Relational Database Systems 2
6. Query Optimization

Christoph Lofi
Benjamin Köhncke
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
5 Query Processing

• The Query Processor
  – How do DBMS actually answer queries?
  – Query Parsing/Translation
    • Relational Algebra
  – Query Optimization
  – Query Execution
  – Implementation of Joins
6.1 Introduction into query optimization
6.2 Motivating example
6.3 Algebraic query rewriting
6.4 Execution cost estimation
6.5 The SQL EXPLAIN statement
6.6 Choosing a plan
6.1 Introduction

• **Remember:** query processor

![Diagram showing the components of a database management system (DBMS) including disks, operating system, storage manager, query processor with evaluation engine, query optimizer, and parser, and applications/queries.]
• Heart of the Query Processor is the **Query Optimizer**
  
  – Translation of a query into a relational algebra expression leads to a first naïve query plan
  
  – The query optimizer transforms it into an **efficient** plan
    
    • Choice of physical operators
    • **Operator** sequence and grouping
  
  – The chosen plan is annotated and handed over to the evaluation engine
6.1 Introduction

• Query optimizer rewrites the naïve (canonical) query plan into a more efficient evaluation plan
6.1 Basic Considerations

- **Bottom-up vs. top-down approaches**
  - Either optimize individual queries and generalize the algorithms (bottom-up)
  - Or choose general algorithms for classes of queries to be applied to each individual query (top-down)
  - Most DBS are build using a top-down approach

- **Heuristics vs. cost-based optimization**
  - General heuristics allow to improve performance of most queries
  - Costs estimated from statistics allow for a good optimization of each specific query
  - Most DBS use a hybrid approach between heuristics and cost estimations
6.1 Preparing the Query

• **Basic mapping** from (declarative) query languages into a suitable internal format
  – Replace language keywords by respective operators while keeping the relations, attributes, conditions,…
  – Remember: mapping SQL into relational algebra

  • **SELECT** `attribute_1,…,attribute_n`  
    \[ \pi (attribute_1,\ldots,attribute_n) \]
  
  • **FROM** `relation_1,…,relation_k`  
    \[ (relation_1 \times \ldots \times relation_k) \]
  
  • **WHERE** `condition_1 AND/OR \ldots AND/OR condition_m`  
    \[ \sigma (condition_1 AND/OR \ldots AND/OR condition_m) \]
6.1 Preparing the Query

• Decompose query into \textbf{query blocks}
  – Exactly one SELECT and FROM clause
  – At most one WHERE, GROUP BY and HAVING clause

• No nesting allowed
  – Nested subqueries are usually optimized independently

• \textbf{Query normalization}
  – WHERE clause in conjunctive normal form

• Advantage:
  – Query expressions can be processed in parallel
6.1 Operator Trees

• All evaluation plans are usually **tree shaped sequences** of the relational algebra operators
  
  – Relations as leaf nodes
  – Operators as internal nodes with one or more children

  – SELECT \( \text{attribute}_1, \ldots, \text{attribute}_n \)
    FROM \( \text{relation}_1, \ldots, \text{relation}_k \)
    WHERE \( \text{condition}_1 \ \text{AND/OR} \ \ldots \ \text{AND/OR} \ \text{condition}_m \)

\[
\pi (\text{attribute}_1, \ldots, \text{attribute}_n)
\]
\[
\sigma (\text{condition}_1 \ \text{AND/OR} \ \ldots \ \text{AND/OR} \ \text{condition}_m)
\]

\[
\times
\]

\[
\ldots
\]

\[
\times
\]

\[
\text{relation}_1
\]

\[
\text{relation}_2
\]

\[
\text{relation}_k
\]
6.1 Operator Execution

• The algebraic representation of each operator abstracts from the actual algorithm used for evaluation

• Each operator takes all input relation(s), calculates the respective results, and puts the results into a temporary table
  – Intermediate results may be large (e.g., Cartesian product)
  – Materialization is often expensive
6.1 Pipelining

• If the query is composed of several operators results can also be **pipelined** between operators
  – For example, the result of a join can be directly pipelined into a selection
    • Every record produced by the join is immediately checked for the selection condition(s)
    • Thus, selection is applied on-the-fly

