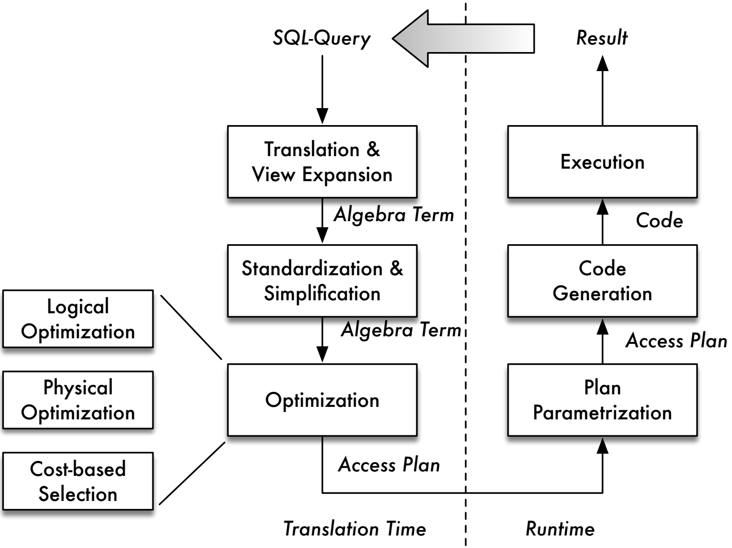
Part VIII

Query Processing and Optimization

Query Processing and Optimization

Overview

Overview


SQL-Query
Result
Translation & View Expansion
Execution
Algebra Term
Code
Standardization & Simplification
Code Generation
Logical Optimization
Algebra Term
Access Plan
Physical Optimization
Optimization
Plan Parametrization
Cost-based Selection
Access Plan
Translation Time
Runtime

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

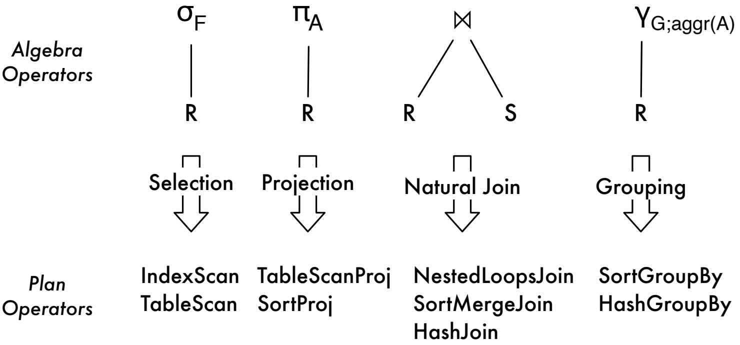
Phases of Query Processing (2)

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 $\longrightarrow$ **Operator Tree**
    - Operators as Nodes
    - Edges represent data flow
  - Later phases $\longrightarrow$ **Access or query plan** (query execution plan – QEP)
    - Concrete algorithms as operators

Logical vs. physical Operators



- R, S – Relations
- A – Attribute Set
- F – Condition
- G – Grouping Elements

Star-Join

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

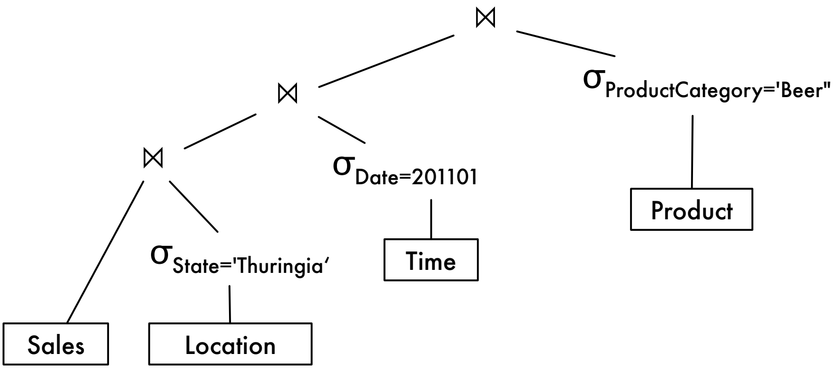
$\Rightarrow$ Heuristics of classical relational optimizers often fail in this regard!

