Dresden Database Systems Group

9. Relational Plan Operators

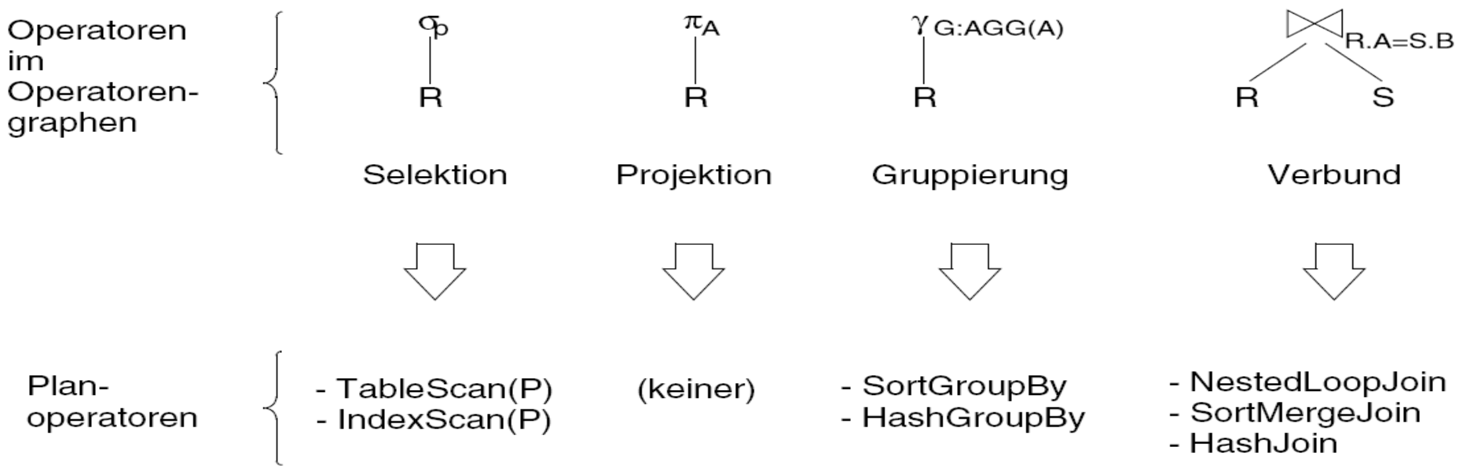
Architecture of Database Systems

Plan Operators

Dresden Database Systems Group

WHAT ARE PLAN OPERATORS?

- physically executable operators
- physical realization of the logical operators originating in the Relational Algebra
- basis of the query execution plan



Structure

Unary operators

- selection and projection
- grouping
- aggregation
- sorting

Join Operators (Binary Operators)

- join operators
- nested-loop join
- sort-merge join
- hash-join
  - comparison
  - data-Skew

Unary Operators

Selection and Projection

Plan operators for projection

- column elimination is trivial
  - is typically carried out in combination with sorting, selection or joining
  - preceding operator is only providing required columns
- duplicate elimination is realized by grouping on all distinct attributes without additional aggregation

Plan operators for selection

- use of scan operators
  - definition of start and stop condition
  - definition of simple search arguments
- relation scan / table scan
- index scan
  - selection of the most cost-effective index

Selection and Projection (3)

```
SELECT *
  FROM Turnover
 WHERE Month BETWEEN 1 AND 6
```

Example

- implementation by table scan

```
currentScanID := open-rel-scan(Turnover-RelationID);
currentTID := next-TID(currentScanID);
while (not end-of-scan(currentScanID))
   currentTuple := fetch-tuple(Persons-RelationID, currentTID);
   if currentTuple.Month >= 1 and currentTuple.Month <= 6
      return(currentTuple);
   currentTID := next-TID(currentScanID);
close-scan (currentScanID);
```

- implementation by index scan

```
currentScanID := open-index-scan(turnover-Month-IndexID, 1, 6);
currentTID := next-TID(currentScanID);
while (not end-of-scan(currentScanID))
   currentTuple := fetch-tuple(turnover-RelationID, currentTID);
   return(currentTuple);
   currentTID := next-TID(currentScanID);
close-scan (currentScanID);
```

