Relational Database Systems 2
7. Query Optimization II

Wolf-Tilo Balke
Jan-Christoph Kalo
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
7 Query Optimization

7.1 Introduction into heuristic query optimization
7.2 Simple heuristics commonly used
7.3 Heuristics in action
7.4 Complex heuristics
7.5 Optimizer hints
7.1 Introduction

• Remember: **query processor**
• Query optimizers rewrites the naïve (canonical) query plan into a more efficient evaluation plan
  – Relational algebra equivalences allow for creating equivalent plans
7.1 Choosing the Right Plan

• An exhaustive search strategy for finding the best plan for each query is **prohibitively expensive**
  – Always consider the time needed for **evaluating** the query together vs. the time needed for **optimizing** it
  • Total response time consists of both

• Credo for today:
  **Not the optimal plan is needed, but the really crappy plans have to be avoided!**
7.1 Choosing the Right Plan

• We start with a canonical operator tree built from the relational algebra query expression
  – Last lecture: choosing profitable access paths/indexes for each operator in the tree based on cost models
  – This lecture: altering the structure of the tree heuristically to make it more profitable
• **Realistic** cost models are difficult to find…

• But there are some **common assumptions** that can almost always be expected to be beneficial
  – **Example:** keep intermediate results small
    • Better DB buffer utilization
    • Less work for following operators

• **Use heuristics** to improve canonical operator tree step by step
  – Heuristics are based on last lectures **transformation rules** and do not change the results
Optimization heuristics are part of the “magic” within database cores

- Good heuristics are developed during long trial and error processes
- Heuristics are not always equally effective
  - Depends on query profile, data statistics,…
  - May be counterproductive sometimes
- Query Optimizer has to decide when a heuristic pays off and when not
7.2 Simple Heuristics

• The most important heuristics for query optimization
  – Apply selections as early as possible
  – Apply projections as early as possible
  – Avoid Cartesian products
    • or if unavoidable use them as late as possible
  – Use pipelining for adjacent unary operators
7.2 Selections

• Applying a restrictive **selection** operation early keeps the **number** of intermediate results small
  – ‘It is not useful to deal with records that are kicked out of the result at a later stage anyway’
  – Further operations will have to be applied to less records and thus perform faster
  – DB buffer can be used more efficiently
7.2 Selections

• **Break Selections**
  
  – Break up conjunctive select statements
    • Selections are commutative and associative
  
  – Prepares for further optimization by higher degree of freedom

\[
\sigma (\text{condition}_1 \text{ AND } \ldots \text{ AND } \text{condition}_m)
\]
7.2 Selections

• **Push Selections**
  – Change operator sequence to push selects as far down into the tree as possible

• **Remember Relational Algebra equivalences**
7.2 Selections

• Still, pushing selections is only a **heuristic**…
  
  – Assume \( condition_1 \) only removes 1% of records from \( relation_1 \) and has no index, whereas the join condition removes 99% of records and can use an index
  
  – Similar: **expensive predicates** like distance, nearest-neighbor, etc. in spatial DBS
• Applying **projections** early minimizes the **size** of records in intermediate results
  
  • Because tuples get shorter after projection, more of them will fit into a block of the same size
  
  • Hence, the same number of tuples will be contained in a smaller number of blocks
  
  • There are less blocks to be processed by subsequent operations, thus query execution will be faster
7.2 Projections

• “Push Projections”
  – Break up cascading projections, commute them and move them down the tree as deep as possible
  - Condition 1 involves attribute\(_2\)

\[\pi_{(\text{attribute}_1, \text{attribute}_n)}\]
\[\sigma_{\text{condition}_1}\]
relation\(_1\)
relation\(_2\)

\[\pi_{(\text{attribute}_1)}\]
\[\sigma_{\text{condition}_1}\]
\[\pi_{\text{attribute}_n}\]
relation\(_1\)
relation\(_2\)

\[\pi_{(\text{attribute}_1, \text{attribute}_2)}\]
7.2 Cartesian Products

• Cartesian products are among the most expensive operations producing huge intermediate results
  • Often not all combinations from the base relations are needed and selections can be applied
  • Native joins can use specialized algorithms and are usually more efficient than Cartesian products by orders of magnitudes
7.2 Cartesian Products

