Database Management Systems

Query optimization

DBMS Architecture

SQL INSTRUCTION

OPTIMIZER

CONCURRENCY CONTROL

MANAGEMENT OF ACCESS METHODS

BUFFER MANAGER

RELIABILITY MANAGEMENT

DATABASE

Index Files

Data Files

System Catalog
Query optimizer

- It selects an efficient strategy for query execution
  - It is a fundamental building block of a relational DBMS
- It guarantees the *data independence* property
  - The form in which the SQL query is written does not affect the way in which it is implemented
  - A physical reorganization of data does not require rewriting SQL queries

- It automatically generates a *query execution plan*
  - It was formerly hard-coded by a programmer
- The automatically generated execution plan is usually more efficient
  - It evaluates many different alternatives
  - It exploits statistics on data, stored in the system catalog, to make decisions
  - It exploits the best known strategies
  - It dynamically adapts to changes in the data distribution
Lexical, syntactic and semantic analysis

- Analysis of a statement to detect
  - **Lexical errors**
    - e.g., misspelled keywords
  - **Syntactic errors**
    - errors in the grammar of the SQL language
  - **Semantic errors**
    - references to objects which do not actually exist in the database (e.g., attributes or tables)
    - information in the data dictionary is needed
Lexical, syntactic and semantic analysis

Output
- Internal representation in (extended) relational algebra

Why relational algebra?
- It explicitly represents the order in which operators are applied
  - It is procedural (different from SQL)
- There is a corpus of theorems and properties
  - Exploited to modify the initial query tree

Query optimizer

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

DATA DICTIONARY
### Algebraic optimization

- Execution of algebraic transformations considered to be always beneficial
  - Example: anticipation of selection with respect to join
- Should eliminate the difference among different formulations of the same query
- This step is usually independent of the data distribution
- Output
  - Query tree in "canonical" form

### Query optimizer

- SQL Query
- Lexical, syntactic and semantic analysis
- Data Dictionary
- Internal representation based on relational algebra
- Algebraic optimization
- "Canonical" query tree
- Cost based optimization
Cost based optimization

Selection of the “best” execution plan by evaluating *execution cost*

- Selection of
  - the best access method for each table
  - the best algorithm for each relational operator among available alternatives
- Based on a cost model for access methods and algorithms
- Generation of the code implementing the best strategy

Output

- Access program in executable format
  - It exploits the internal structures of the DBMS
- Set of dependencies
  - conditions on which the validity of the query plan depends
    - e.g., the existence of an index
Query optimizer

SQL QUERY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

“CANONICAL” QUERY TREE

“CANONICAL” QUERY TREE

COST BASED OPTIMIZATION

COST BASED OPTIMIZATION

DATA PROFILES (STATISTICS ON DATA)

DATA PROFILES (STATISTICS ON DATA)

ACCESS PROGRAM

ACCESS PROGRAM

SET OF DEPENDENCIES

SET OF DEPENDENCIES

Execution modes

Compile and go

 Compiled and immediate execution of the statement

No storage of the query plan

Dependencies are not needed
Execution modes

Compile and store
- The access plan is stored in the database together with its dependencies
- It is executed *on demand*
- It should be recompiled when the data structure changes

Algebraic optimization
Algebraic optimization

SQL QUERY

DATA DICTIONARY

LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

INTERNAL REPRESENTATION BASED ON RELATIONAL ALGEBRA

ALGEBRAIC OPTIMIZATION

“CANONICAL” QUERY TREE

COST BASED OPTIMIZATION

DATA PROFILES (STATISTICS ON DATA)

ACCESS PROGRAM

SET OF DEPENDENCIES

- It is based on equivalence transformations
  - Two relational expressions are *equivalent* if they both produce the same query result for any arbitrary database instance

- Interesting transformations
  - reduce the size of the intermediate result to be stored in memory
  - prepare an expression for the application of a transformation which reduces the size of the intermediate result
Transformations

1. Atomization of selection
   - $\sigma_{F_1 \land F_2}(E) \equiv \sigma_{F_2}(\sigma_{F_1}(E)) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$