Problems with the Scan Application

Dresden Database Systems Group

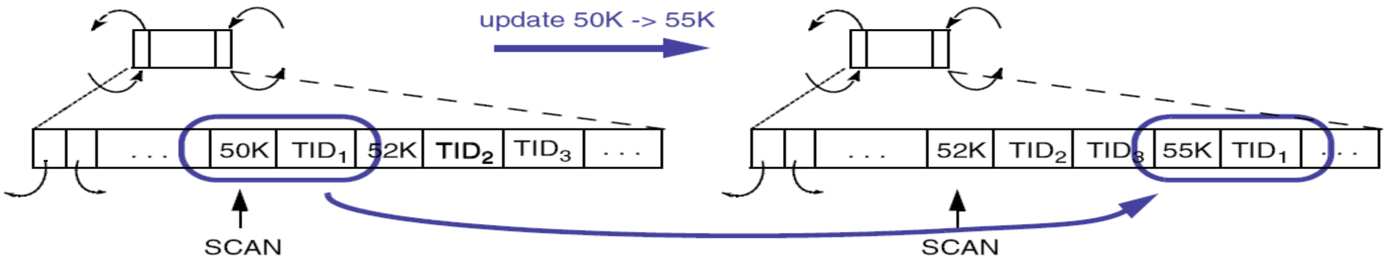
Situation

- declarative statement (SQL statement) is evaluated in a record-oriented manner
- conflict, if an object to be updated is used by scan

Example

```
UPDATE EMP SET SALARY = SALARY * 1.1 [WHERE SALARY BETWEEN 45K AND 70K]
```

- query optimization decides that an existing index on salary is used to execute this statement



HALLOWEEN-Problem !!! P.S.: Woher stammt der Name?

Grouping

Logical processing order

Verbund-operationen	→	Selektion	→	Gruppierung	→	Aggregation	→	Selektion	→	Sortierung
FROM & WHERE		WHERE		GROUP BY		SELECT		HAVING		ORDER BY

Hash-based approach

- applying a hash function to the values of the grouping attributes
- hash table keeps results of aggregation functions per combination

Sort-based approach

- sort on all grouping attributes (any preferred sort order?)
- in the sorted data stream, tuples with equivalent values of the grouping attributes, that is, all tuples of a group, are located one after the other
- data stream is read in and aggregation values are formed up to the respective next value change within the group attributes

Grouping-Algorithm: Sort-based approach

Dresden Database Systems Group

```
Input:G1,...,Gn // Grouping attributes from the GROUP BY clause
      AGG(), A // Aggregation function and attribute A to be aggregated
Begin
   // Sort the data stream by the grouping attributes
   SORT(G1,...,Gn)
   // Processing the entire input stream
   While (Input current not yet processed)
      ($g1,...,$gn,$val) := ReadNextTuple(InputStream)
      // Within the same group the entries are aggregated.
      If (Current tuple has the same values in G1,...,Gn as last tuple)
         $aggrset := $aggrset ∪ {$val};
      Else
         // When changing a group (or at the end), an output tuple is generated
         //   by the aggregation function over the set of values
         $aggrval = AGG($aggrset)
         WriteNewTuples(OutputStream, ($g1,...,$gn, $aggrval))
         $aggrset := {};
      End If
   End While
End
```

Aggregation

Dresden Database Systems Group

PRINCIPLE

- similar to the projection
  - grouping corresponds to projection with aggregation Q(R) be the attribute of R to which an aggregation function is to be applied

MIN, MAX

- parallel calculation is always possible
- MIN(Q(R)) => MIN (MIN (Q(R_1)), ... , MIN (Q(R_n)))
- MAX(Q(R)) => MAX (MAX (Q(R_1)), ... , MAX (Q(R_n)))
- parallel calculation of the local minima/maxima

SUM, COUNT, AVG

- only applicable where duplicate elimination is not required
- SUM (Q(R)) => Σ SUM (Q(R_i))
- COUNT (Q(R)) => Σ COUNT (Q(R_i))
- AVG (Q(R)) => SUM (Q(R)) / COUNT (Q(R))