“Force Joins”

- Replace Cartesian products with matching selections representing a join by explicit join operations
7.2 Hill Climbing

- A **greedy strategy** of applying these simple heuristics can be implemented by a **hill climbing** technique:
  
  - **Input:** canonical query plan
    
    - **Step 1:** Break up all selections
    - **Step 2:** Push selections as far as possible
    - **Step 3:** Break, Eliminate, Push and Introduce Projection. Try to project intermediate result sets as strong as possible.
    - **Step 4:** Collect selections and projections such that between other operators there is only a single block of selections followed by a single block of projections (and no projections followed by selections)
    - **Step 5:** Combine selections and Cartesian products to joins
    - **Step 6:** Prepare pipelining for blocks of unary operators
  
  - **Output:** Improved query plan
7.3 Heuristics in Action

• 3 Relations
  – \( R(A1,A2,A3,Z1) \)
  – \( S(B1,B2,B3,Z2) \)
  – \( T(C1,Z1,Z2) \)

• View

\[
\text{CREATE VIEW V (A1,A3,Z1,B2,B3,C1) AS SELECT A1, A3, R.Z1, B2, B3, C1 FROM R, S, T WHERE R.Z1=T.Z1 AND S.Z2=T.Z2}
\]

• Query

\[
\text{SELECT A1 FROM V WHERE C1 > 199}
\]
1. Break Selection

*Use Algebraic Transform Rule 1*

\[
\sigma_{c_1 \land c_2 \land \ldots \land c_n}(R) \\
\equiv \\
\sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))
\]
7.3 Heuristics in Action

2. Push Selection
Use Algebraic Transform Rules 2,4,8
Place selections as deep into operator tree as possible.
Eliminate superfluous projections.

\[
\sigma_{c_1} (\sigma_{c_2} (R)) \equiv \\
\sigma_{c_2} (\sigma_{c_1} (R))
\]

\[
\pi_{a_1, a_2, \ldots, a_n} (\sigma_c (R)) \equiv \\
\sigma_c (\pi_{a_1, a_2, \ldots, a_n} (R))
\]

\[
\sigma_c (R \bowtie S) \equiv \\
\sigma_{c_1} (\sigma_{c_2} (R)) \bowtie (\sigma_{c_3} (S))
\]
7.3 Heuristics in Action

3. Break, Eliminate, Push and Introduce Projection

Use Algebraic Transform Rule 3,4,9

Break up Projections and push them as far as possible.
Also, remove unnecessary projections and introduce new ones reducing intermediate results.

\[
\pi_{\text{list}_1} (\pi_{\text{list}_2} (\ldots (\pi_{\text{list}_n} (R)) \ldots)) \\
\equiv \pi_{\text{list}_1}
\]

\[
\pi_{a_1, a_2, \ldots a_n} (\sigma_c (R)) \\
\equiv \sigma_c (\pi_{a_1, a_2, \ldots a_n} (R))
\]

\[
\pi_{\text{list}} (R \Join_c S) \\
\equiv (\pi_{\text{list}_1} (R)) \Join_c (\pi_{\text{list}_2} (S))
\]
4. Collect selections and projections such that between other operators there is only a single block of selections followed by a single block of projections (and no projections followed by selections)
7.3 Heuristics in Action

5. Combine selections and Cartesian products to joins

*Use Algebraic Transform Rule 7*

\[
R \bowtie_{c1} S \equiv \sigma_{c1}(R \times S)
\]
6. Prepare pipelining for blocks of unary operators

\[ \pi_{A_1} \therefore R.Z_1 = T.Z_1 \]
\[ \pi_{Z_1} \therefore S.Z_2 = T.Z_2 \]
\[ \sigma_{T.C_1 > 199} \therefore S \]
\[ \therefore T \]
7. Enjoy your optimized query plan

\[
\begin{align*}
\pi_{A1} & \\
\sigma_{C1>199} & \\
\pi_{A1,A3,R.Z1,B2,B3,C1} & \\
\sigma_{S.Z2=T.Z2 \land R.Z1=T.Z1} & \\
\times & \\
\times & \\
R & T & S & S
\end{align*}
\]
7.4 Complex Heuristics