Optimization of Star-Joins (2)

- Example: Join over fact table `Sales` and the three dimension tables `Product`, `Time` und `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 connected by a join condition in the query will not be considered

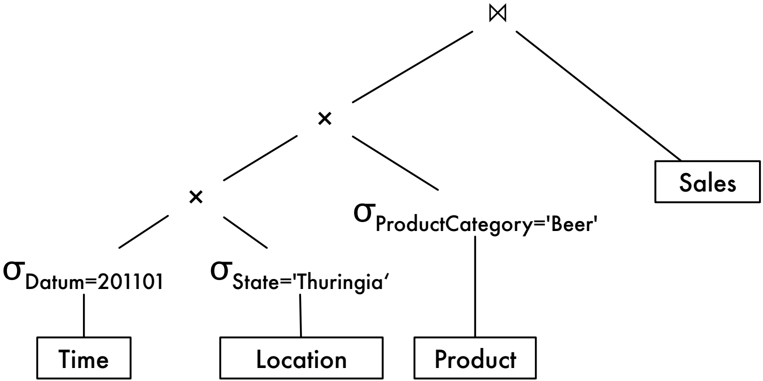
Optimization of Star-Joins (3)

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



Optimization of Star-Joins (4)

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



Star-Join: Calculation example

- Annahmen:
  - Table Sales: 10.000.000 Tuples
  - 10 Shops in Thuringia (out of 1000 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

Plan	Operation	Number of Resulting Tuples
A	1. Join	100.000
	2. Join	20.000
	3. Join	1.000
B	1. Cross Product	200
	2. Cross Product	10.000
	3. Join	1.000

Semi-Join of Dimension Tables

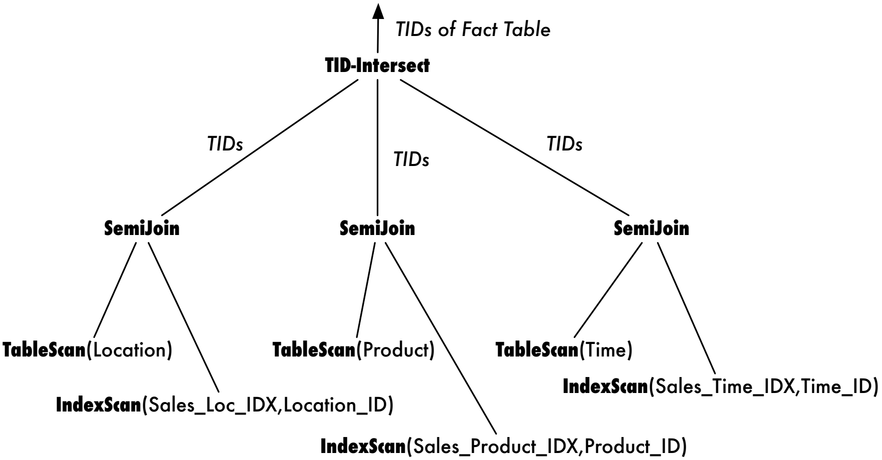
- Calculation of the Cross product for the dimension tables only for sufficiently restictive selection conditions for dimensions useful
- Avoidance of the complete calculation of the cross product
  $\leadsto$ Use of the Semi-Join

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)



Optimization of the GROUP BY

Optimization of the GROUP BY

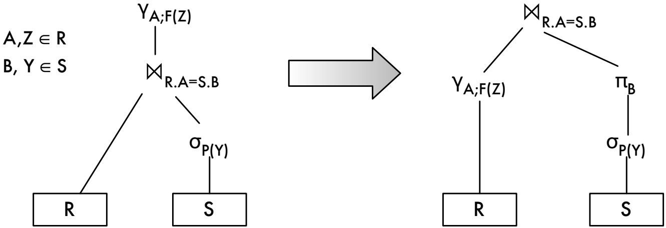
- Special treatment of grouping and aggregation operations during the optimization
- Logical/Algebraic Optimization: "Push-down" of groupings $\rightsquigarrow$ reduction of intermediate result cardinality
  - Invariant Grouping
  - Early Pre-Grouping
- Physical/Internal Optimization
  - Special implementation for **`GROUP BY`**, **`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:

```
SELECT S_Time_ID, S_Location_ID, SUM(Turnover)
FROM Sales, Time, Location
WHERE S_Time_ID=T_ID AND
   S_Location_ID=L_ID
   AND Year < 2010 AND State <> "THUR"
GROUP BY S_Time_ID, S_Location_ID
```

Invariant Grouping (2)



Early Pre-Grouping

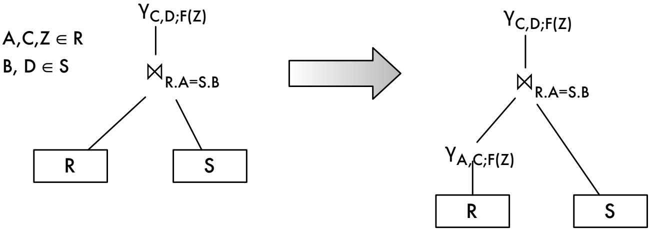
- Invariant Grouping: restriktive precondition $\rightsquigarrow$ seldom used
- Idea: Insertion of an additional grouping operator before the join (similar to a projection)
- Usable if
  - Grouping condition contains the join attributes
  - Aggregated attributes do not depend on the attributes of the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