• Advantages
  – No creation of temporary tables necessary
  – No expensive writing to/ reading from disk
6.1 Pipelining

no pipelining

pipelining

Pipline 1

Pipline 2
6.1 Pipelining

• Within a pipeline, only tuples are passed among operations
  – Each operation has a buffer for storing tuples

• Pipelines can be executed in two ways
  – **Demand Driven** (Pull)
    • Top-Down
    • Operations actively demand next tuple from their inputs
  – **Producer Driven** (Push)
    • Each operation has an input buffer
    • Buffer is filled eagerly by previous operations using all available inputs
6.1 Pipelining – Iterator Interfaces

• Interfaces for demand driven pipelines
• The sequence of operators given by the evaluation plan has to be coordinated in the execution
• Relational operators support a uniform iterator interface hiding the implementation details
  – OPEN allocates buffer space for inputs/outputs and passes on parameters (e.g., selection conditions)
  – GET_NEXT repetitively calls operator specific code and can be used to control progression rates for operators
  – CLOSE deallocates all state information
6.1 Pipelining

• Pipelining restricts available operations
• Pipelining usually works well for
  – Selection, projection
  – Index nested loop joins
• Pipelining usually does not work well for
  – Sorting
  – Hash joins and merge joins
• Sometimes, materialization will be more efficient than pipelining
  – Hard to estimate
  – e.g., introducing materializing sorts to allow for merge joins
6.1 Algebraic Query Optimization

• Relational algebra usually allows for alternative, yet equivalent evaluation plans
  – Respective execution costs may strongly differ
    • Memory space, response time, etc.

• Idea: Find the best plan, before actually executing the query
6.1 Static vs. Dynamic Plans

• Basically there are two possible cases:
  – **Static plans**, where the best plan is known a-priori for a certain kind of query
    • The respective operator sequence and access paths are saved and always used for queries of a kind
    • Pre-optimized statements can be immediately evaluated
  – **Dynamic plans**, where the best plan has to be found at run-time for some query
    • Used, if querying behavior is very heterogeneous
### 6.2 Optimization Example

#### students

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
<th>lastName</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
<td>Kent</td>
<td>m</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
<td>Lane</td>
<td>f</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
<td>Luther</td>
<td>m</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
<td>Xavier</td>
<td>m</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
<td>Magnus</td>
<td>m</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
<td>Gray</td>
<td>f</td>
</tr>
<tr>
<td>9876</td>
<td>Bruce</td>
<td>Banner</td>
<td>m</td>
</tr>
<tr>
<td>11875</td>
<td>Peter</td>
<td>Parker</td>
<td>m</td>
</tr>
<tr>
<td>12546</td>
<td>Raven</td>
<td>Darkholme</td>
<td>f</td>
</tr>
</tbody>
</table>

#### courses

<table>
<thead>
<tr>
<th>crsNr</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Intro. to being a Superhero</td>
</tr>
<tr>
<td>101</td>
<td>Secret Identities 2</td>
</tr>
<tr>
<td>102</td>
<td>How to take over the world</td>
</tr>
</tbody>
</table>

#### exams

<table>
<thead>
<tr>
<th>matNr</th>
<th>crsNr</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>1.3</td>
</tr>
<tr>
<td>6676</td>
<td>102</td>
<td>1.3</td>
</tr>
<tr>
<td>5119</td>
<td>101</td>
<td>1.7</td>
</tr>
</tbody>
</table>

**Note:** The table sizes are indicated by the number of bytes: **4 Byte**, **30 Byte**, **30 Byte**, **1 Byte**.
6.2 Optimization Example

- **SQL Statement**
  
  "SELECT lastName, result, title FROM students s, exams e, courses c WHERE e.result <= 1.3 AND s.matNr = e.matNr AND e.crsNr = c.crsNr"

- **Canonical Relational Algebra Expression**
  
  – Expression directly mapped from the SQL query
  
  – "\( \pi_{\text{lastName}, \text{result}, \text{title}} \sigma_{\text{result} \leq 1.3 \land \text{exams.crsNr} = \text{courses.crsNr} \land \text{students.matNr} = \text{exams.matNr}} \text{students} \times \text{exams} \times \text{courses} \)"

