Query Optimization

- Introduction
- Statistical (Catalog) Information for Cost Estimation
- Estimation of Statistics
- Cost-based vs. rule-based optimization
- Revisiting Selection Algorithms
- Transformation of Relational Expressions and Equivalence rules
- Dynamic Programming for Choosing Evaluation Plan
There exist many alternative ways of evaluating a given query:

- Equivalent relational algebraic expressions
- Different algorithms for each operation (previous section)
- Different coordination between operations
- Different paging mechanisms, etc.
An *evaluation plan* defines all of these options for a given query.

- Also known as a *query plan*, or *query execution plan*.
Cost difference between different plans can be enormous:
- 17 hours vs. 1/2 hour for 100k row, two-table join.

Optimizer estimates the cost of a plan based on statistics that it maintains:
- Number of tuples, number of distinct values for attributes, attribute histograms, etc.
- Statistics are estimated for intermediate results as well.
- Statistics must be accurate and up-to-date.

Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm:
- Merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
- Nested-loop join may be costlier than most other algorithms, but may provide opportunity for pipelining.
Cost-based optimization - a query plan is developed as follows:
1. Generate logically equivalent expressions using equivalence rules.
2. Annotating resultant expressions to get alternative query plans.
3. Choosing the cheapest plan based on estimated cost.

Rule-based (Heuristic) optimization - a query plan is developed by applying rules, or heuristics, that should reduce query cost.
- No cost estimate is made.
- Example - apply selections as early as possible.

Relative to rule-based optimization, cost-based optimization is expensive, but worthwhile for queries on large datasets.

Most real query optimizers incorporate elements of both.
Recall the table “statistics” that we have been using...

- \( n_r \): number of tuples in a relation \( r \).
- \( b_r \): number of blocks containing tuples of \( r \).
- \( f_r \): blocking factor of \( r \), i.e., the number of tuples of \( r \) that fit into one block.

If tuples of \( r \) are stored together physically in a file, then:

\[
b_r = \left\lfloor \frac{n_r}{f_r} \right\rfloor
\]
More generally:

\[ b_r >= \left\lfloor \frac{n_r}{f_r} \right\rfloor \]
Some additional “statistics” that the query optimizer will make use of…

- $V(A, r)$: number of distinct values that appear in $r$ for attribute $A$
  - $V(\text{branch-name, account})$, $V(\text{city, branch})$
  - Same as the size of $\prod_A(r)$.

- $SC(A, r)$: selection cardinality for attribute $A$ of relation $r$, average number of records that satisfy equality on $A$.
  - $SC(\text{branch-name, account})$, $SC(\text{city, branch})$
  - For simplicity, we assume a uniform distribution, i.e., $SC(A,r) = n_r / V(A,r)$
  - More generally, a histogram is usually stored for each attribute of a relation, e.g., $SC(v, A, r)$

- $\lceil SC(A, r)/f_r \rceil$ — number of blocks that these records will occupy if the relation is sorted on attribute $A$. 
Statistical Information about Indices

- \( \text{min}(A,r) \): minimum value on attribute \( A \) in relation \( r \).

- \( \text{max}(A,r) \): maximum value on attribute \( A \) in relation \( r \).

- \( f_i \): average fan-out of internal nodes of index \( i \), for tree-structured indices such as B+ trees (note: overloaded notation!)

- \( HT_i \): number of levels in (height of) index \( i \):
  - For a B+ tree on attribute \( A \) of relation \( r \), \( HT_i = \left\lceil \log_f(V(A,r)) \right\rceil \).
  - For a hash index, \( HT_i \) is 1 (or 2)

- \( LB_i \): number of lowest-level B+ tree index blocks in \( i \) — i.e, the number of blocks at the leaf level of the index.
In addition to the I/O used by specific algorithms, the following properties of the result of a query are frequently estimated:

- Result size, primarily in terms of the number of tuples.
- The number of distinct values for a specific attribute, i.e., \( V(A, r) \).

These estimates are independent of the algorithm used for an operation.

Nonetheless, they are frequently used to evaluate the cost of an algorithm.

- Particularly when the result of a sub-query is provided to another query.
Some of the previous analysis can now be refined…

**A2 (binary search).** Applicable if selection is an equality comparison on the attribute on which file is ordered.

- Assume that the blocks of a relation are stored contiguously
- Cost estimate becomes (number of disk blocks to be scanned):

  \[
  E_{a2} = \left\lceil \log_2(b_r) \right\rceil + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil - 1
  \]

- If condition for equality is on a key attribute, then \( SC(A,r) = 1 \)
A4 (primary index on nonkey, equality) Retrieve multiple records.
  - Records will be on consecutive blocks.
  - Cost = $HT_i + \text{number of blocks containing retrieved records}$.

$$E_{A4} = HT_i + \left[ \frac{SC(A, r)}{f_r} \right]$$
A5  (*secondary index, equality*).