- Simple transformations and hill climbing already lead to a vastly improved operator tree, but more sophisticated heuristics can do even better
  - Special operations
  - View merging
  - Eliminate common sub-expressions
  - Replace uncorrelated sub-queries by joins
  - Sort elimination
  - Dynamic filters
  - Exploit integrity constraints
  - Selectivity reordering
7.4 Complex Heuristics

• Apply **special operations**
  – Provide specialized algorithms for frequently occurring sub-trees / operation patterns
  – Scan operator tree for sub-trees that can be executed by a specialized algorithm
  – Typical examples are **non-standard joins**
    • Semi-joins, anti-joins, nest-joins,…
    • Remember: semi-join $R \bowtie S$ for relations $R$ and $S$ selects all tuples from $R$ that have a *natural* join partner in $S$
View Merging

- A non-materialized view has to be (re-)computed at query time
  - Is really the entire view needed for answering the query?
- If many queries contain views and some additional selections, the view definition can be merged into queries
  - More freedom for the query optimizer
  - Allows for a better plan for evaluation
7.4 View Merging

• Example

  • `CREATE VIEW power AS (SELECT h.secret_ID, s.ability FROM heroes h, superpowers s WHERE h.name = s.hero_ID)`
  
  • `SELECT secret_ID FROM power WHERE ability = 'Mutation'`
7.4 View Merging

- After view merging the selection can be **pushed down** onto table ‘superpowers’

  - SELECT h.secret_ID FROM heroes h, superpowers s
  WHERE h.name = s.hero_ID AND ability = ‘Mutation’
Eliminate common sub-expressions

- Sometimes different operators in query plans need the same input
- The respective expression will be unnecessarily evaluated several times
  - Often simple logical equivalences like DeMorgan’s laws, etc. apply and prevent multiple evaluations of the same condition
  - Intermediate results e.g., from joins can be materialized and used by all following operators
• Example:
  – Logical rewriting:
    • **SELECT * FROM** heroes h, superpowers s **WHERE**
      
      (h.name = s.hero_ID **AND** s.ability = ‘X-ray Vision’) **OR** 
      (h.name = s.hero_ID **AND** s.ability = ‘Invisibility’)

  – Is equivalent to
    • **SELECT * FROM** heroes h, superpowers s **WHERE** h.name = s.hero_ID **AND**
      (s.ability = ‘X-ray Vision’ **OR** s.ability = ‘Invisibility’)

7.4 Common Sub-Expressions
7.4 Sub-Query Flattening

- **Subqueries** are optimized independently of the main query
  - The plan chosen can be suboptimal, because selections cannot be applied early
    - Similar to the case of views
  - The result of the subquery is usually not processed after retrieval
    - Especially duplicate elimination can severely improve performance
• Find all superheroes with improved sight

  – SELECT secret_ID
  FROM heroes
  WHERE name IN ( SELECT hero_ID
                   FROM superpowers
                   WHERE ability LIKE ‘%Vision’ )

  – With the normal execution the matching records from the superpowers table will be scanned for every single row in the heroes table
    • Duplicates in superpowers will be evaluated several times
• But the query can be rewritten into a semijoin

  – `SELECT secret_ID
    FROM heroes h, superpowers s
    WHERE h.name <semijoin> s.hero_ID
    AND s.ability LIKE ‘%vision’`

  – The semijoin will remove duplicates on the fly and usually leads to a severe speed-up in response time

• A rewriting with a regular join is also possible

  – But needs a unique sort operation on the superpowers table to filter out duplicates
7.4 Sort Elimination