<table>
<thead>
<tr>
<th>lastName</th>
<th>result</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnus</td>
<td>1.3</td>
<td>How to take over the world</td>
</tr>
<tr>
<td>Kent</td>
<td>1.3</td>
<td>Intro. to being a Superhero</td>
</tr>
</tbody>
</table>
6.2 Optimization Example

• Create Canonical **Operator Tree**
  – Operator tree visualized the order of primitive functions
  – (Note: Illustration is not really canonical tree as selection is already separated)

\[
\begin{align*}
\pi_{\text{lastName}, \text{result}, \text{title}} \\
\sigma_{\text{result} \leq 1.3} \land \text{exams.crsNr} = \text{courses.crsNr} \land \text{students.matNr} = \text{exams.matNr} \\
\text{students} \times \text{exams} \times \text{courses}
\end{align*}
\]
6.2 Optimization Example

How much **space** is needed for the intermediate results?

\[ \sum = 26,136 \]

\[ \pi_{lastName, result, title} \]

\[ \sigma_{result \leq 1.3} \]

\[ \sigma_{exams.crsNr = courses.crsNr} \]

\[ \sigma_{students.matNr = exams.matNr} \]

\[ 2 \times 68B = 136B \]

\[ 2 \times 115B = 230B \]

\[ 6 \times 115B = 690B \]

\[ 18 \times 115B = 2,070B \]

\[ 162 \times 115B = 18,630B \]

\[ 54 \times 81B = 4,374B \]

\[ 9 \times 65B = 585B \]

\[ 6 \times 16B = 96B \]

\[ 3 \times 34B = 102B \]

Total: 26913 B
Remember: task of query optimization

- Transform **canonical operator tree** into more efficient **final operator tree** for evaluation
- Canonical and final tree have **equal semantics**, but different operators / execution orders
- Common **Heuristics** and/or **DB statistics** are used to transform canonical tree step by step
  - Heuristic query optimization
  - Cost-based query optimization
6.2 Optimization Example

Example: Final Operator Tree

\[
\begin{align*}
\text{students} & \rightarrow \pi_{\text{lastName}, \text{matNr}} & 9 \times 65B = 585B \\
\text{exams} & \rightarrow \pi_{\text{lastName}, \text{result}, \text{crsNo}} & 9 \times 34B = 306B \\
\text{courses} & \rightarrow \sigma_{\text{result} \leq 1.3} & 2 \times 16B = 32B \\
\text{exams} & \rightarrow \pi_{\text{result}, \text{crsNo}} & 2 \times 76B = 152B \\
\text{students} & \rightarrow \pi_{\text{lastName}, \text{matNr}} & 2 \times 42B = 84B \\
\text{exams} & \rightarrow \pi_{\text{result}, \text{crsNo}} & 2 \times 50B = 100B \\
\text{courses} & \rightarrow \sigma_{\text{result} \leq 1.3} & 2 \times 16B = 32B \\
\end{align*}
\]

\[
\begin{align*}
\text{students} \bowtie \text{exams} \bowtie \text{courses} & \rightarrow \pi_{\text{lastName}, \text{result}, \text{title}} & \text{Total: } 1593B \\
\end{align*}
\]
6.3 Algebraic Query Rewriting

• All transformations are based on a set of relational algebra equivalences
  – Algebra allows for symbolic calculations
  – **Transformation rules** transform an operator tree into another, equivalent tree step by step
  – Results of a query are **never affected** by transformations
6.3 Algebraic Query Rewriting

• Selections

1. Cascading $\sigma$
   
   $\sigma_{c_1 \land c_2 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

2. Commutativity of $\sigma$
   
   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$
6.3 Algebraic Query Rewriting

• Projections

3. Cascading $\pi$
   - Only the last projection in a cascade takes effect
   - $\pi_{\text{list}_1} (\pi_{\text{list}_2} (\ldots (\pi_{\text{list}_n} (R))\ldots) \equiv \pi_{\text{list}_1}$
   - $\text{list}_1 \supseteq \text{list}_2, \text{list}_3, \ldots, \text{list}_n$