2. Cascading projections
   - $\pi_X(E) \equiv \pi_X(\pi_{X,Y}(E))$
Transformations

1. Atomization of selection
   \[ \sigma_{F_1 \land F_2} (E) \equiv \sigma_{F_2} (\sigma_{F_1} (E)) \equiv \sigma_{F_1} (\sigma_{F_2} (E)) \]

2. Cascading projections
   \[ \pi_X (E) \equiv \pi_X (\pi_{X,Y} (E)) \]

3. Anticipation of selection with respect to join (pushing selection down)
   \[ \sigma_F (E_1 \bowtie E_2) \equiv E_1 \bowtie (\sigma_F (E_2)) \]
   - \( F \) is a predicate on attributes in \( E_2 \) only

4. Anticipation of projection with respect to join
   \[ \pi_L (E_1 \bowtie p E_2) \equiv \pi_L ((\pi_{L_1,J} (E_1)) \bowtie p (\pi_{L_2,J} (E_2))) \]
   - \( L_1 = L - \text{Schema}(E_2) \)
   - \( L_2 = L - \text{Schema}(E_1) \)
   - \( J = \text{set of attributes needed to evaluate join predicate } p \)
5. Join derivation from Cartesian product
   - \( \sigma_F (E_1 \times E_2) \equiv E_1 \bowtie_F E_2 \)
   - predicate \( F \) only relates attributes in \( E_1 \) and \( E_2 \)

6. Distribution of selection with respect to union
   - \( \sigma_F (E_1 \cup E_2) \equiv (\sigma_F (E_1)) \cup (\sigma_F (E_2)) \)
5. Join derivation from Cartesian product
   - $\sigma_F (E_1 \times E_2) \equiv E_1 \bowtie_F E_2$
   - predicate $F$ only relates attributes in $E_1$ and $E_2$

6. Distribution of selection with respect to union
   - $\sigma_F (E_1 \cup E_2) \equiv (\sigma_F (E_1)) \cup (\sigma_F (E_2))$

7. Distribution of selection with respect to difference
   - $\sigma_F (E_1 - E_2) \equiv (\sigma_F (E_1)) - (\sigma_F (E_2))$
   - $\equiv (\sigma_F (E_1)) - E_2$

8. Distribution of projection with respect to union
   - $\pi_X (E_1 \cup E_2) \equiv (\pi_X (E_1)) \cup (\pi_X (E_2))$
8. Distribution of projection with respect to union
   \[ \pi_X(E_1 \cup E_2) \equiv (\pi_X(E_1)) \cup (\pi_X(E_2)) \]

\[ \Rightarrow \text{Can projection be distributed with respect to difference?} \]
\[ \pi_X(E_1 - E_2) \equiv (\pi_X(E_1)) - (\pi_X(E_2)) \]

This equivalence only holds if \( X \) includes the primary key or a set of attributes with the same properties (unique and not null)
9. Other properties

- \( \sigma_{F_1 \lor F_2}(E) \equiv (\sigma_{F_1}(E)) \cup (\sigma_{F_2}(E)) \)
- \( \sigma_{F_1 \land F_2}(E) \equiv (\sigma_{F_1}(E)) \cap (\sigma_{F_2}(E)) \)

10. Distribution of join with respect to union

- \( E \bowtie (E_1 \cup E_2) \equiv (E \bowtie E_1) \cup (E \bowtie E_2) \)

▷ All binary operators are commutative and associative except for difference
Example:

Tables

EMP (Emp#, ………, Dept#, Salary)
DEPT (Dept#, DName,……………)

SQL query

SELECT DISTINCT DName
FROM EMP, DEPT
WHERE EMP.Dept#=DEPT.Dept#
AND Salary > 1000;

Example: Algebraic transformations

\[ \pi_{DName} (\sigma_{\text{EMP.Dept#} = \text{DEPT.Dept#} \land \text{Salary} > 1000} (\text{EMP} \times \text{DEPT})) \]
Example: Algebraic transformations

\[ \pi_{\text{DName}} (\sigma_{\text{EMP.Dept#}=\text{DEPT.Dept#}} \land \text{Salary} > 1000 (\text{EMP} \times \text{DEPT})) \]