• Sorting large result sets is the **most resource intensive** operation in query processing
  – Especially for intermediate results
  – Sort operations are explicitly introduced by SQL constructs like DISTINCT, GROUP BY or ORDER BY
  – Some can be avoided, if sort column
    • Only shows a single value
    • Is retrieved in order
    • Has already been ordered before
• Sort elimination considers the current ordering of intermediate sets before actually executing a sort operator
  – Traversal of a suitable index might already have produced result sets in sorted order
  – Same holds for sort-merge joins
    • When considering join order, performing sort-merge joins as early as possible might lead to better performance
  – Also for example “unique” constraints also often produce ordered result sets
7.4 Dynamic Filters

- **Dynamic filters** are useful whenever operators (like joins or views) are fully computed although a query allows for a restrictive binding
  - Relevant bindings are **dynamically computed** during optimization time of the query
  - Bindings are used as **filters and pushed** as far as possible into the operator tree
  - Often used in stored procedures, etc.
7.4 Dynamic Filters

• Example: Which hero is the uncle/aunt of ‘John’?
  – Based on table: parent(parent, child)
  – Uncles can only be derived via a sibling relationship needing a self-join of the parents table:
    • CREATE VIEW sibling(X,Y) AS
      (SELECT p.child, q.child
       FROM parent p, parent q
       WHERE p.parent = q.parent AND p.child ≠ q.child)
    • CREATE VIEW uncle_aunt(X,Y) AS
      (SELECT p.child, s.X
       FROM sibling s, parent p
       WHERE s.Y = p.parent)
    • SELECT Y FROM uncle_aunt WHERE X = ‘John’
With view merging we can derive the following operator tree:

- The full self join of sibling has to be computed although only some pairs of siblings are relevant.
- s.child='John' can act as a binding for parent p.
7.4 Dynamic Filters

- Idea:
  - create a dynamical filter
    \( F := \pi_{\text{parent}} \sigma_{\text{child}=\text{‘John’}}(\text{parent}) \)
    and apply it already during sibling computation
  - Filter can be applied on parent \( q \) using a semijoin restricting parent \( q \) to records having a child ‘John’
7.4 Dynamic Filters

- Final operator tree
  - Siblings now **only** computes siblings for people having a child ‘John’
  - Intermediate results are **much smaller**
7.4 Semantic Optimization

• **Semantic knowledge** about the data can also be used for optimization tasks
  – Exploit known dependencies and integrity constraints

• Queries can either be
  – Replaced by queries where more conditions derived from the constraint have been added
    • Usually these queries show a higher selectivity
  – Replaced by queries having entirely different conditions given by semantic transformations
    • These queries allow for different access paths
• Example

  – Villains can be divided into ‘rogues’ and ‘supervillains’ and an integrity constraint is that only once you have a secret lair you can be a supervillain, otherwise you are just a rogue

  – Equivalent Queries:
    • SELECT name FROM villains WHERE reputation = ‘supervillain’
    • SELECT name FROM villains WHERE address = ‘secret lair’
• **Selectivity reordering** uses commutativity to rearrange binary operations
  – Most restrictive operations should be applied first
    • May be those with anticipated smallest size, fewest records, …
    • Hard to estimate – Guess or use selectivity statistics
  – Aims especially at reducing intermediate results

• **Most important: join order optimization**
  – Next lecture…
7.4 Selectivity reordering

• Example: correlation in WHERE clauses
  
  – SELECT * FROM villains
    WHERE reputation = ‘supervillain’
    AND income < 50k
  
  • Naïve: simply multiply selectivities of both constraints
  • But will probably not return any rows…

  – Keeping statistics is difficult
  • Number of potential column combinations is exponential
7.4 Selectivity reordering

- Even more severe: transient data
  - Intermediate results stored in a table do not allow for precomputed statistics, but may affect other operators

- Thus, selectivity statistics can change over time and always are incomplete
  - Dynamic sampling (e.g., in ORACLE 10g) supports gathering additional statistics during optimization time
    - Gather a set of samples from all tables involved and test for statistical connections on the fly
• Even the most sophisticated heuristics (as well as cost-based heuristics) can go **wrong**
  – There is no perfect optimizer that is always right
  – Avoiding more mistakes is more costly
    • Trade-Off: sub-optimal query execution vs. optimization time

