Übersetzung &
Sichtauflösung
Standardisierung &
Vereinfachung
Optimierung
Plan-
parametrisierung
Code-
Erzeugung
Übersetzungszeit
Laufzeit
SQL-Anfrage
Ergebnis
Algebraterm
Algebraterm
Zugriffsplan
Zugriffsplan
Ausführung
Code
Logische
Optimierung
Physische
Optimierung
Kostenbasierte
Auswahl
Phases of Query Processing

1. **Translation and View Expansion**
   - Simplify arithmetic expressions in the query plan
   - Resolve subqueries
   - Insert the view definition

2. **Logical or algebraic optimization**
   - Transform query plan irrespective of the specific storage form; and pulling in of selections in other operations

3. **Physical or Internal optimization**
   - Take into account concrete storage techniques (indexes, clusters)
   - Select algorithms
   - Several alternative internal plans
Phases of Query Processing (2)

4 Cost-Based Selection
▶ Use statistic information (size of tables, selectivity of attributes) for the selection of a specific internal plan

5 Plan Parametrization
▶ For Pre-compiled queries: (e.g., Embedded-SQL): Replace placeholders with values

6 Code Generation
▶ Convert the access plan into executable code
Phases of Query Processing (3)

- Representation of requests during the processing
  - Algebra expressions $\rightarrow$ **Operator Tree**
    - Operators as Nodes
    - Edges represent data flow
  - Later phases $\rightarrow$ **Access or query plan** (query execution plan – QEP)
    - Concrete algorithms as operators
Logical vs. physical Operators

**Algebra-operatoren**

- $\sigma_F$ (Selektion)
- $\pi_A$ (Projektion)
- $\bowtie$ (Verbund)
- $\gamma_G;\text{aggr}(A)$ (Gruppierung)

**Plan-operatoren**

- IndexScan
- TableScan
- NestedLoopsJoin
- HashJoin
- TableScanProj
- SortProj
- SortMergeJoin
- HashGroupBy
- SortGroupBy

$R, S$ – Relations
$A$ – Attribute Set
$F$ – Condition
$G$ – Grouping Elements
Optimization of Star-Joins

- Star-Joins are a typical pattern for Data Warehouse queries
- Typical properties by the Star-Schema:
  - Very large fact table
  - Clearly smaller, independent dimension tables

⇒ Heuristics of classical relational optimizers often fail in this regard!
Optimization of Star-Joins (2)

- Example: Join over fact table Sales and the three dimension tables Product, Time and Geography:
  - 4-Way Join
  - In RDBMS usually only pairwise Join: Sequence of pairwise joins required
  - 4! possible Join orders
  - Heuristic to reduce the number of combinations to check: Joins between relations that are not linked by a Join condition in the query will not be considered
Optimization of Star-Joins (3)

- Heuristic gives for example the following query plan (Plan A):

```
Verkauf
Zeit
σProduktkategorie='Bier'
⋈
⋈
⋈
σBundesland='Thüringen'
σDatum=201101
σProduktkategorie='Bier'
Produkt
```
Optimization of Star-Joins (4)

- The following query plan (Plan B) is usually not considered (with cross product of the dimension tables):

\[ \sigma_{\text{Produktkategorie}='Bier'} \times \sigma_{\text{Bundesland}='Thüringen'} \times \sigma_{\text{Datum}=201101} \times \sigma_{\text{Zeit}} \times \sigma_{\text{Ort}} \times \text{Verkauf} \]
Star-Join: Calculation example

**Annahmen:**
- Table Sales: 10.000.000 Tuples
- 10 Shops in Thüringen (out of 100 in Germany)
- 20 days of sale in January 2010 (out of 1000 stored days)
- 50 products in the product category "Beer" (out of 1000)
- Uniform distribution / same selectivity of the individual attribute values

<table>
<thead>
<tr>
<th>Plan</th>
<th>Operation</th>
<th>Number of Resulting Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1. Join</td>
<td>1.000.000</td>
</tr>
<tr>
<td></td>
<td>2. Join</td>
<td>20.000</td>
</tr>
<tr>
<td></td>
<td>3. Join</td>
<td>1.000</td>
</tr>
<tr>
<td>B</td>
<td>1. Cross Product</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>2. Cross Product</td>
<td>10.000</td>
</tr>
<tr>
<td></td>
<td>3. Join</td>
<td>1.000</td>
</tr>
</tbody>
</table>
Semi-Join of Dimension Tables