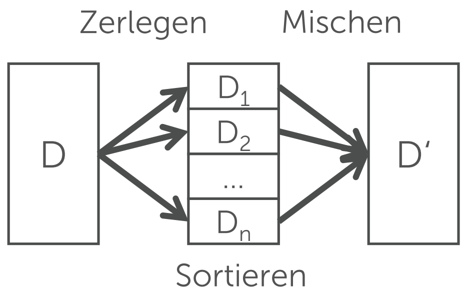
Sorting

If possible sort by index scan

- index on the sorting attribute
- easy reading of the index
- no explicit sorting required
- additional attributes may still have to be loaded (FETCH)

General: External sorting

- decomposition of the input into several runs
- sorting and buffering the sorted runs
- successively mix until a sorted run is created
- block size of the size of the available memory
- adjust data into the working memory, eliminates the mixing



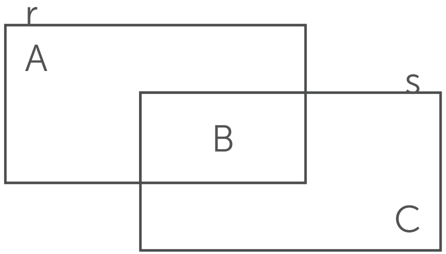
Join Operators (Binary Operators)

Join Operators

Binary Operators

- match in all attributes
  - A: difference r–s
  - B: intersection r ∩ s
  - C: difference s–r
  - A ∪ C: symmetric difference (r–s)∪(s–r)
  - A ∪ B ∪ C: union r ∪ s

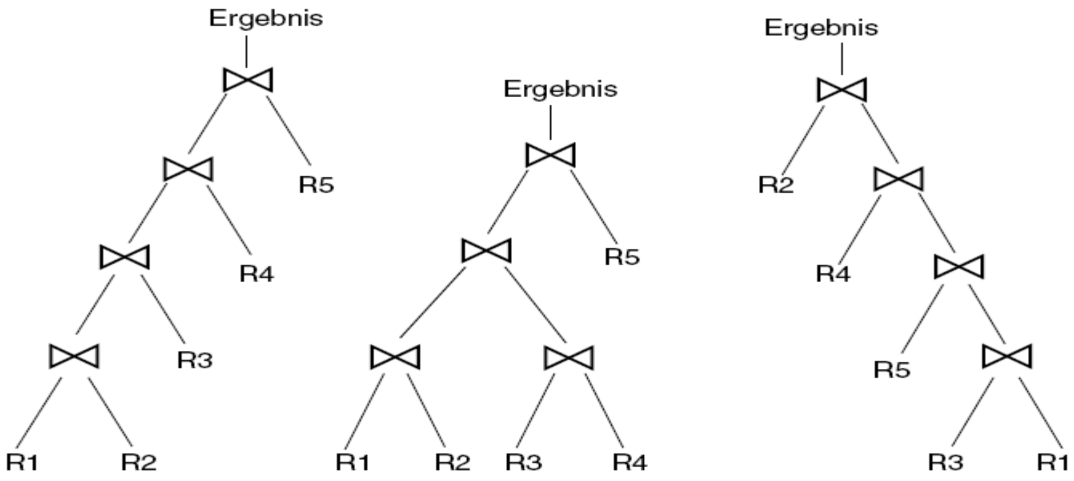
- match on some attributes
  - A: Left-sided anti-semi-join (linksseitig Anti-Semi-Verbund)
  - B: join
  - C: Anti-semi-join on the right-hand side (rechtsseitig Anti-Semi-Verbund)
  - A ∪ B: Left-Outer-Join (linksseitig äußerer Verbund)
  - A ∪ C: Anti-join
  - B ∪ C: Right-Outer-Join (rechtsseitig äußerer Verbund)
  - A ∪ B ∪ C: Full-Outer-Join (vollständig äußerer Verbund)



Join Operators (2)

JOIN VIA SEVERAL RELATIONS (N-WAY JOIN)