4. Commuting $\pi$ with $\sigma$
   - Only possible, if selection condition $c$ does only work on projected attributes $a_1, \ldots, a_n$
   - $\pi_{a_1, a_2, \ldots, a_n} (\sigma_c (R)) \equiv \sigma_c (\pi_{a_1, a_2, \ldots, a_n} (R))$
• Joins and Cartesian products

5. Commutativity of $\times$ (and $\bowtie$)
   - $R \times S \equiv S \times R$
   - $R \bowtie S \equiv S \bowtie R$

6. Associativity of $\times$ (or $\bowtie$)
   - $R \times (S \times T) \equiv (S \times R) \times T$
   - $R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T$

• Both together allow for **arbitrary order** of joins
7. Constructing $\bowtie$ from $\sigma_c$ and $\times$
   - $R \bowtie_{c_1} S \equiv \sigma_{c_1}(R \times S)$

8. Commuting $\sigma_c$ with $\bowtie$ (or $\times$)
   - Condition $c$ is concatenation of clauses involving either attributes from $R$, or from $S$ connected with $\land$
     - $c_1$ contains clauses from $c$ with attributes in $R$ and $S$
     - $c_2$ contains only clauses from $c$ with attributes in $R$
     - $c_3$ contains only clauses from $c$ with attributes in $S$
   - $\sigma_c(R \bowtie S) \equiv \sigma_{c_1}(\sigma_{c_2}(R)) \bowtie (\sigma_{c_3}(S))$
7. Commuting $\pi_{list}$ with $\bowtie_c$ (or $\times$)

- Attribute list $list$ contains only attributes from $R$, $S$
  - $list1$ contains all attributes from $list$ with attributes in $R$
  - $list2$ contains all attributes from $list$ with attributes in $S$
  - $c$ involves only attributes in $list$
  - $\pi_{list}(R \bowtie_c S) \equiv (\pi_{list1}(R)) \bowtie_c (\pi_{list2}(S))$

- If $c$ also involves attributes not on $list$
  - $list1$ and $list2$ are extended with those attributes not in $list$
  - Additional projection necessary
  - $\pi_{list}(R \bowtie_c S) \equiv \pi_{list}(\pi_{list1}(R)) \bowtie_c (\pi_{list2}(S))$
6.3 Algebraic Query Rewriting

• Set operations

8. Commutativity of $\cup$ and $\cap$
   - $R \cup S \equiv S \cup R$ and $R \cap S \equiv S \cap R$

9. Associativity of $\cup$ and $\cap$
   - $R \cup (S \cup T) \equiv (R \cup S) \cup T$
   - $R \cap (S \cap T) \equiv (R \cap S) \cap T$

10. Commuting $\sigma$ with set operations
    - $\Theta \in \{\cup, \cap, -\}$: $\sigma_c (R \Theta S) \equiv (\sigma_c (R) \Theta \sigma_c (S))$

11. Commuting $\pi$ with set operation
    - $\Theta \in \{\cup, \cap, -\}$: $\pi_{list} (R \Theta S) \equiv (\pi_{list} (R) \Theta \pi_{list} (S))$
6.4 Cost-based Optimization

• All transformations can be applied to the canonical evaluation plan
  – However, there is no best operator sequence that is always optimal
  – Efficiency depends on the current data instance, the actual implementation of base operations, the existence of access paths and indexes, etc.

• Idea: assign average costs to operators (nodes) and estimate costs for each query plan
6.4 Database Statistics

• By weighting specific statistics cost-optimizer make **assumptions** about the system’s bottleneck
  – Focusing on expected block hits for operators assumes that the bottleneck is **I/O-bound**
    • Typical for database systems relying secondary storage
    • Block hits for reading indexes are often ignored
  – Focusing on CPU statistics assumes that bottleneck is **CPU-bound**
    • E.g., main memory databases
6.4 Database Statistics