- Retrieve a single record if the search-key is a candidate key
  - Cost = $HT_i + 1$

- Retrieve multiple records if search-key is not a candidate key
  - Cost = $HT_i + SC(A,r)$
Selections of the form $\sigma_{A \leq V}(r)$

- indices
- a linear or binary search (exercise)

First, let's estimate $c$, the number of tuples that will be returned, i.e., that satisfy the condition (assume the attribute is an int type, for simplicity):

- $c = 0$ if $v < \min(A, r)$
- $c = n_r$ if $v > \max(A, r)$
- $c = n_r \times \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$ otherwise

- In absence of statistical information (i.e., min and max) $c$ is assumed to be $n_r/2$.

*The case of $\sigma_{A \geq V}(r)$ is symmetric - a similar analysis applies.*
Now, let's determine the cost, i.e., the amount of block I/O.

**A6 (primary index, comparison \( \sigma_{A \geq V}(r) \)):**
- Assume the index is a B-tree; for a hash index, the analysis is more complicated.
- Use the index to find the first tuple \( \geq v \).
- Scan relation sequentially from that point.
- Cost estimate is:
  \[
  HT_i + \left\lceil \frac{c}{f_r} \right\rceil
  \]
  From the previous page.
  - In the absence of statistical information (e.g., min and max), \( c \) is assumed to be \( n_r/2 \) in which case:
  \[
  HT_i + \left\lceil \frac{b_r}{2} \right\rceil
  \]
- For \( \sigma_{A \leq V}(r) \) just scan relation sequentially till first tuple > \( v \) (exercise)
A7 (secondary index, comparison).

- As with the previous example, assume the index is a B-tree.
- For $\sigma_{A \geq V}(r)$ use the index to find first leaf-level index entry $\geq v$ and scan the leaf-level of the index sequentially from there, following pointers to records.
- In the worst case this requires an I/O for each retrieved record.
- Cost estimate is:

$$HT_i + \left\lceil \frac{LB_i \cdot c}{n_r} \right\rceil + c$$

- In the absence of statistical information, $c$ is assume to be $n_r/2$ in which case:

$$HT_i + \left\lceil \frac{LB_i}{2} \right\rceil + \left\lceil \frac{n_r}{2} \right\rceil$$

- For $\sigma_{A < V}(r)$ just scan leaf pages of the index finding pointers to records, till first entry $> v$; estimate is similar.
Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples in every legal database.

An equivalence rule asserts that two expressions are equivalent.

Equivalence rules could be used in either a cost-based or rule-based optimizer.
1. Conjunctive selection operations can be deconstructed into a sequence of individual selections:

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

2. Selection operations are commutative:

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

3. Only the last in a sequence of projection operations is needed (assumes that \( t_i \leq t_{i+1} \)):

\[ \Pi_{t_1} (\Pi_{t_2} (\ldots (\Pi_{t_n}(E))\ldots)) = \Pi_{t_1}(E) \]
4. Selections can be combined with Cartesian products and theta joins:
   a. \( \sigma_0 (E_1 \times E_2) = E_1 \bowtie_0 E_2 \)
   b. \( \sigma_{01} (E_1 \bowtie_{02} E_2) = E_1 \bowtie_{01 \bowtie 02} E_2 \)

5. Theta-join operations (and natural joins) are commutative:
   \[
   E_1 \bowtie_0 E_2 = E_2 \bowtie_0 E_1
   \]

6. (a) Natural join operations are associative:
   \[
   (E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)
   \]

   (b) Theta joins are associative in the following manner:
   \[
   (E_1 \bowtie_{01} E_2) \bowtie_{02 \bowtie 03} E_3 = E_1 \bowtie_{01 \bowtie 03} (E_2 \bowtie_{02} E_3) 
   \]
   \( \Rightarrow ? \)

   where \( \theta_2 \) involves attributes from only \( E_2 \) and \( E_3 \)
7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in $\theta_0$ involve only the attributes of one of the expressions ($E_1$) being joined.

   $$\sigma_{\theta_0}(E_1 \Join_0 E_2) = (\sigma_{\theta_0}(E_1)) \Join_0 E_2$$

   (b) When $\theta_1$ involves only the attributes of $E_1$ and $\theta_2$ involves only the attributes of $E_2$.

   $$\sigma_{\theta_1 \land \theta_2}(E_1 \Join_0 E_2) = (\sigma_{\theta_1}(E_1)) \Join_0 (\sigma_{\theta_2}(E_2))$$
8. The projections operation distributes over the theta join operation as follows:

(a) Consider a join \( E_1 \bowtie_\theta E_2 \), and let \( L_1 \) and \( L_2 \) be sets of attributes from \( E_1 \) and \( E_2 \), respectively. If \( \theta \) involves only attributes from \( L_1 \cup L_2 \):

\[
\prod_{L_1 \cup L_2} (E_1 \bowtie_\theta E_2) = (\prod_{L_1} (E_1)) \bowtie_\theta (\prod_{L_2} (E_2))
\]