- Calculation of the Cross product for the dimension tables only for sufficiently restrictive selection conditions for dimensions useful
- Avoidance of the complete calculation of the cross product
  ↞ Use of the Semi-Join
Semi-Join of Dimension Tables (2)

1. On the fact table a simple B+-Tree is used for each dimension as an index.
2. Through Semi-Joins with the dimension tables the sets of tuple identifiers (TID) of the potentially relevant tuples is determined.
3. The intersection of those TID sets is computed (e.g., by using efficient main memory methods):
   - Contains all TIDs of the tuples that fulfill all restrictions for all dimensions.
4. After that a "normal" pairwise Join is performed.

Not the whole fact table goes into the join, but instead only the relevant tuples! (in the example: 1.000 instead of 10.000.000 tuples)
Semi-Join of Dimension Tables (3)

TIDs der Faktentabelle

- SemiJoin
  - TableScan(Ort)
  - IndexScan(Verkauf_Ort_IDX, Ort_ID)
- SemiJoin
  - TableScan(Produkt)
  - IndexScan(Verkauf_Produkt_IDX, Produkt_ID)
- SemiJoin
  - TableScan(Zeit)
  - IndexScan(Verkauf_Zeit_IDX, Zeit_ID)
Optimization of the GROUP BY

- Special treatment of grouping and aggregation operations during the optimization

Logical/Algebraic Optimization: "Push-down" of groupings \(\leadsto\) reduction of intermediate result cardinality
  - Invariant Grouping
  - Early Pre-Grouping

Physical/Internal Optimization
  - Special implementation for \texttt{GROUP BY}, \texttt{CUBE} and other OLAP-Functions
Invariant Grouping

- Idea: Shifting a grouping operation "down" (Invariance w.r.t. position)
- Usable if
  - Join partner does not directly contribute to the result (implicit selection)
  - Grouping attributes have the role of a foreign key in the join
- Example:

```sql
SELECT S_Time_ID, S_Location_ID, SUM(Revenue)
FROM Sales, Time, Location
WHERE S_Time_ID=T_ID AND
  S_Location_ID=L_ID
  AND Year < 2010 AND County <> "THÜR"
GROUP BY S_Time_ID, S_Location_ID
```
Invariant Grouping (2)

A, Z ∈ R
B, Y ∈ S

\( Y_{A;F(Z)} \)
\( \sigma_{P(Y)} \)
\( \bowtie \)
\( R.A = S.B \)

\( \pi_B \)

\( \sigma_{P(Y)} \)

© Sattler / Saake / Köppen
Data Warehouse Technologies
Last Change: 06.02.2020
Early Pre-Grouping

- Invariant Grouping: restriktive precondition $\rightsquigarrow$ seldom used
- Idea: Insertion of an additional grouping operator before the join (similar to a projection)
- Usable if
  - Grouping condition contains the join attributes
  - Aggregated attributes do not depend on the attributes of the join partner
Early Pre-Grouping (2)

Example:

```sql
SELECT Year, L_City, SUM(Revenue), COUNT(Revenue)
FROM Sales, Time, Location
WHERE S_Time_ID = T_ID AND
    S_L_ID = L_ID
GROUP BY Year, L_City
```
Early Pre-Grouping (3)

\[
A, C, Z \in R \\
B, D \in S
\]

\[
R \bowtie S
\]

\[
\gamma_{C,D;F(Z)}
\]

\[
R.A = S.B
\]

\[
R \bowtie S
\]

\[
\gamma_{A,C;F(Z)}
\]

\[
A, C, Z \bowtie R
\\
B, D \bowtie S
\]

\[
\gamma_{C,D;F(Z)}
\]
Early Pre-Grouping (4)

- Also required: Adjustment of the aggregation function
Implementation of the grouping operator

- Implementation of GROUP BY
- Implementation of OLAP-Functions
- Computation of the CUBE
- Iceberg-Cubes
GROUP BY-Implementations

- **Sort-based**
  - Pre-sorting the relation or sorting read (Index-Scan)
  - Process
    1. Sorting
    2. Iteration over tuples
    3. Aggregation of the values and output of the aggregated value in case of a group change

- **Hash-based**
  - Hashfunction over grouping attributes $h(G_1, \ldots, G_n)$
  - Process
    1. Insertion of tuples in hash tables using $h(G_1, \ldots, G_n)$
    2. Iteration through hash table
    3. Application of aggregation functions
Implementation of OLAP-Functions