- decomposition into n-1 two-way joins
- number of join sequences depends on the selected join attributes
- n! different sequences possible
- optimal evaluation order dependent on
  - plan operators
  - “matching" sort order for join attributes
  - size of the operands, etc.
- different join series with two-way joins (example: n = 5)



Join Operators (3)

Properties of the Join Operation

- expensive and frequent -> optimization candidate!
- typical: equality join; general join predicate rather rare
- standard scenario

```
SELECT *
FROM R, S
WHERE R.JA Θ S.JA          // join predicate
AND P(R.SA)                // local selection
AND P(S.SA)
```

Possible Access Paths

- table scans via R and S
- scans via I_R(R.JA) and I_S(S.JA)
  - sorting order to R.JA and S.JA !!!
- scans via I_R(R.SA) and / or I_S(S.SA)
- fast selection for R.SA and S.SA !!!
- ... any other combinations

Nested-Loop Join

Dresden Database Systems Group

Assumptions

- records in R and S are not ordered according to the join attributes
- there are no index structures $I_R(JA)$ and $I_S(JA)$

Algorithm for Θ-Join

```
Scan via S
for every record s, if P_S applies:
   Scan via R
   for every record r, if P_R AND (r.JA Θ s.JA) apply:
      take combined record (r||s) into the result
```

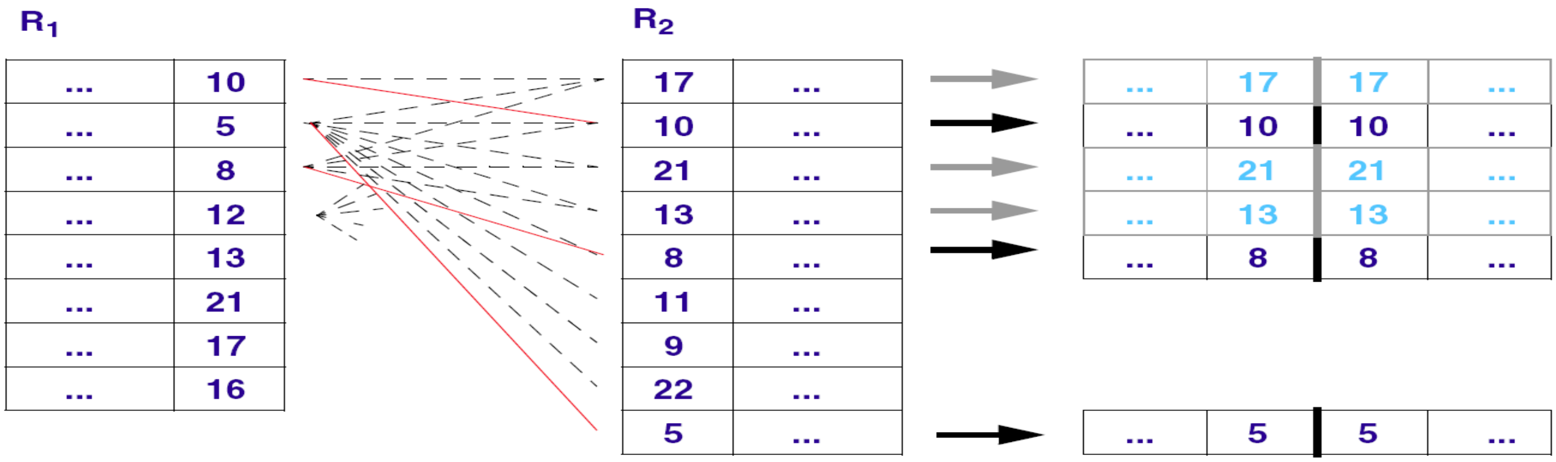
Complexity

- $O(N^2)$

Nested-Loop Join(2)

Dresden Database Systems Group

Example



- assumption: Foreign key/primary key relationship!