Prop #1

\[ \pi_{\text{DName}} (\sigma_{\text{Salary} > 1000} (\sigma_{\text{EMP.Dept#}=\text{DEPT.Dept#}} (\text{EMP} \times \text{DEPT}))) \]

Example: Algebraic transformations

\[ \pi_{\text{DName}} (\sigma_{\text{EMP.Dept#}=\text{DEPT.Dept#}} \land \text{Salary} > 1000 (\text{EMP} \times \text{DEPT})) \]

Prop #1

\[ \pi_{\text{DName}} (\sigma_{\text{Salary} > 1000} (\sigma_{\text{EMP.Dept#}=\text{DEPT.Dept#}} (\text{EMP} \times \text{DEPT}))) \]

Prop #5

\[ \pi_{\text{DName}} (\sigma_{\text{Salary} > 1000} (\text{EMP} \bowtie \text{DEPT})) \]
Example: Algebraic transformations

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} (\text{EMP} \bowtie \text{DEPT})) \]

Prop #3

\[ \pi_{\text{DName}}(\sigma_{\text{Salary} > 1000} (\text{EMP})) \bowtie \text{DEPT} \]

Prop #2 and #4

\[ \pi_{\text{DName}} \left( (\pi_{\text{Dept#}}(\sigma_{\text{Salary} > 1000}(\text{EMP})) \bowtie \pi_{\text{Dept#}, \text{DName}}(\text{DEPT})) \right) \]
Example: Query tree

Final query tree

Example: Cardinalities

- Cardinality (EMP) ≈ 10,000
- Cardinality (DEPT) ≈ 100
- Cardinality (EMP where Salary > 1000) ≈ 50
Cost based optimization

- SQL Query
- Lexical, syntactic, and semantic analysis
- Internal representation based on relational algebra
- Algebraic optimization
- "Canonical" query tree
- Cost based optimization
- Data dictionary
- Data profiles (statistics on data)
- Access program
- Set of dependencies
Cost based optimization

- It is based on
  - Data profiles
    - statistical information describing data distribution for tables and intermediate relational expressions
  - Approximate cost formulas for access operations
    - Allow evaluating the cost of different alternatives for executing a relational operator

Database Management Systems

Data profiles
Table profiles

- Quantitative information on the characteristics of tables and columns
  - cardinality (# of tuples) in each table $T$
    - also estimated for intermediate relational expressions
  - size in bytes of tuples in $T$
  - size in bytes of each attribute $A_j$ in $T$
  - number of distinct values of each attribute in $T$
    - cardinality of the active domain of the attribute
  - min and max values of each attribute $A_j$ in $T$

Table profiles are stored in the data dictionary
Profiles should be periodically refreshed by re-analyzing data in the tables
- Update statistics command
- Executed on demand
  - immediate execution during transaction processing would overload the system
Table profiles are exploited to estimate the size of intermediate relational expressions.

- For the selection operator
  \[ \text{Card} (\sigma_{A_i = v} (T)) \approx \frac{\text{Card} (T)}{\text{Val} (A_i \text{ in } T)} \]

  \[ \text{Val} (A_i \text{ in } T) = \# \text{ of distinct values of } A_i \text{ in } T \text{ (active domain)} \]

  It holds only under the hypothesis of uniform distribution.
Query tree

- Internal representation of the relational expression as a query tree

```
π_{DName} (π_{Dept#} (σ_{Salary>1000} EMP)) ∪ π_{Dept#,DName} DEPT
```

- Leaves correspond to the physical structures
  - tables, indices

- Intermediate nodes are operations on data supported by the given physical structure
  - e.g., scan, join, group by
Sequential scan

- Executes sequential access to all tuples in a table
  - also called full table scan
- Operations performed during a sequential scan
  - Projection
    - discards unnecessary columns
  - Selection on a simple predicate \((A_i = v)\)
  - Sorting based on an attribute list
  - Insert, update, delete

Sorting

- Classical algorithms in computer science are exploited
  - e.g., quick sort
- Size of data is relevant
  - memory sort
  - sort on disk
Predicate evaluation