(b) Consider a join \( E_1 \bowtie_\theta E_2 \), and let \( L_1 \) and \( L_2 \) be sets of attributes from \( E_1 \) and \( E_2 \), respectively. Now let \( L_3 \) be attributes of \( E_1 \) that are involved in join condition \( \theta \), but are not in \( L_1 \cup L_2 \), and let \( L_4 \) be attributes of \( E_2 \) that are involved in join condition \( \theta \), but are not in \( L_1 \cup L_2 \). Then:

\[
\prod_{L_1 \cup L_2} (E_1 \bowtie_\theta E_2) = \prod_{L_1 \cup L_2} ((\prod_{L_1 \cup L_3} (E_1)) \bowtie_\theta (\prod_{L_2 \cup L_4} (E_2)))
\]
9. The set operations union and intersection are commutative:

\[ E_1 \cup E_2 = E_2 \cup E_1 \]
\[ E_1 \cap E_2 = E_2 \cap E_1 \]

10. Set union and intersection are associative:

\[ (E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3) \]
\[ (E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3) \]
11. The selection operation distributes over $\cup$, $\cap$ and $-$

$$
\sigma_{\theta} (E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2)
$$
and similarly for $\cup$ and $\cap$ in place of $-$
$$
\sigma_{\theta} (E_1 - E_2) = \sigma_{\theta}(E_1) - E_2
$$
and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$
\Pi_L (E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))
$$

There could be other rules…
- Query - Find the names of all customers who have an account at some branch located in Brooklyn:

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$$

- Transformation using rule 7a gives:

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$$

- Performing the selection as early as possible reduces the size of the relation to be joined.
Example with Multiple Transformations

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

\[ \Pi_{\text{customer-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"} \land \text{balance} > 1000} (\text{branch} \bowtie (\text{account} \bowtie \text{depositor}))) \]

- Transformation using join associatively (Rule 6a):

\[ \Pi_{\text{customer-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"} \land \text{balance} > 1000} ((\text{branch} \bowtie \text{account}) \bowtie \text{depositor})) \]

- Applying rules 7a followed by 7b provides an opportunity to apply the “perform selections early” rule, resulting in the sub-expression:

\[ \Pi_{\text{customer-name}} (((\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch} \bowtie \sigma_{\text{balance} > 1000} (\text{account})) \bowtie \text{depositor})) \]
The previous transformation drawn graphically:

(a) Initial Expression Tree

(b) Tree After Multiple Transformations
Projection Operation Example

Consider:

\[ \Pi_{\text{customer-name}} ((\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch} \times \text{account}) \times \text{depositor}) \]

When we compute:

\[ (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch} \times \text{account}) \]

we obtain a relation whose schema is:

\[ (\text{branch-name, branch-city, assets, account-number, balance}) \]

Add and push projections using equivalence rules 8a and 8b to eliminate unneeded attributes from intermediate results:

\[ \Pi_{\text{cn}} (\Pi_{\text{an}} (\Pi_{\text{bn}} (\sigma_{\text{bc} = \text{"Brooklyn"}} (\text{branch}) \times \Pi_{\text{an, bn}} (\text{account}) \times \text{depositor})) \]
Join Ordering Example

- Equivalence rule 6 states that:
  \[(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)\]

- Consequently, if \(r_2 \bowtie r_3\) is quite large relative to the size of \(r_1 \bowtie r_2\), then it might be better to evaluate the expression as:
  \[(r_1 \bowtie r_2) \bowtie r_3\]

so that the size of the temporary relation is minimized.
Consider the expression:
\[ \Pi_{\text{customer-name}} (\sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}) \bowtie \text{account} \bowtie \text{depositor}) \]

Could compute \text{account} \bowtie \text{depositor} first, and join result with:
\[ \sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}) \]
but \text{account} \bowtie \text{depositor} is likely to be a large relation.

Probably better to first compute:
\[ \sigma_{\text{branch-city} = \text{"Brooklyn"}} (\text{branch}) \bowtie \text{account} \]
since this is likely to be relatively small.
The theta join operation is both commutative and associated (6 & 7).

Given relations \( r_1, r_2, \ldots, r_n \), there are therefore a large number of join orders; more specifically, there are \( (2(n - 1))!/(n - 1)! \) join orders (Exercise: derive the formula).

- \( n = 7 \), the number is 665280
- \( n = 10 \), the number is greater than 176 billion!

Explicitly generating and evaluating each join order would be expensive and also redundant (as noted previously).

Is there a way to examine all the orderings more efficiently?
Equivalent join orderings frequently share common sub-expressions:

\[
\begin{align*}
((r_1 \bowtie r_2) \bowtie r_3) \bowtie r_4 \\
(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4) \\
((r_1 \bowtie r_2) \bowtie r_4) \bowtie r_3)
\end{align*}
\]

Time and space requirements can be reduced by developing a plan for each unique sub-expression at most once, and gradually combining them from the bottom-up, i.e., \textit{dynamic programming}.