Nested-Loop Join with Index Access

Dresden Database Systems Group

Assumptions

- there are index structures $I_R(JA)$ and $I_S(JA)$

Algorithm for Θ-Join with index access

```
Scan via S
for every record s, if P_S applies:
   Use I_R(JA) to find all TIDs for records with r.JA = s.JA
   for each TID:
   get record r, if P_R applies:
      take combined record (r||s) into the result
```

Note

- usually block or page-wise approach, which contradicts the idea of a layered architecture
- the same principle is used to implement set operations

Sort-Merge Join

Dresden Database Systems Group

Assumptions

- there are index structures I_R(JA) and I_S(JA)

Algorithm for exploiting index structures I_R(R.JA) and I_S(S.JA)

- phase 1:
  sorting of R and S according to R.JA and S.JA (if not already present), early elimination of unneeded tuples (by checking P_R, P_S)
- phase 2:
  step-holding scans over assorted R- and S-relations with execution of the join at r.JA = s.JA
- pseudocode:

```
Step-by-step scans via I_R(JA) and I_S(JA):
for each two keys from I_R(JA) and I_S(JA), if r.JA = s.JA:
   fetch tuples for corresponding TIDs, if P_R and P_S apply:
      take the combined record (r||s) into the result
```

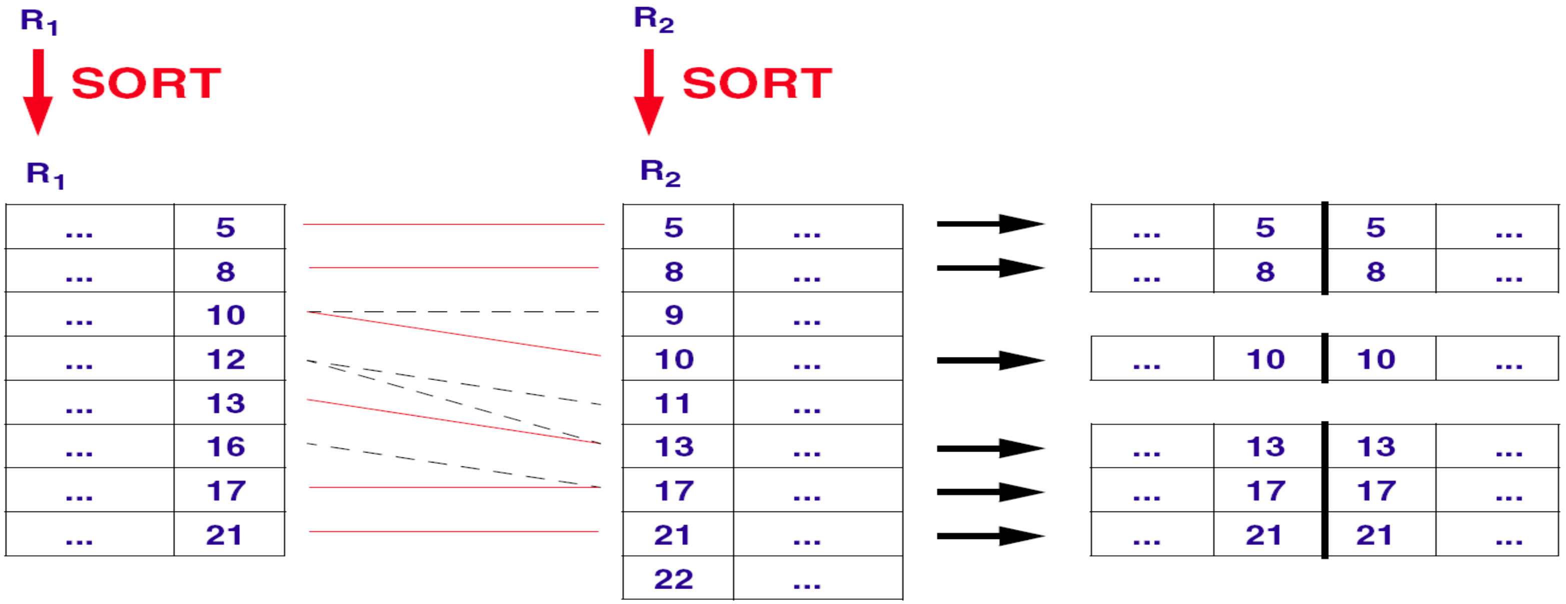
Complexity

- O(N log N)