- If available, it may exploit index access
  - B+-tree, hash, or bitmap
- Simple equality predicate $A_i = v$
  - Hash, B+-tree, or bitmap are appropriate
- Range predicate $v_1 \leq A_i \leq v_2$
  - only B+-tree is appropriate
- For predicates with limited selectivity, full table scan is better
  - if available, consider bitmap

B+-tree versus bitmap

<table>
<thead>
<tr>
<th>NK</th>
<th>Data space (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>35</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>45</td>
<td>9</td>
</tr>
</tbody>
</table>

B-tree: $NR \times \text{Len(Pointer)}$
Bitmap: $NR \times NK \times 1$ bit

Len(Pointer) = 4×8 bit

Conjunction of predicates $A_i = v_1 \land A_j = v_2$

- The most selective predicate is evaluated first
  - Table is read through the index
- Next the other predicates are evaluated on the intermediate result

Optimization

- First compute the intersection of bitmaps or RIDs coming from available indices
- Next table read and evaluation of remaining predicates

Example: Predicate evaluation

Which female students living in Piemonte are exempt from enrollment fee?

<table>
<thead>
<tr>
<th>RID</th>
<th>Gender</th>
<th>Exempt</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
<td>Y</td>
<td>Piemonte</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>Y</td>
<td>Liguria</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
<td>N</td>
<td>Puglia</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>N</td>
<td>Sicilia</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>Y</td>
<td>Piemonte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender</th>
<th>Exempt</th>
<th>Piemonte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

RID 5
Disjunction of predicates $A_i = v_1 \lor A_j = v_2$

- Index access can be exploited only if all predicates are supported by an index
- Otherwise full table scan

Join operation

- A critical operation for a relational DBMS
  - Connection between tables is based on values instead of pointers
  - Size of the intermediate result is typically larger than the smaller table
- Different join algorithms
  - Nested loop
  - Merge scan join
  - Hash join
  - Bitmapped join
Nested loop

A single full scan is done on the outer table.

For each tuple in the outer table:
- A full scan of the inner table is performed, looking for corresponding values.

Also called “brute force”
**Nested loop**

- **Efficient when**
  - inner table is small and fits in memory
  - optimized scan
  - join attribute in the inner table is indexed
  - index scan
- **Execution cost**
  - The nested loop join technique is *not symmetric*
  - The execution cost depends on which table takes the role of inner table

**Merge scan**

<table>
<thead>
<tr>
<th>Left table</th>
<th>Right table</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>e</td>
</tr>
</tbody>
</table>

- **join attribute**
**Merge scan**

- Both tables are sorted on the join attributes
- The two tables are scanned in parallel
  - tuple pairs are generated on corresponding values
- Execution cost
  - The merge scan technique is *symmetric*
  - requires sorting both tables
    - may be sorted by a previous operation
    - may be read through a clustered index on join attributes
- More used in the past
  - efficient for large tables, because sorted tables may be stored on disk

**Hash Join**

From left table

<table>
<thead>
<tr>
<th>HASH(a)</th>
<th>d</th>
<th>e</th>
<th>a</th>
<th>c</th>
<th>j</th>
<th>p</th>
</tr>
</thead>
</table>

From right table

<table>
<thead>
<tr>
<th>HASH(a)</th>
<th>e</th>
<th>m</th>
<th>a</th>
<th>w</th>
</tr>
</thead>
</table>

Join Attribute

- Buckets for left table
- Buckets for right table
Hash join

- Application of the same hash function to the join attributes in both tables
  - Tuples to be joined end up in the same buckets
  - Collisions are generated by tuples yielding the same hash function result with different attribute value
  - A local sort and join is performed into each bucket
- Very fast join technique

Bitmapped join index

- Bit matrix that precomputes the join between two tables A and B
  - One column for each RID in table A
  - One row for each RID in table B
- Position (i, j) of the matrix is
  - 1 if tuple with RID j in table A joins with tuple with RID i in table B
  - 0 otherwise
- Updates may be slow