This applies to expressions other than joins also.
To find best plan for a set $S$ of $n$ relations, consider all possible plans of the form $S_1 \bowtie (S - S_1)$ where $S_1$ is any non-empty proper subset of $S$.

Recursively compute the costs for joining the relations in $S_1$ and the relations in $(S - S_1)$, and then choose the cheapest of the $2^n - 1$ alternatives.

Whenever the plan for any subset is computed, store it for later use so that it doesn’t need to be re-computing.
Example, suppose \( S = \{r_1, r_2, r_3, r_4, r_5, r_6\} \)

<table>
<thead>
<tr>
<th>( S_1 )</th>
<th>( S - S_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{r_3}</td>
<td>{r_1, r_2, r_4, r_5, r_6}</td>
</tr>
<tr>
<td>{r_6}</td>
<td>{r_1, r_2, r_3, r_4, r_5}</td>
</tr>
<tr>
<td>{r_3, r_4}</td>
<td>{r_1, r_2, r_5, r_6}</td>
</tr>
<tr>
<td>{r_1, r_2, r_3}</td>
<td>{r_4, r_5, r_6}</td>
</tr>
<tr>
<td>{r_1, r_4, r_5, r_6}</td>
<td>{r_2, r_3}</td>
</tr>
<tr>
<td>{r_2, r_3, r_4, r_5}</td>
<td>{r_1, r_6}</td>
</tr>
<tr>
<td>{r_2, r_3, r_4}</td>
<td>{r_1, r_6}</td>
</tr>
</tbody>
</table>
Dynamic Programming
Join Order Optimization Algorithm

\( S \): A set of relations to be joined \( \{r_1, r_2, \ldots, r_n\} \)

\( bestplan \): An array containing one location for each non-empty proper subset of \( S \).

- Each location contains two values \( bestplan[S].cost \) and \( bestplan[S].plan \)
- For each set \( S \) containing one relation, \( bestplan[S].cost = 0 \), for all others \( bestplan[S].cost = \infty \)

```
procedure findbestplan(S)
    if (bestplan[S].cost \neq \infty)
        // bestplan[S] has been computed earlier, so return it
        return bestplan[S];
    
    // bestplan[S] has not been computed earlier, so compute it now
    for (each non-empty subset \( S1 \) of \( S \) such that \( S1 \neq S \)) loop
        P1 = findbestplan(S1);
        P2 = findbestplan(S - S1);
        A = best algorithm for joining results of \( P1 \) and \( P2 \);
        cost = \( P1.cost + P2.cost + \text{cost of } A \);
        if (cost < bestplan[S].cost) then
            bestplan[S].cost = cost;
            bestplan[S].plan = “execute \( P1.plan \); execute \( P2.plan \);
            join results of \( P1 \) and \( P2 \) using \( A \)”
        end if;
    end loop;
    return bestplan[S];
end;
```
Cost of Dynamic Programming Algorithm

- Worst case running time is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!

- Space used is $O(n2^n)$

- Can be improved by considering only:
  - left-deep join orders
  - Join orders that return tuples in a useful (“interesting”) order
In *left-deep join trees* the right-hand-side input for each join is a relation, not the result of an intermediate join.

The worst case running time of the modified algorithm is $O(n2^n)$

- Space complexity remains at $O(n2^n)$
Rule-Based (Heuristic) Optimization

- Cost-based optimization is expensive.

- Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion.

- Heuristic optimization transforms the query-tree by using a set of rules that typically improve query performance:
  - Perform selections early
  - Perform projections early
  - Perform the 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 projections as far down the tree as possible, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).

6. Identify those sub-trees whose operations can be pipelined, and execute them using pipelining.
Structure of Query Optimizers

- IBM System R:
  - Built as part of a research project in the 1970’s at IBM.
  - The first implementation of SQL.
  - First RDBMS to demonstrate good transaction processing performance.
  - The origin of DB2 can be traced back to System R.

- The System R Optimizer (Starburst):
  - Considers only left-deep join orders.
  - Uses dynamic programming to reduce complexity.
  - Supports pipelined evaluation.
  - Also uses heuristics to push selections and projections down the query tree.

- See the class website for related links.
End of Chapter

(Extra slides with details of selection cost estimation follow)
As noted previously, cost-based optimizers use equivalence rules to systematically generate expressions equivalent to a given expression, using an algorithm such as:

\[ S = \{ e \}; \]

```
repeat
  for (each \( ex1 \) in \( S \)) loop
    for (each equivalence rule \( r \)) loop
      if (\( r \) applies to \( ex1 \)) then {
        apply \( r \) to \( ex1 \) to get \( ex2 \);
        add \( ex2 \) to \( S \);
      }

  until (no new expressions have been found);
```

This approach is over-simplified, and very expensive in terms of both time and space, i.e., it does not “prune” the search space very well, if at all.
Some query optimizers integrate heuristic selection and the generation of alternative access plans.

- System R and Starburst use a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization.

Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.