Sort-Merge Join(2)

Dresden Database Systems Group

Example



Hash Join– Classic Hashing

Step 1

- step-by-step reading the (smaller) table R
- composition of a hash table with $h_A(r(JA))$ according to values of R(JA)
- dividing into p sections R_i ($1 \leq i \leq p$) such, that
  - each of the p sections fits into the available main storage
  - P_R applies to every record that has been hashed

Step 2

- probing for each set of S with P_S
- in case of success: implementation of join, i.e. concatenating the tuples

Step 3

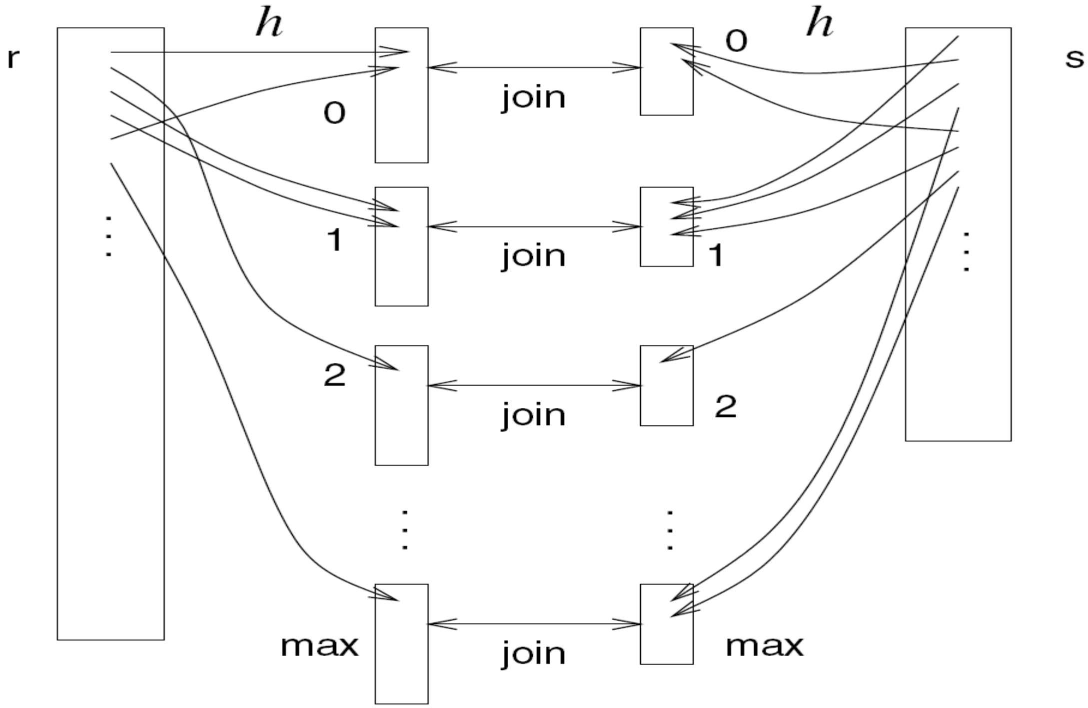
- repeat steps 1 and 2 until all p sections are processed

Complexity

- O(p * N)
- ideal case: R fits into the main memory (or processor cache), (P = 1)

Hash Join– Classic Hashing (2)