• For each node in the operator tree
  – The cost of performing the corresponding operation has to be estimated
    • Consider input size, available indexes, I/O and processing costs
    • Consider whether pipelining applies or result materialization has to be used
  – The size of the result has to be estimated
    • Important to estimate expected input for parent node
    • Distinguish sorted/unsorted results
6.4 Database Statistics

• **Estimation of costs** starts with simple parameters
  – Database buffer size
  – Cardinality of the input base relations
    • Number of records
    • Number (distribution) of distinct domain values
  – Relation size
    • Number of pages on disk
  – Available access paths
    • Index cardinalities (#keys), sizes (#pages), heights, ranges
• Maintained in the DBMS’s **system catalog**
6.4 Estimating Result Sizes

- Important factor for finding good plans is to keep **intermediate results** of operators small
  
  - `SELECT attribute_list`  
    `FROM relation_list`  
    `WHERE condition_1 AND … AND condition_n`

  - **Maximum number** of result records?
    - Product of cardinalities of all relations (incl. duplicates)

  - Conditions in **WHERE** clause eliminate result records
    - **Selectivity** of the conditions
    - **Reduction factors** help to estimate real result size

  - **Projections** do not reduce record number (unless duplicates are removed)
6.4 Estimating Result Sizes

real size \approx \text{maximum size} \times \prod_i (\text{reduction factor}_{\text{condition } i})

- **Assumption:** All reductions are statistically independent
  - Somewhat unrealistic, but…?!

- **How to estimate reduction factors?**
  - Depends on the kind of the condition
    - column = value
    - column\_1 = column\_2
    - column \{>, <, \geq, \leq\} value
    - column IN {list of values}
6.4 Estimating Result Sizes

- **Condition**: `column = value`
  - Simple assumption in early query optimizers, if column was not indexed: reduction factor of 0.1 (System R)
    - Today, statistics about the distinct values and histograms can do a lot better
  - If the column is indexed by some index $I$, the reduction factor can be approximated by $1/#\text{keys}(I)$
6.4 Estimating Result Sizes

• **Condition:** \texttt{column}_1 = \texttt{column}_2

  – If there are indexes \( I_1 \) and \( I_2 \) on columns 1 and 2, the reduction factor can be estimated by
    \[
    \frac{1}{\max(\#\text{keys}(I_1), \#\text{keys}(I_2))}
    \]
  • The formula assumes that for each key in the smaller index there is a matching partner in the larger index

  – If only one column has an index \( I \), the estimation can be simplified to \( \frac{1}{\#\text{keys}(I)} \)

  – If neither column has an index, statistics about the distinct domain values can be used like above
6.4 Estimating Result Sizes

• **Condition**: `column {>, <, ≥, ≤} value`
  
  – If there is an index `I`, the reduction factor can be approximated by \((\text{high}(I) - \text{value})/(\text{high}(I) - \text{low}(I))\)
  
  – If there is no index, or the column is not of an arithmetic type usually a factor a little less than 0.5 is assumed
    
    • The assumption is that value is somewhere in the middle of the domain value range
  
  – For range queries \((\text{value} < \text{column} < \text{value})\) result sizes can be estimated as disjunctions of both conditions
6.4 Estimating Result Sizes

• Condition: column IN {list of values}
  – Here if there is an index $I$, the reduction factor is chosen as $\#\text{list values}*(1/\#\text{keys}(I))$
    • Analogously to column = value
    • Generally, the factor should be at most 0.5

• Conditions of the kind column IN (subquery) are handled similar
  – Ratio of the estimated subquery result size to the number of distinct values in column in the outer relation
6.4 Estimating Block Accesses

- The number of **DB block accesses** using an index for a simple selection `column = value` (assuming uniform value distribution) depends on
  - the **type of index** and
  - the **result size** selected from the indexed column

  - If there is a **primary index**, then both result size and number of blocks accessed is about 1
  - If there is a **cluster index**, the expected number of blocks accessed for a selection is
    \[ \#blocks \approx \frac{\#result\,size}{(\#records\,-\,in\,-\,relation/\#blocks\,-\,in\,-\,relation)} \]
  - For a **secondary index** the expected number of blocks accessed is \( \#blocks \approx \#result\,size \)
    - …if each block is looked up individually
  - If no index is given, \( \#blocks \approx \#blocks\,-\,in\,-\,relation \)
6.4 Estimating Block Accesses

