1}{n_r}\right) \cdot \left(1 - \frac{S_2}{n_r}\right) \cdot \ldots \cdot \left(1 - \frac{S_n}{n_r}\right)\right)$

- negation: $\sigma_{\bar{\theta}}(r)$. Estimated number of tuples: $n_r - \text{size}(\sigma_{\theta}(r))$
Estimating Output Sizes: Joins

- **R JOIN S: R.a = S.a**
  - |R| = 10,000; |S| = 5000

- **CASE 1: a is key for S**
  - Worst case: each tuple of R joins with exactly one tuple of S
  - So: |R JOIN S| = |R| = 10,000

- **CASE 2: a is key for R**
  - Each S tuple can match w/ only a single R tuple.
  - So: |R JOIN S| = |S| = 5,000

Equi-joins simplify things.

- **CASE 3: a is not a key for either**
  - Reason with the distributions on a
  - Say: the domain of a: V(A, R) = 100 (the number of distinct values a can take)
  - THEN, assuming uniformity
    - For each value of a
      - We have 10,000/100 = 100 tuples of R with that value of a
      - We have 5000/100 = 50 tuples of S with that value of a
      - All of these will join with each other, and produce 100 * 50 = 5000
    - So total number of results in the join:
      - 5000 * 100 (distinct values) = 500,000
  - We can improve the accuracy if we know the distributions on a better
    - Say using a histogram
Estimating Output Sizes: Other Ops

- Projection: $\Pi_A(R)$
  - If no duplicate elimination, THEN $|\Pi_A(R)| = |R|$
  - If distinct used (duplicate elimination performed): $|\Pi_A(R)| = V(A, R)$

- Set operations: (heuristic upper bounds)
  - Union ALL: $|R \cup S| = |R| + |S|$
  - Intersect ALL: $|R \cap S| = \min(|R|, |S|)$
  - Except ALL: $|R - S| = |R|$
  - Union, Intersection, Except (with duplicate elimination)
    - Somewhat more complex reasoning based on the frequency distributions etc…

- And so on …

Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation
- Summary
Query Optimization

- Integral component of query processing
  - Why?
- One of the most complex pieces of code in a database system
- Active area of research
  - E.g. XML Query Optimization?
  - What if you don’t know anything about the statistics
  - Better statistics
  - Etc …