Early Pre-Grouping (2)

- Example:

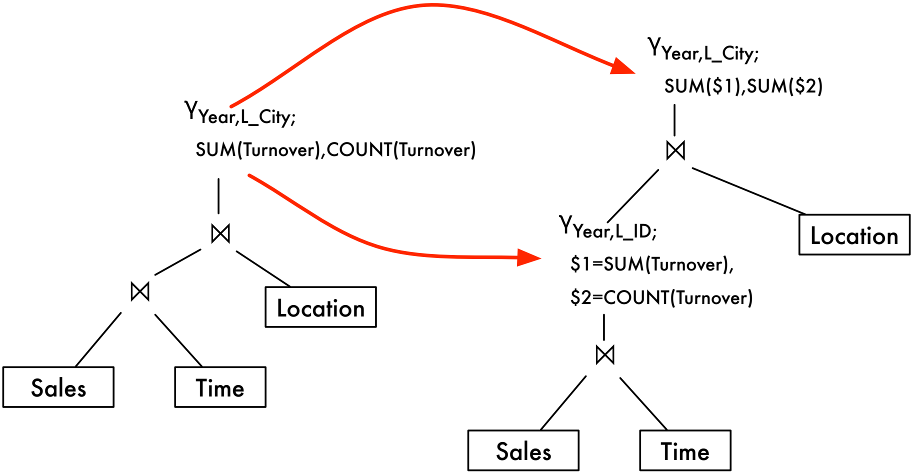
```
SELECT Year, L_City, SUM(Turnover), COUNT(Turnover)
FROM Sales, Time, Location
WHERE S_Time_ID = T_ID AND
   S_L_ID = L_ID
GROUP BY Year, L_City
```

Early Pre-Grouping (3)



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. Sortinge
    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 **OVER**()-clause
  - Apply aggregation functions
- Sorting by
  - Attributes of the global grouping
  - Attributes of the **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)

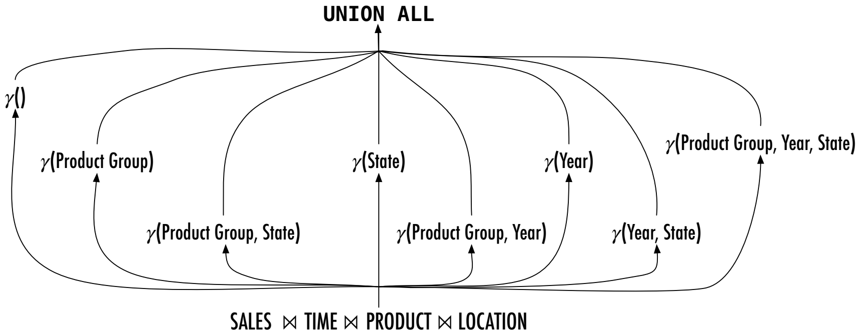
global grouping attributes $G_1 \ldots G_n$,
locale sorting attributes of **OVER**(): $O_1 \ldots O_p$
aggregation function **AGG**(),
locale partitioning attributed (opt.) $P_1 \ldots P_m$,
lower and upper window border (opt.) $W_u \ldots W_o$

```
sort(G1,...,Gn,P1,...,Pm,O1,...,Op);
while (t = next_tuple()) {
    if (t has equal values w.r.t. G1...Gn,P1...Pm like last tuple)
        aggrlist := concat(aggrlist, t);
    else {
        // Partition switch
        for i := 1 to length(aggrlist) {
            // Compute absolute window borders low, high
            aggrval := AGG({aggrlist[low]...aggrlist[high]});
            output(G1,...,Gn, P1,...,Pm, O1,...,Op, aggrval);
        }
        aggrlist := ();
    }
}
```

Calculation of the CUBE

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 aggegates 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