• What to do, if you know how to evaluate a query but the query optimizer decides for a different plan?
  – **Optimization hints** override the optimizer’s decision
• DB administrators may provide **optimization hints** to override optimizer heuristics and results
  – Uses explain statement’s PLAN_TABLE as INPUT
  – Allow user to specify desired access path to optimizer
  – Design point – support "fallback" to previous access path
  – Experienced / daring users can design their own access path ("what-if" analysis)
7.5 Optimization Hints

• When should optimization hints be used?
  – Temporary fix for badly optimized query plans
  – Access path regresses from previously good path
    • Query planner switched to a worse plan due to
      – Version update
      – Environmental change
      – Statistics update
      – Maintenance Upgrades
    • Manually revert to old plan
7.5 Optimization Hints

– Optimizer unable to find a good plan
  • Might be weakness of optimizer
  • Optimizer needs additional statistics which cannot be provided

– Manually stabilize access path
  • Prevent optimizer of changing plans to guarantee unchanged response times
7.5 Optimization Hints

– Excessive prepare time – prevent optimizer from wasting time

• Repeatedly execute complex dynamic SQL
• Optimal access path is known (e.g., by ‘what-if’ analysis)
• Prepare cost very expensive
  – Complex join can be several minutes
  – Significant CPU / memory consumption
• Provide optimizer hint which is same path that it normally chooses
• Hints are provided by directly modifying the Explain PLAN_TABLE via SQL
  – Powerful, but time consuming and complicated
  – Good DBMS offer tools to graphically provide and validate hints
    • i.e. Visual Hint for DB2, Oracle SQL Developer

• In the following: IBM Optimization Service Center
### 7.5 Optimization Hints

Statement Selection – Can access all explained or cached statements

**Queries List**

Select the query source. Then specify how you want to view the queries by selecting a view. To create a custom view, click New.

- **Query source:** Statement cache
- **View name:** ACCUM_CPU_DESC

#### Advisors

- **Tools**

#### All of the rows in the `ACCUM_CPU_DESC` view view the rows is 9.

<table>
<thead>
<tr>
<th>STMT_ID</th>
<th>ACCESS_PLAN</th>
<th>CPU</th>
<th>STAT_ELAP</th>
<th>STAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3180</td>
<td></td>
<td></td>
<td>0.00940714124955383</td>
<td>12</td>
</tr>
<tr>
<td>3181</td>
<td></td>
<td>3259567157895</td>
<td>0.008941718380825192</td>
<td>19</td>
</tr>
<tr>
<td>3182</td>
<td></td>
<td>0.1665736455835548</td>
<td>0.01665736455835548</td>
<td>31</td>
</tr>
<tr>
<td>3183</td>
<td></td>
<td>0.013845866653508392</td>
<td>0.013845866653508392</td>
<td>18</td>
</tr>
<tr>
<td>4702</td>
<td></td>
<td>0.03636269734002606</td>
<td>0.03636269734002606</td>
<td>60</td>
</tr>
<tr>
<td>4703</td>
<td></td>
<td>0.013225768696443707</td>
<td>0.013225768696443707</td>
<td>20</td>
</tr>
<tr>
<td>4704</td>
<td></td>
<td>0.031796031635181576</td>
<td>0.031796031635181576</td>
<td>30</td>
</tr>
</tbody>
</table>
7.5 Optimization Hints

Overview of the Query Manipulation Interface
7.5 Optimization Hints

Relations and their conditions and interplay
7.5 Optimization Hints

Visual Hint Editor: Each operation in operator tree can be manipulated. Here: Changing access path to a relation.
7.5 Optimization Hints

Join Order Editor: Relations can be reordered...
7.5 Optimization Hints

... and individual joins can be changed.
7 Outlook: Join Order Optimization

- Basic join order optimization
- Join cost and size estimations
- Left-deep join trees
- Dynamic programming
- Greedy strategy
- Randomized algorithms