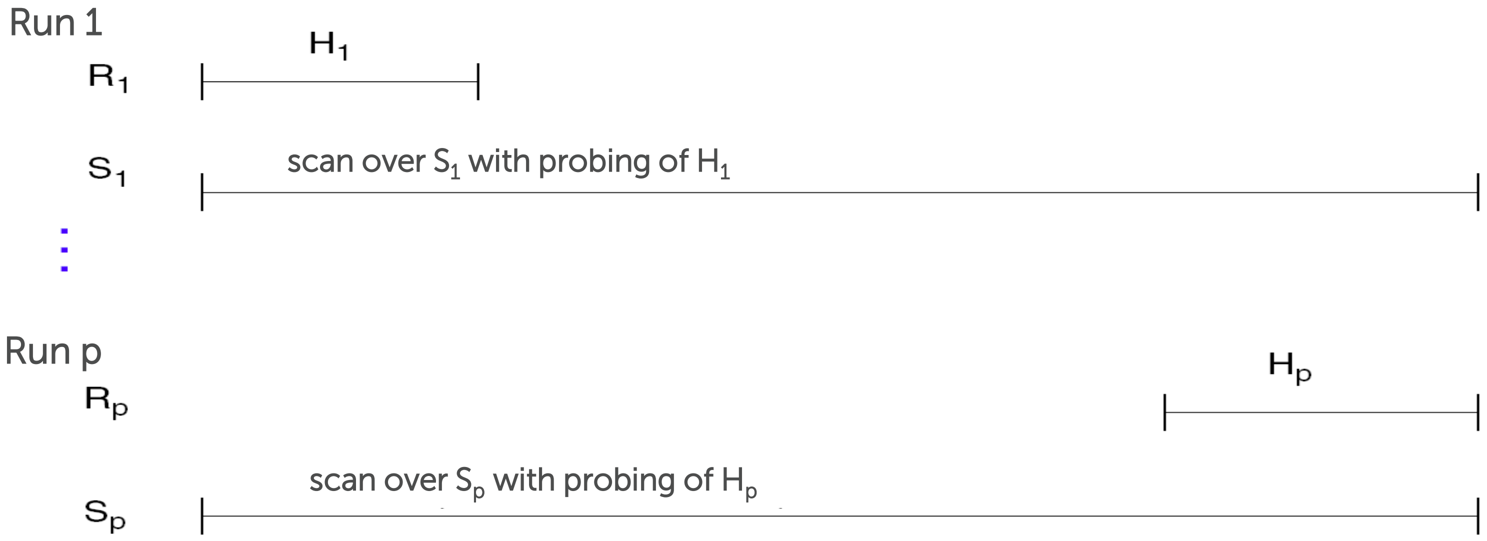
ILLUSTRATION OF PARTITIONING



Hash Join– Classic Hashing (3)

STRUCTURE OF THE HASH TABLE AND PROBING

- hash tables $H_i (1 \leq i \leq p)$ are built up step-by-step in main memory
- after each run of S, the current hash table is deleted