- This expense is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses.
Join Operation: Running Example

Running example: depositor customer

Catalog information for join examples:

- \( n_{\text{customer}} = 10,000 \)
- \( f_{\text{customer}} = 25 \), which implies that \( b_{\text{customer}} = \frac{10000}{25} = 400 \)
- \( n_{\text{depositor}} = 5000 \)
- \( f_{\text{depositor}} = 50 \), which implies that \( b_{\text{depositor}} = \frac{5000}{50} = 100 \)
- \( V(\text{customer-name, depositor}) = 2500 \), which implies that, on average, each customer has two accounts.
  Also assume that customer-name in depositor is a foreign key on customer.
Let $R$ and $S$ be the sets of attributes for relations $r$ and $s$, respectively.

The size estimate for the natural join of $r$ and $s$ depends on the common attributes.

- **Question**: Why not just use $|r| \times |s|$?
- This would certainly be a valid upper-bound.

If $R \cap S = \emptyset$, then $r \times s$ is the same as $r \times s$.

- The Cartesian product $r \times s$ contains $n_r \times n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
Estimation of the Size of Joins

- If \( R \cap S \) is a key for \( R \), then a tuple of \( s \) will join with at most one tuple from \( r \).
  - Therefore, the number of tuples in \( r \) \( s \) is no greater than the number of tuples in \( s \)
  - Question: Could it be greater than the number of tuples in \( r \)?

- If \( R \cap S \) is a foreign key in \( S \) referencing \( R \), then the number of tuples in \( r \) \( s \) is exactly the same as the number of tuples in \( s \).
  - The case for \( R \cap S \) being a foreign key referencing \( S \) is symmetric.

- In the example query depositor \( \text{customer, customer-name in depositor} \) is a foreign key of \( \text{customer} \)
  - Hence, the result has exactly \( n_{\text{depositor}} \) tuples, which is 5000
Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for $R$ or $S$.

  If we assume that every tuple $t$ in $R$ produces tuples in $R \times S$, the number of tuples in $R \times S$ is estimated to be:

  $$\frac{n_r \times n_s}{V(A,s)} = n_r \times \frac{n_s}{V(A,s)} = n_r \times SC(A,s)$$

  If the reverse is true, the estimate is:

  $$\frac{n_r \times n_s}{V(A,r)} = n_s \times \frac{n_r}{V(A,s)} = n_s \times SC(A,r)$$

- Conjecture: $\min(\frac{n_r \times n_s}{V(A,r)}, \frac{n_r \times n_s}{V(A,s)})$

  - In other words, (according to the book) the lower of these two estimates is probably the more accurate one ($V(A,r)$ is probably not equal to $V(A,s)$).
  
  - This is probably true since $A$ is neither a key nor a foreign key, and so neither $r$ nor $s$ is likely to have every tuple included in the result.
  
  - Question: Since we typically do a worst-case analysis, shouldn’t the larger of the two be used?
Compute the size estimates for depositor customer without using information about foreign keys:

- \( V(\text{customer-name, depositor}) = 2500 \), and \( V(\text{customer-name, customer}) = 10000 \)
- The two estimates are \( 5000 \times 10000/2500 = 20,000 \) and \( 5000 \times 10000/10000 = 5000 \)
- We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.
Outer join:

- Estimated size of \( r \times s = \text{size of } r \times s + \text{size of } r \)
  - Case of right outer join is symmetric
- Estimated size of \( r \times s = \text{size of } r \times s + \text{size of } r + \text{size of } s \)
- As in the previous case, these establish upper bounds.

Note that many of these estimates may be quite inaccurate, but typically provide upper bounds on result sizes.

The slides contain at the end of the chapter contain estimates for the number of distinct values produced by various operations.

- We will skip these, and you are not responsible for them.
Size Estimation for Other Operations

- Projection: estimated size of $\Pi_A(r) = V(A, r)$

- Aggregation: estimated size of $A\{F(r) = V(A, r)$

- Set operations
  - For unions/intersections of selections on the same relation rewrite and use size estimate for selections:
    - $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ can be rewritten as $\sigma_{\theta_1 \cup \theta_2}(r)$.
    - $\sigma_{\theta_1}(r) \cap \sigma_{\theta_2}(r)$ can be rewritten as $\sigma_{\theta_1}(\sigma_{\theta_2}(r))$.
  - For operations on different relations:
    - estimated size of $r \cup s = \text{size of } r + \text{size of } s$.
    - estimated size of $r \cap s = \text{min}(\text{size of } r, \text{size of } s)$.
    - estimated size of $r - s = r$. 
**Optimizing Nested Subqueries**

- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**

- E.g.
  