- Sequential evaluation
- For each OLAP-Function:
  - Input data sorted according to \texttt{OVER()-clause}
  - Apply aggregation functions
- Sorting by
  - Attributes of the global grouping
  - Attributes of the \texttt{OVER()-clause (PARTITION BY and ORDER BY)}
- In case of multiple OLAP-Functions in a query
  - Sequential evaluation, i.e., possibly repeated sorting for usage of shared sorting prefixes
Implementation of OLAP-Functions (2)

- Global grouping attributes \( G_1 \ldots G_n \),
- Locale sorting attributes of \( \text{OVER}(\cdot) \): \( O_1 \ldots O_p \),
- Aggregation function \( \text{AGG}(\cdot) \),
- Locale partitioning attributed (opt.) \( P_1 \ldots P_m \),
- Lower and upper window border (opt.) \( W_u \ldots W_o \).

\[
\text{sort}(G_1, \ldots, G_n, P_1, \ldots, P_m, O_1, \ldots, O_p); \\
\text{while} \ (t = \text{next\_tuple}()) \ {\{ \\
\quad \text{if} \ (t \ \text{has equal values w.r.t.} \ G_1 \ldots G_n, P_1 \ldots P_m \ \text{like last tuple}) \\
\quad \text{aggrlist} := \text{concat}(\text{aggrlist}, \ t); \\
\quad \text{else} \ {\{ \\
\quad \quad \text{// Partition switch} \\
\quad \quad \text{for} \ i := 1 \ \text{to} \ \text{length(\text{aggrlist})} \ {\{ \\
\quad \quad \quad \text{// Compute absolute window borders low, high} \\
\quad \quad \quad \text{aggrval} := \text{AGG}([\text{aggrlist}[\text{low}]\ldots\text{aggrlist}[\text{high}]]); \\
\quad \quad \quad \text{output}(G_1, \ldots, G_n, \ P_1, \ldots, P_m, \ O_1, \ldots, O_p, \ \text{aggrval}); \\
\quad \quad \}} \\
\quad \text{aggrlist} := (); \\
\quad \}} \\
\}} \\
\}
\]
Calculation of the CUBE: naive Approach

- Separate calculation of all grouping combinations
- Final unification
CUBE and Aggregation functions

- Algebraic functions enable
  - Calculation of less detailed aggregates from more detailed aggregates (more dimensions)
  - Partial order ("grid") of the GROUP BY operations of the CUBE
    - Data Cube Lattice
  - GROUP BY is a child of another GROUP BY if the parent operation can be used to calculate the child operation → Derivability
Derivability

- Derivability of grouping combinations $G_i$
- Direct Derivability:
  - $G_2$ is derivable from $G_1$ if
  - $G_2$ has exactly one attribute less than $G_1$: $G_2 \subset G_1$ and $|G_2| = |G_1| - 1$
  - or in $G_1$ exactly one attribute $A_i$ is replaced by $B_i$ where the following holds true: $A_i \rightarrow B_i$

⇒ Data Cube Lattice

- Derivability:
  - Grouping combinations: $G_2$ is within a data cube lattice derivable from $G_1$ when there is a path from $G_1$ to $G_2$
Data Cube Lattice

- Produktgruppe (P)
- Bundesland (B)
- Jahr (J)

Ebene:

- 0
- 1
- 2
- 3
Computation of the CUBE

- Idea:
  - Using of the grid view (Derivabiity)
  - GROUP-BYs with common grouping attributes can share partitions, sorted parts etc.

- Appriach
  - Based on sorting: PipeSort
  - Based on hashing: PipeHash
Optimized Computation: Principle
Optimization Potential

- Smallest-parent
  - Computation of a GROUP-BY based on the minimal previously computed Parent-GROUP-BY

- Cache-results
  - Temporary storage of the results (in the main memory) of a GROUP-BY for subsequent GROUP-BYs (Example.: $ABC \rightarrow AB$)

- Amortize-scans
  - Joint calculation of multiple GROUP-BYs in a scan (Example: $ABC$, $ACD$, $ABD$, $BCD$ aus $ABCD$)

- Share-sorts
  - For sort-based approaches: Temporary storage and shared use of sorted parts