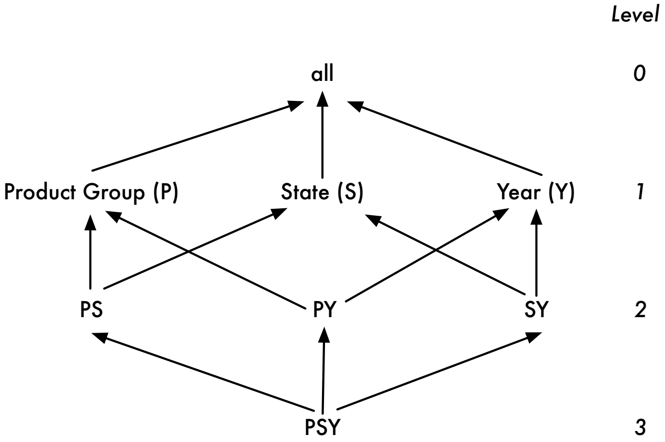
Derivability

- 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: $A_i \rightarrow B_i$

$\Rightarrow$ **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

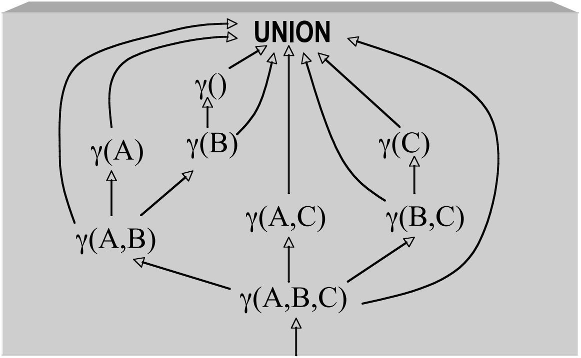
Data Cube Lattice



Computation of the CUBE

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

Product	Year	State	Sales
RedWine	2009	SANH	230
RedWine	2009	THUR	210
RedWine	2010	SANH	200
...	...	...	...
Beer	2009	SANH	568

  - Re-sorting for the grouping (Product, State)
- Consideration of the sort costs in a cost model

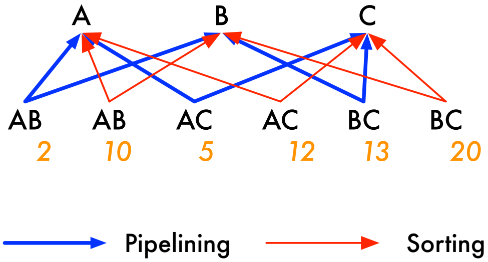
Cost Model

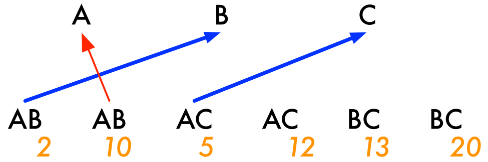
- Cost Types
  - *S-Costs* (Still to sort): Calculation of GROUP-BY j from GROUP-BY i, if i not sorted yet
  - *A-Costs* (Already sorted): Calculation of GROUP-BY j from GROUP-BY i, if $i$already sorted
- Estimation based on data distribution, system parameters, etc.

PipeSort

- Input: Search grid
  - Graph with nodes to represent GROUP-BY (Data Cube Lattice)
  - 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





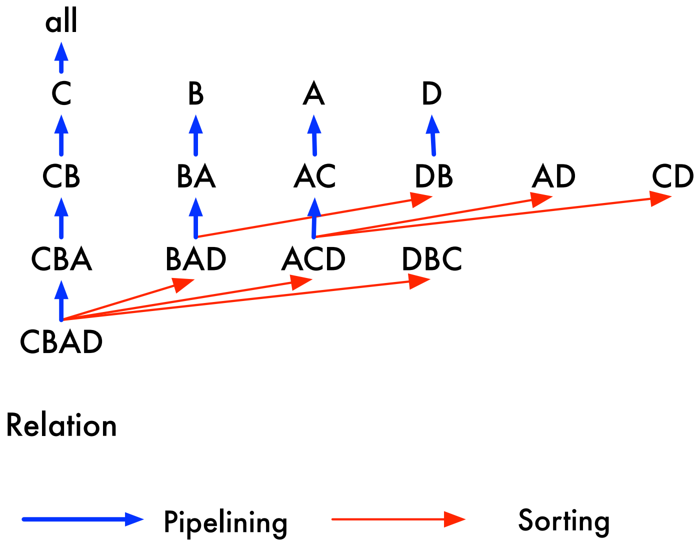
PipeSort: Algorithm

- 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



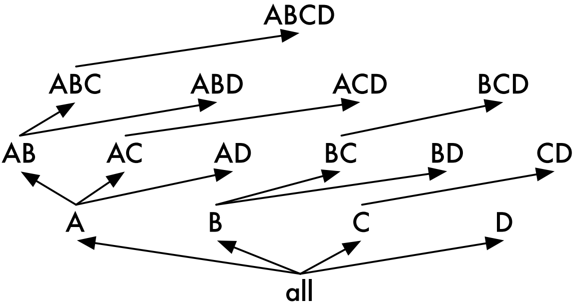
Iceberg Cubes

- Idea: Exploitation of the monotony of aggregations

If an aggregation group does not fulfill the **`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



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 charakterics 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