  ```sql
  select customer-name 
  from borrower 
  where exists (select * 
                from depositor 
                where depositor.customer-name = borrower.customer-name)
  ```

- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level `from` clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery
Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result

SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques

E.g.: earlier nested query can be rewritten as

```
select customer-name
from   borrower, depositor
where depositor.customer-name = borrower.customer-name
```

- Note: above query doesn’t correctly deal with duplicates, can be modified to do so as we will see

In general, it is not possible/straightforward to move the entire nested subquery from clause into the outer level query from clause
- A temporary relation is created instead, and used in body of outer level query
In general, SQL queries of the form below can be rewritten as shown

- Rewrite: select ...
  from $L_1$
  where $P_1$ and exists (select *
    from $L_2$
    where $P_2$)

- To: create table $t_1$ as
  select distinct $V$
  from $L_2$
  where $P_2^1$
  select ...
  from $L_1$, $t_1$
  where $P_1$ and $P_2^2$

- $P_2^1$ contains predicates in $P_2$ that do not involve any correlation variables
- $P_2^2$ reintroduces predicates involving correlation variables, with relations renamed appropriately
- $V$ contains all attributes used in predicates with correlation variables
In our example, the original nested query would be transformed to

```
create table t_1 as
  select distinct customer-name
  from depositor
  select customer-name
  from borrower, t_1
  where t_1.customer-name = borrower.customer-name
```

The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.

Decorrelation is more complicated when

- the nested subquery uses aggregation, or
- when the result of the nested subquery is used to test for equality, or
- when the condition linking the nested subquery to the other query is **not exists**, or
- and so on.
A **materialized view** is a view whose contents are computed and stored.

Consider the view

```sql
create view branch-total-loan(branch-name, total-loan) as
select branch-name, sum(amount)
from loan
groupby branch-name
```

Materializing the above view would be very useful if the total loan amount is required frequently

- Saves the effort of finding multiple tuples and adding up their amounts
The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**

Materialized views can be maintained by recomputation on every update.

A better option is to use **incremental view maintenance**

- Changes to database relations are used to compute changes to materialized view, which is then updated.

View maintenance can be done by

- Manually defining triggers on insert, delete, and update of each relation in the view definition.
- Manually written code to update the view whenever database relations are updated.
- Supported directly by the database.
The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**

- Set of tuples inserted to and deleted from \( r \) are denoted \( i_r \) and \( d_r \)

To simplify our description, we only consider inserts and deletes

- We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple

We describe how to compute the change to the result of each relational operation, given changes to its inputs

We then outline how to handle relational algebra expressions
Join Operation

Consider the materialized view \( v = r \Join s \) and an update to \( r \)

Let \( r^{\text{old}} \) and \( r^{\text{new}} \) denote the old and new states of relation \( r \)

Consider the case of an insert to \( r \):

- We can write \( r^{\text{new}} \Join s \) as \( (r^{\text{old}} \cup i_r) \Join s \)
- And rewrite the above to \( (r^{\text{old}} \Join s) \cup (i_r \Join s) \)
- But \( (r^{\text{old}} \Join s) \) is simply the old value of the materialized view, so the incremental change to the view is just \( i_r \Join s \)

Thus, for inserts \( v^{\text{new}} = v^{\text{old}} \cup (i_r \Join s) \)

Similarly for deletes \( v^{\text{new}} = v^{\text{old}} - (d_r \Join s) \)
Selection and Projection Operations

- Selection: Consider a view \( v = \sigma_{\theta}(r) \).
  - \( v^{new} = v^{old} \cup \sigma_{\theta}(i_r) \)
  - \( v^{new} = v^{old} - \sigma_{\theta}(d_r) \)

- Projection is a more difficult operation
  - \( R = (A,B) \), and \( r(R) = \{ (a,2), (a,3) \} \)
  - \( \Pi_A(r) \) has a single tuple \((a)\).
  - If we delete the tuple \((a,2)\) from \( r \), we should not delete the tuple \((a)\) from \( \Pi_A(r) \), but if we then delete \((a,3)\) as well, we should delete the tuple

- For each tuple in a projection \( \Pi_A(r) \), we will keep a count of how many times it was derived
  - On insert of a tuple to \( r \), if the resultant tuple is already in \( \Pi_A(r) \) we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from \( r \), we decrement the count of the corresponding tuple in \( \Pi_A(r) \)
    - if the count becomes 0, we delete the tuple from \( \Pi_A(r) \)
Aggregation Operations

- **count**: \( v = \text{Ag}_{\text{count}}(B) \).
  - When a set of tuples \( i_r \) is inserted
    - For each tuple \( r \) in \( i_r \), if the corresponding group is already present in \( v \), we increment its count, else we add a new tuple with count = 1
  - When a set of tuples \( d_r \) is deleted
    - For each tuple \( t \) in \( i_r \), we look for the group \( t.A \) in \( v \), and subtract 1 from the count for the group.
      - If the count becomes 0, we delete from \( v \) the tuple for the group \( t.A \)

- **sum**: \( v = \text{Ag}_{\text{sum}}(B) \).
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from \( v \)
    - Cannot simply test for sum = 0 (why?)

To handle the case of **avg**, we maintain the **sum** and **count** aggregate values separately, and divide at the end
min, max: \( v = {}^{A}g_{\min (B)} (r) \).