• The number of **DB block accesses** using an index for a simple selection column {>, <, ≥, ≤} value (assuming uniform value distribution) again depends on
  – the **type of index** and the **result size** selected from the indexed column
  • If there is a **primary or cluster index**, then number of blocks accessed is about \#blocks ≈ \#blocks-in-relation / 2
  • For a **secondary index** the expected number of blocks accessed is \#blocks ≈ \#records-in-relation / 2
  • If no index is given, \#blocks ≈ \#blocks-in-relation
6.4 Estimating Block Accesses

• The number of **DB block accesses** using an index for a selection **column = column** basically depends on the selectivity of the join
  
  – Ordering joins differently may lead to plans with vastly differing costs
  
  – Next lecture: **join order optimization**

  – Worst case is a full **Cartesian product** between two relations \( \mathbf{R} \times \mathbf{S} \) where each tuple of \( \mathbf{R} \) has to be joined with each tuple of \( \mathbf{S} \)
    
    • \( \# \text{blocks} \approx \# \text{blocks}_R \times \# \text{records}_S \)
6.4 Estimated Sizes

- Example Database: IMDB Data
  - Internet Movie Database
  - Contains (among others)
    - 1,181,300 movies of 7 types
    - 2,226,551 persons
    - 15,387,808 associations actors-movies
Filter Movies: `title.production_year = x`

- System R heuristic: 0.1
- Distinct values in index: 0.0075
- Sample Queries
  - Year=2000 : 0.0235
  - Year=1970 : 0.0077
  - Year=1940 : 0.0017
  - Data is skewed! Histograms should provide better results!
6.4 Estimated Sizes

• Filter Actor Assignments: `name.id=cast_info.person_id`
  – Distinct values: Reduction Factor 8.46 E-7
  – Sample Queries
    • Reduction Factor 8.46 E-7

• Estimate Number of Block Accesses: `title.production_year<1920`
  – Assume 16 records per block -> 73,832 blocks
  – No Index: 73,832 blocks
  – Secondary Index: 484,333 blocks (estimated reduction factor 0.59)
    • Usage of index seems not a good idea here…
    • However, real result size is just 56,502 records
      – Now a good idea?
If several indexes match the conditions in the WHERE clause, each offers an alternative access path

- The selectivity of an access path is the number of DB pages on disk that have to be retrieved to evaluate a selection, relative to the whole relation
  - Usually worst: relation scans
- Choosing the right access path determines the I/O-bound efficiency of the query evaluation
6.4 Choosing the Access Path

• Deciding for an single-index access path is implemented by predicate splitting
  – For each condition compute the least expensive form of evaluation (considering all applicable indexes)
  – Start with the global least expensive condition, use the respective access path, and apply all other predicates on the respective result in the database buffer

• Predicate splitting leads to suboptimal, but usually sufficient results
Deciding for an multiple-index access path helps, if several selection conditions are indexed with secondary indexes

- **Block(or record)-IDs** can be retrieved for suitable search keys from several indexes and the respective set is intersected
- If the set is then ordered by Block-Ids, the actual data can be efficiently accessed
- All selection conditions not matching some index can then be applied in the database buffer
6.4 Choosing the Access Path

- Deciding for an **sorted-index access path** is efficient, if a tree index exists that can be traversed in the specific order needed
  - e.g., for aggregations and GROUP BY clauses
  - All remaining selection conditions are applied on-the-fly for each retrieved tuple
  - The strategy works well for cluster indexes
6.4 Choosing the Access Path

- Deciding for an **index-only access path** is rarely possible, but very efficient
  - Works only, if **all attributes** in the query are part of the search key for some **dense index**
    - All information is provided by the index
    - There is always only one index entry per data record
  
- **Selection conditions can be applied directly** on the index entries
  - Only the index has to be scanned, the actual data does not need to be accessed
6.5 EXPLAIN Statements