- Share-partitions
  - For hash-based approaches: Temporary storage and joint use of partitions
Meaning of the Sorting Order

- Assumption: Grouping based on sorting
  - potentially requires re-sorting

- Example:

<table>
<thead>
<tr>
<th>Product</th>
<th>Year</th>
<th>Region</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotwein</td>
<td>2009</td>
<td>SANH</td>
<td>230</td>
</tr>
<tr>
<td>Rotwein</td>
<td>2009</td>
<td>THÜR</td>
<td>210</td>
</tr>
<tr>
<td>Rotwein</td>
<td>2010</td>
<td>SANH</td>
<td>200</td>
</tr>
<tr>
<td>Bier</td>
<td>2009</td>
<td>...</td>
<td>568</td>
</tr>
</tbody>
</table>

- Re-sorting for the grouping (Product, Region)

- Consideration of the sort costs in a cost model
Cost Model

- Cost Types
  - **S-Costs** (Still to sort): Calculation of GROUP-BY \(j\) from GROUP-BY \(i\), if \(i\) not sorted yet
  - **A-Costs** (Already sorted): Calculation of GROUP-BY \(j\) from GROUP-BY \(i\), if \(i\) already sorted

- Estimation based on data distribution, system parameters, etc.
PipeSort

- **Input: Search grid**
  - Graph with nodes to represent GROUP-BY (Aggregationsgitter)
  - Directed edge connects GROUP-BY \( i \) with GROUP-BY \( j \)
    - \( i \) is parent node of \( j \)
    - \( j \) can be generated from \( i \)
    - \( j \) has exactly one attribute less than \( i \)
  - Level \( k \) refers to all GROUP-BYs with \( k \) attributes

- **Annotations of edges** \( e_{ij} \) with A- and S-Costs

- **Output: Subgraph of the search grid**
  - Each node is connected to a single parent node
  - Determines sorting order while preserving pipelining
    - Special expanded tree

- **Goal: Subgraph with minimal summe of edge costs**
PipeSort: Example

A → B → C
AB → AC → BC
2 → 5 → 13 → 20
1 → 12 → 13 → 20

Pipeline Sortierung

A → B → C
AB → AC → BC
2 → 5 → 13 → 20
1 → 12 → 13 → 20

Calculation of the CUBE
PipeSort: Algorithm

- Can be traced back to a known Graph-Algorithm
- Level-wise approach: $k = 0..N - 1$ ($N$: Number of attributes)
- Transformation of level $k + 1$ by $k$ copies of each node
- Each copied node has connections with the same node as the original
- Edge costs for original node: $A(e_{ij})$, otherwise: $S(e_{ij})$
- Search for graph with minimal costs
  - Forming of pairs and minimization of the total costs ("hungarian" method – *weighted bipartite matching problem*)
PipeSort: Sorting Order

- Each node $h$ in level $k$ is connected to a node $g$ in level $k + 1$
- $h \rightarrow g$ over $A()$-edge: $h$ determines attribute order for sorting $g$
- $h \rightarrow g$ over $S()$-edge: $g$ is re-sorted for the calculation of $h$
PipeSort: Sort Plan

Relation

Pipeline   Sortierung
Iceberg Cubes

- Idea: Exploitation of the monotony of aggregations

If an aggregation group does not fulfill the \texttt{COUNT}-condition, then this condition is also not fulfilled by groups with additional attributes.

- Approach: Bottom-Up-ConstruCtion of a cube and Minimal-Support-Pruning (similar to Apriori)
Iceberg Cube bottom up
Calculation of the CUBE

Iceberg-Cube: Computation

BottomUpCube(input, dim):
  aggregate(input);
  write(outputRec);
  for (d:=dim; d<numDims; d++) {
    C := cardinality[d]; /* Cardinality of the dimension */
    Partition(input, d, C, dataCount[d]);
    k := 0;
    for (i:=0; i < C; i++) {
      c := dataCount[d][i]; /* Size of Partition */
      if (c >= minsup) {
        outputRec.dim[d] := input[k].dim[d];
        BottomUpCube(input[k...k+c], d+1);
      }
      k += c;
    }
    outputRec.dim[d] = ALL;
  }
Summary

- Special characteristics of DW queries require specific optimization methods
- Rewriting techniques:
  - Join order for Star-Join
  - Push-down of groupings
- Operator implementation
  - CUBE and Iceberg-CUBE
  - OLAP-Functions