- Handling insertions on \( r \) is straightforward.
- Maintaining the aggregate values min and max on deletions may be more expensive. We have to look at the other tuples of \( r \) that are in the same group to find the new minimum.
Set intersection: \( v = r \cap s \)

- when a tuple is inserted in \( r \) we check if it is present in \( s \), and if so we add it to \( v \).
- If the tuple is deleted from \( r \), we delete it from the intersection if it is present.
- Updates to \( s \) are symmetric
- The other set operations, \textit{union} and \textit{set difference} are handled in a similar fashion.

Outer joins are handled in much the same way as joins but with some extra work

- we leave details to you.
To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.

E.g. consider $E_1 \cdot E_2$ where each of $E_1$ and $E_2$ may be a complex expression

- Suppose the set of tuples to be inserted into $E_1$ is given by $D_1$
  - Computed earlier, since smaller sub-expressions are handled first

- Then the set of tuples to be inserted into $E_1 \cdot E_2$ is given by $D_1 \cdot E_2$
  - This is just the usual way of maintaining joins
Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view \( v = r \) \( s \) is available
  - A user submits a query \( r \) \( s \) \( t \)
  - We can rewrite the query as \( v \) \( t \)
    - Whether to do so depends on cost estimates for the two alternative

- Replacing a use of a materialized view by the view definition:
  - A materialized view \( v = r \) \( s \) is available, but without any index on it
  - User submits a query \( \sigma_{A=10}(v) \)
  - Suppose also that \( s \) has an index on the common attribute B, and \( r \) has an index on attribute A.
    - The best plan for this query may be to replace \( v \) by \( r \) \( s \), which can lead to the query plan \( \sigma_{A=10}(r) \) \( s \)

- Query optimizer should be extended to consider all above alternatives and choose the best overall plan
Materialized view selection: “What is the best set of views to materialize?”.  
  ➢ This decision must be made on the basis of the system workload.

Indices are just like materialized views, problem of index selection is closely related, to that of materialized view selection, although it is simpler.

Some database systems, provide tools to help the database administrator with index and materialized view selection.
Selection Cost Estimate Example

\[ \sigma_{\text{branch-name}} = \text{“Perryridge”}(\text{account}) \]

- Number of blocks is \( b_{\text{account}} = 500 \): 10,000 tuples in the relation; each block holds 20 tuples.

- Assume \text{account} is sorted on \text{branch-name}.
  - \( V(\text{branch-name}, \text{account}) \) is 50
  - \( 10000/50 = 200 \) tuples of the \text{account} relation pertain to Perryridge branch
  - \( 200/20 = 10 \) blocks for these tuples
  - A binary search to find the first record would take \( \lceil \log_2(500) \rceil = 9 \) block accesses

- Total cost of binary search is \( 9 + 10 - 1 = 18 \) block accesses (versus 500 for linear scan)
Selections Using Indices

- **Index scan** – search algorithms that use an index; condition is on search-key of index.

- **A3** *(primary index on candidate key, equality)*. Retrieve a single record that satisfies the corresponding equality condition, $E_{A3} = HT_i + 1$

- **A4** *(primary index on nonkey, equality)* Retrieve multiple records. Let the search-key attribute be $A$.

- **A5** *(equality on search-key of secondary index)*.
  - Retrieve a single record if the search-key is a candidate key, $E_{A5} = HT_i + 1$
  - Retrieve multiple records (each may be on a different block) if the search-key is not a candidate key, $E_{A3} = HT_i + SC(A,r)$
Consider the query is $\sigma_{\text{branch-name} = \text{“Perryridge”}}(\text{account})$, with the primary index on \textit{branch-name}.

- Since $V(\text{branch-name, account}) = 50$, we expect that $10000/50 = 200$ tuples of the \textit{account} relation pertain to the Perryridge branch.

- Since the index is a clustering index, $200/20 = 10$ block reads are required to read the \textit{account} tuples.

- Several index blocks must also be read. If B$^+$-tree index stores 20 pointers per node, then the B$^+$-tree index must have between 3 and 5 leaf nodes and the entire tree has a depth of 2. Therefore, 2 index blocks must be read.

- This strategy requires 12 total block reads.
Selections Involving Comparisons

selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using a linear file scan or binary search, or by using indices in the following ways:

- **A6 (primary index, comparison).** The cost estimate is:

$$E_{AB} = HT_i + \left\lfloor \frac{c}{f_r} \right\rfloor$$

where $c$ is the estimated number of tuples satisfying the condition. In absence of statistical information $c$ is assumed to be $n_r/2$.

- **A7 (secondary index, comparison).** The cost estimate:

$$E_{m} = HT_r + \frac{LB_i \cdot c}{n_r}$$

where $c$ is defined as before. (Linear file scan may be cheaper if $c$ is large!).
Consider a selection on `account` with the following condition: `where branch-name = “Perryridge” and balance = 1200`

Consider using algorithm A8:

- The `branch-name` index is clustering, and if we use it the cost estimate is 12 block reads (as we saw before).