<table>
<thead>
<tr>
<th>RID</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Bitmapped join

Typically used in OLAP queries
- joining several tables with a large central table

Example
- Exam table, joined to Student and Course tables

Exploits one or more bitmapped join indices
- One for each pair of joined tables

Access to the large central table is the last step

Complex queries may exploit jointly
- bitmapped join indices
- bitmap indices for predicates on single tables
Example: Bitmapped join

Average score of male students for exams of courses in the first year of the master degree

- **STUDENT** (Reg#, SName, Gender)
- **COURSE** (Course#, CName, CourseYear)
- **EXAM** (Reg#, Course#, Date, Grade)

```sql
SELECT AVG (Grade) FROM STUDENT S, EXAM E, COURSE C WHERE E.Reg# = S.Reg# AND E.Course# = C.Course# AND CourseYear = '1M' AND Gender = 'M';
```

Bitmapped join

- **STUDENT** (Reg#, SName, Gender)
- **COURSE** (Course#, CName, CourseYear)
- **EXAM** (Reg#, Course#, Date, Grade)

```sql
SELECT AVG (Grade) FROM STUDENT S, EXAM E, COURSE C WHERE E.Reg# = S.Reg# AND E.Course# = C.Course# AND CourseYear = '1M' AND Gender = 'M';
```
### Bitmapped join

<table>
<thead>
<tr>
<th>RID&lt;sub&gt;cy&lt;/sub&gt;</th>
<th>RID&lt;sub&gt;G&lt;/sub&gt;</th>
<th>RID&lt;sub&gt;G&lt;/sub&gt;</th>
<th>RID&lt;sub&gt;cy&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

AND

 RIDs of Exam table for tuples to be read

bitmap for Course-Exam predicates and join

bitmap for Student-Exam predicates and join

---

### Group by

- **Sort based**
  - Sort on the group by attributes
  - Next compute aggregate functions on groups

- **Hash based**
  - Hash function on the group by attributes
  - Next sort each bucket and compute aggregate functions

- **Materialized views** may be exploited to improve the performance of aggregation operations
Execution plan selection

Cost based optimization

Inputs
- Data profiles
- Internal representation of the query tree

Output
- “Optimal” query execution plan
- Set of dependencies

It evaluates the cost of different alternatives for
- reading each table
- executing each relational operator

It exploits approximate cost formulas for access operations
The search for the optimal plan is based on the following dimensions:

- The way data is read from disk
  - e.g., full scan, index
- The execution order among operators
  - e.g., join order between two join operations
- The technique by means of which each operator is implemented
  - e.g., the join method
- When to perform sort (if sort is needed)

The optimizer builds a tree of alternatives in which:

- each internal node makes a decision on a variable
- each leaf represents a complete query execution plan
Example

Given 3 tables
- R, S, T

Compute the join
R ⋈ S ⋈ T

Execution alternatives
- 4 join techniques to evaluate (for both joins)
- 3 join orders
- In total, at most
  - 4 * 4 * 3 = 48 different alternatives

Example

\[
\begin{align*}
R & \rightarrow S \rightarrow T \\
(R & \rightarrow S) & \rightarrow T \\
(S & \rightarrow R) & \rightarrow T \\
(R & \rightarrow S) & \rightarrow T \\
(R & \rightarrow S) & \rightarrow T \\
S & \rightarrow R \\
T & \rightarrow R \\
T & \rightarrow R
\end{align*}
\]

- NESTED LOOP
- INNER
- MERGE SCAN
- HASH JOIN
- LEAF NODE
Best execution plan selection

The optimizer selects the leaf with the lowest cost.

General formula:
\[ C_{Total} = C_{I/O} \times n_{I/O} + C_{cpu} \times n_{cpu} \]
- \( n_{I/O} \) is the number of I/O operations
- \( n_{cpu} \) is the number of CPU operations

The selection is based on operation research optimization techniques,
- e.g., branch and bound

The final execution plan is an approximation of the best solution.

The optimizer looks for a solution which is of the same order of magnitude of the “best” solution:
- For compile and go
  - it stops when the time spent in searching is comparable to the time required to execute the current best plan.