• How to know which plan/access paths the database chose for evaluating a query?
  – Use the so called **EXPLAIN**-statement
  – **EXPLAIN** analyzes the execution of a given SQL query and stores its results into explain tables within the DB
    • Shows operation **execution order**
    • Collects **metrics** for each basic operation
    • Allows for bottleneck **analysis** and manual query **optimization** (‘what-if’ analysis)
6.5 EXPLAIN Statements
6.5 EXPLAIN Statements

• Before using Explain manually, explain tables need to be created

• Explain Tables contain results/measurements per
  – Statement
  – Operator
  – Predicate
  – Object
  – Stream
  – Instance
• Execute an explain statement
  – „EXPLAIN PLAN SET queryno=<qno> FOR <query>”
  – Result of explained query are stored in explain tables, identified by queryno

• Example
  – “EXPLAIN PLAN SET queryno=127 FOR SELECT * FROM imdb.name”
  – “SELECT total_cost, statement_text FROM explain_statement WHERE queryno=127”
    • Result: <39841, “SELECT * FROM imdb.name”>
### 6.5 EXPLAIN Statements

<table>
<thead>
<tr>
<th>QUERYNO</th>
<th>QBLOCKNO</th>
<th>APPLNAME</th>
<th>PROGNAME</th>
<th>PLANNO</th>
<th>METHOD</th>
<th>CREATOR</th>
<th>TNAME</th>
<th>TABNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1411</td>
<td>1</td>
<td>RQATD</td>
<td>1</td>
<td>0</td>
<td>SYSIBM</td>
<td>SYSTABLES</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACCESTYPE</th>
<th>MATCHCOLS</th>
<th>ACCESSCREATOR</th>
<th>ACCESSNAME</th>
<th>INDEXONLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>SYSIBM</td>
<td>DSNDTX02</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SORTN</th>
<th>UNIQ</th>
<th>SORTN</th>
<th>ORDERBY</th>
<th>SORTN</th>
<th>GROUPBY</th>
<th>SORTC</th>
<th>UNIQ</th>
<th>SORTC</th>
<th>JOIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SORTC</th>
<th>ORDERBY</th>
<th>SORTC</th>
<th>GROUPBY</th>
<th>TSLOCKMODE</th>
<th>TIMESTAMP</th>
<th>REMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
<td>IS</td>
<td>2007073116080531</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PREFETCH</th>
<th>COLUMN</th>
<th>FN</th>
<th>EVAL</th>
<th>MIXOPSEQ</th>
<th>VERSION</th>
<th>COLLID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CAD72_2004-08-31-18.24.46</td>
<td>RQPAR110_ALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COLLID</th>
<th>ACCESS DEGREE</th>
<th>ACCESS PGROUP ID</th>
<th>JOIN DEGREE</th>
<th>JOIN PGROUP ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQPAR110_ALL</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SORTC PGROUP_ID</th>
<th>SORTN PGROUP_ID</th>
<th>PARALLELISM MODE</th>
<th>MERGE JOIN COLS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>------</td>
<td></td>
<td>------</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CORRELATION NAME</th>
<th>PAGE RANGE</th>
<th>JOIN TYPE</th>
<th>GROUP_MEMBER</th>
<th>IBM SERVICE DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>YA. W T â</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WHEN OPTIMIZE</th>
<th>QBLOCK TYPE</th>
<th>BIND TIME</th>
<th>OPPTHINT</th>
<th>HINT USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>2007-07-31-16.08.03.615184</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRIMARY ACCESTYPE</th>
<th>PARENT QBLOCKNO</th>
<th>TABLE_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>
Example Database: IMDB Data

- Internet Movie Database
- Contains (among others)
  - 1,181,300 movies of 7 types
  - 2,226,551 persons
  - 15,387,808 associations actors-movies
Develop cost-based optimization for sample query

“In which cinema movies did Harrison Ford act before 1986?”
6.5 Effect of Access Paths

• **Idea:** Create the statement step-by-step
  – In each step, **examine the query plan** used by the DB

• **SELECT** `n.id` **FROM** `IMDB.NAME` `n` **WHERE**
  `n.name='Ford, Harrison' AND n.imdb_index='I';`
  – Query again, this time with secondary index on name