- The `balance` index is non-clustering, and $V(balance, account = 500, so the selection would retrieve 10,000/500 = 20 accounts. Adding the index block reads, gives a cost estimate of 22 block reads.

- Thus using `branch-name` index is preferable, even though its condition is less selective.

- If both indices were non-clustering, it would be preferable to use the `balance` index.
Consider using algorithm A10:

- Use the index on \textit{balance} to retrieve set $S_1$ of pointers to records with $balance = 1200$.

- Use index on \textit{branch-name} to retrieve set $S_2$ of pointers to records with $branch-name = \text{Perryridge}$.

- $S_1 \cap S_2 =$ set of pointers to records with $branch-name = \text{“Perryridge”}$ and $balance = 1200$.

- The number of pointers retrieved (20 and 200), fit into a single leaf page; we read four index blocks to retrieve the two sets of pointers and compute their intersection.

- Estimate that one tuple in 50 * 500 meets both conditions. Since $n_{\text{account}} = 10000$, conservatively overestimate that $S_1 \cap S_2$ contains one pointer.

- The total estimated cost of this strategy is five block reads.
Statistical Information for Examples

- $f_{account} = 20$ (20 tuples of account fit in one block)
- $V(branch-name, account) = 50$ (50 branches)
- $V(balance, account) = 500$ (500 different balance values)
- $\pi_{account} = 10000$ (account has 10,000 tuples)
- Assume the following indices exist on account:
  - A primary, B*-tree index for attribute branch-name
  - A secondary, B*-tree index for attribute balance
Pictorial Depiction of Equivalence Rules

Rule 4

Rule 5

Rule 6a
If θ only has attributes from E1

©Silberschatz, Korth and Sudarshan
Selections: $\sigma_\theta(r)$

- If $\theta$ forces $A$ to take a specified value: $V(A, \sigma_\theta(r)) = 1$.
  - e.g., $A = 3$

- If $\theta$ forces $A$ to take on one of a specified set of values: $V(A, \sigma_\theta(r)) = \text{number of specified values}$.
  - (e.g., $(A = 1 \lor A = 3 \lor A = 4)$),

- If the selection condition $\theta$ is of the form $A \ op \ r$
  estimated $V(A, \sigma_\theta(r)) = V(A, r) * s$
  - where $s$ is the selectivity of the selection.

- In all the other cases: use approximate estimate of $\min(V(A, r), n_{\sigma_\theta(r)})$
  
  - More accurate estimate can be obtained using probability theory, but this one works fine generally
Estimation for the Number of Distinct Values (Cont.)

Joins: \( r \quad s \)

- If all attributes in \( A \) are from \( r \)
  
  \[
  V(A, r \quad s) = \min (V(A, r), n_{r \quad s})
  \]

- If \( A \) contains attributes \( A_1 \) from \( r \) and \( A_2 \) from \( s \), then estimated \( V(A, r \quad s) = \)

  \[
  \min( V(A_1, r) \cdot V(A_2 - A_1, s), V(A_1 - A_2, r) \cdot V(A_2, s), n_{r \quad s} )
  \]

- More accurate estimate can be obtained using probability theory, but this one works fine generally
Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\Pi_A(r)$ as in $r$.

- The same holds for grouping attributes of aggregation.

- For aggregated values
  - For min($A$) and max($A$), the number of distinct values can be estimated as
    $\min(V(A,r), V(G,r))$ where $G$ denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use $V(G,r)$
Complex Selections Revisited

- **Conjunction:** $\sigma_{\theta_1 \land \theta_2 \land \cdots \land \theta_n}(r)$
  - A8 (*conjunctive selection using one index*) – cost estimate follows directly from the estimates for A1 through A7, assuming all terms are “simple.”
  - A9 (*conjunctive selection using multiple-key index*).
  - A10 (*conjunctive selection by intersection of identifiers*).

- **Disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \cdots \lor \theta_n}(r)$
  - A11 (*disjunctive selection by union of identifiers*).

- **Negation:** $\sigma_{\neg \theta}(r)$
For A9, A10 and A11, the formulas on page 14 for estimating the number of tuples \( c \) satisfying the condition might be very inaccurate.

If \( s_i \) is the number of tuples satisfying \( \theta_i \) in \( r \), then the fraction of tuples satisfying \( \theta_i \) is \( s_i/n_r \) (the book calls this the “selectivity” of \( \theta_i \)).

Conjunction: \( \sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n} (r) \) - An estimate for number of satisfying tuples:

\[
n_r \ast \frac{s_1 \ast s_2 \ast \ldots \ast s_n}{n_r^n}
\]

Disjunction: \( \sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n} (r) \) - An estimate for the number of satisfying tuples:

\[
n_r \ast \left(1 - \frac{s_1}{n_r}\right) \ast \left(1 - \frac{s_2}{n_r}\right) \ast \ldots \ast \left(1 - \frac{s_n}{n_r}\right)
\]

Negation: \( \sigma_{\neg \theta} (r) \). Estimated number of tuples:

\[
n_r - \text{size}(\sigma_{\theta}(r))
\]