![Diagram]

<table>
<thead>
<tr>
<th>Method</th>
<th>Cost</th>
<th>IO</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table Scan</td>
<td>40E3</td>
<td>38E3</td>
<td>5.9E9</td>
</tr>
<tr>
<td>Index Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch</td>
<td>30</td>
<td>3</td>
<td>103E3</td>
</tr>
</tbody>
</table>

Relational Database Systems 2 – Christoph Lofi - Benjamin Köhncke – Institut für Informationssysteme
6.5 Effect of Access Paths

- SELECT c.movie_id
  FROM IMDB.CAST_INFO c
  WHERE c.person_id=260907;
  
  – Query again, this time with secondary index on person_id

<table>
<thead>
<tr>
<th>Table Scan IMDB.CAST_INFO</th>
<th>Cost: 244E3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IO: 235E3</td>
</tr>
<tr>
<td></td>
<td>CPU: 3.5E10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Scan IMDB.CAST_PERSON</th>
<th>Cost: 26</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IO: 4</td>
</tr>
<tr>
<td></td>
<td>CPU: 174E3</td>
</tr>
</tbody>
</table>
6.5 Effect of Access Paths

- `SELECT c.movie_id`
  `FROM IMDB.CAST_INFO c, IMDB.NAME n`
  `WHERE c.person_id=n.id`
  `AND n.name='Ford, Harrison'`
  `AND n.imdb_index='I'`

Cost: 283E3
IO: 275E3
CPU: 3.6E10

Cost: 48
IO: 7
CPU: 274E3
6.5 Effect of Access Paths

- SELECT `t.title`, `t.production_year`
  FROM `IMDB.TITLE` t, `IMDB.CAST_INFO` c, `IMDB.NAME` n
  WHERE c.person_id = n.id AND n.name = 'Ford, Harrison' AND n.imdb_index = 'I'
  AND t.id = c.movie_id

<table>
<thead>
<tr>
<th>Nummer</th>
<th>Selektivität</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0000008465</td>
</tr>
<tr>
<td>3</td>
<td>0.0595144704</td>
</tr>
<tr>
<td>4</td>
<td>0.0000004632</td>
</tr>
<tr>
<td>5</td>
<td>0.0000004491</td>
</tr>
</tbody>
</table>

Cost: 283E3
IO: 275E3
CPU: 3.6E10

Cost: 64
IO: 9
CPU: 361E3
6.5 Effect of Access Paths

- SELECT t.title, t.production_year
  FROM IMDB.TITLE t, IMDB.CAST_INFO c, IMDB.NAME n, IMDB.KIND_TYPE k
  WHERE c.person_id = n.id AND n.name = 'Ford, Harrison' AND n.imdb_index = 'I'
  AND t.id = c.movie_id AND t.production_year < 1986 AND t.kind_id = k.id AND k.kind = 'movie'

Cost: 283E3
IO: 274E3
CPU: 3.6E10

Cost: 71
IO: 10
CPU: 418E3
For each node in the operator tree, the costs can be estimated:

- Aggregation of costs leads to the total cost of retrieving the result using this specific plan
  - Profit function
- Indexes, etc. have strong influence on the costs
  - Sometimes it pays to create an index for evaluating a single query
6.6 Plan Enumeration

• Dynamically finding the best query plan is easy
  1. Apply transformations to generate all possible plans
  2. Assign total costs according to cost model
  3. Choose least expensive plan

• But this exhaustive search strategy for finding the best plan for each query is prohibitively expensive
  – Actually: not the optimal plan is needed, but the crappy plans have to be avoided
A complete inspection of the search space is hardly possible even for simple queries.

- Some more efficient, but approximate ways of selecting plans have to be used.
- Leads to local minima of cost function, but that is ok.

Typical techniques will be discussed later and include:

- Dynamic programming
- Greedy strategies
- Simulated annealing
- …
6 Query Optimization

Motivating example
Algebraic query rewriting
Execution cost estimation
The SQL EXPLAIN statement
Choosing a plan
Introduction into heuristic query optimization
Simple heuristics commonly used
Heuristics in action
Complex heuristics
Optimizer hints