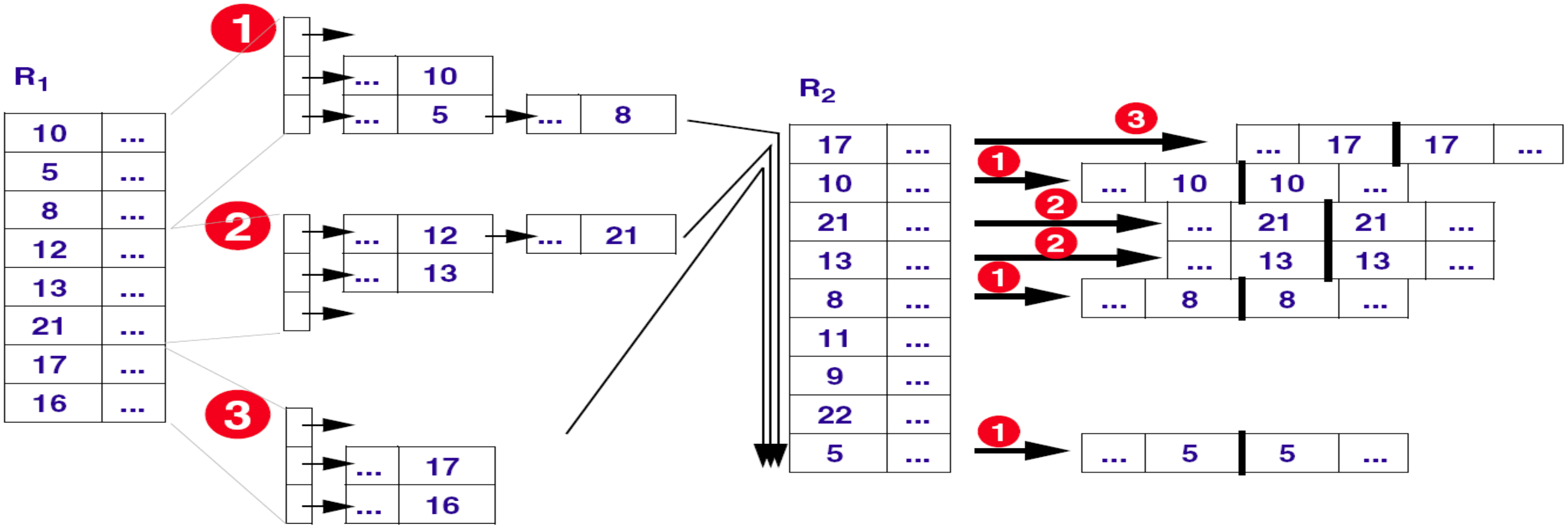


Hash Join – Classic Hashing (4)

Dresden Database Systems Group

Example

- requirement: Main memory capacity = 3 tuples
- hashing of R_1 with $h(x) = x \bmod 3$



Partitioned Hash-Join

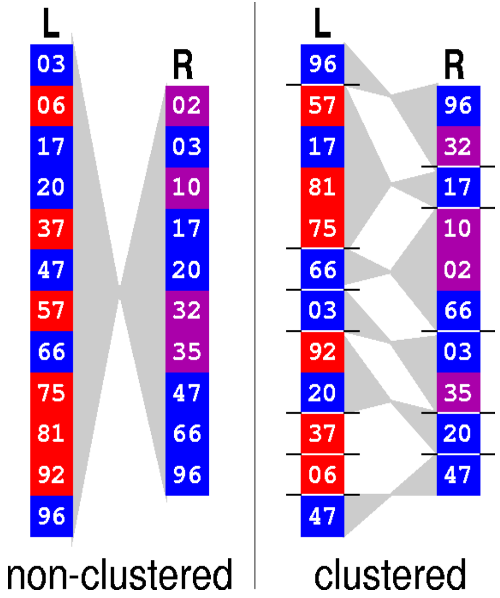
Dresden Database Systems Group

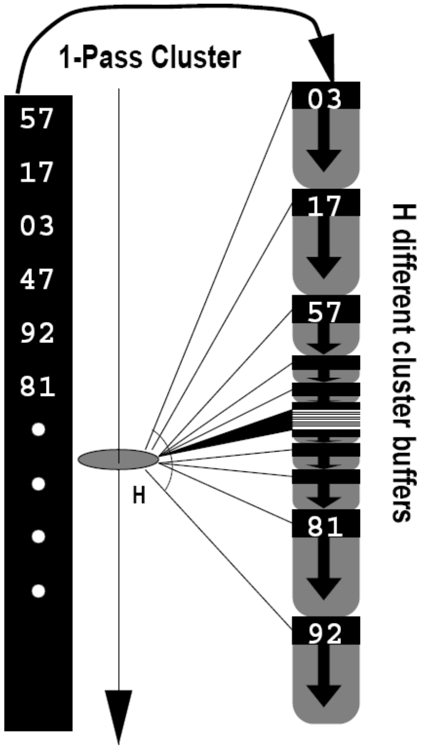
Traditional Hash-Join

- uses random access pattern
- if randomly accessed data is too large for the CPU cache, each tuple will cause cache misses and performance degrades

Partitioned Hash-Join

- both relations are first partitioned on hash-number in H separate clusters
- each cluster fits into L2 memory cache
- clustering operation can become a cache problem
  - random access pattern that writes in H separate locations
  - if H exceeds number of available cache lines, cache trashing occurs





Hash Join– Simple Hashing (2)

STEP 1

- run scan on smaller relation R
- check P_R, and apply the hash function $h_P()$ to each qualified tuple r
- if $h_P(r.JA)$ falls into the selected range, enter it in H_i
- otherwise, store r in a temporary intermediate file for "passed" r tuples

STEP 2

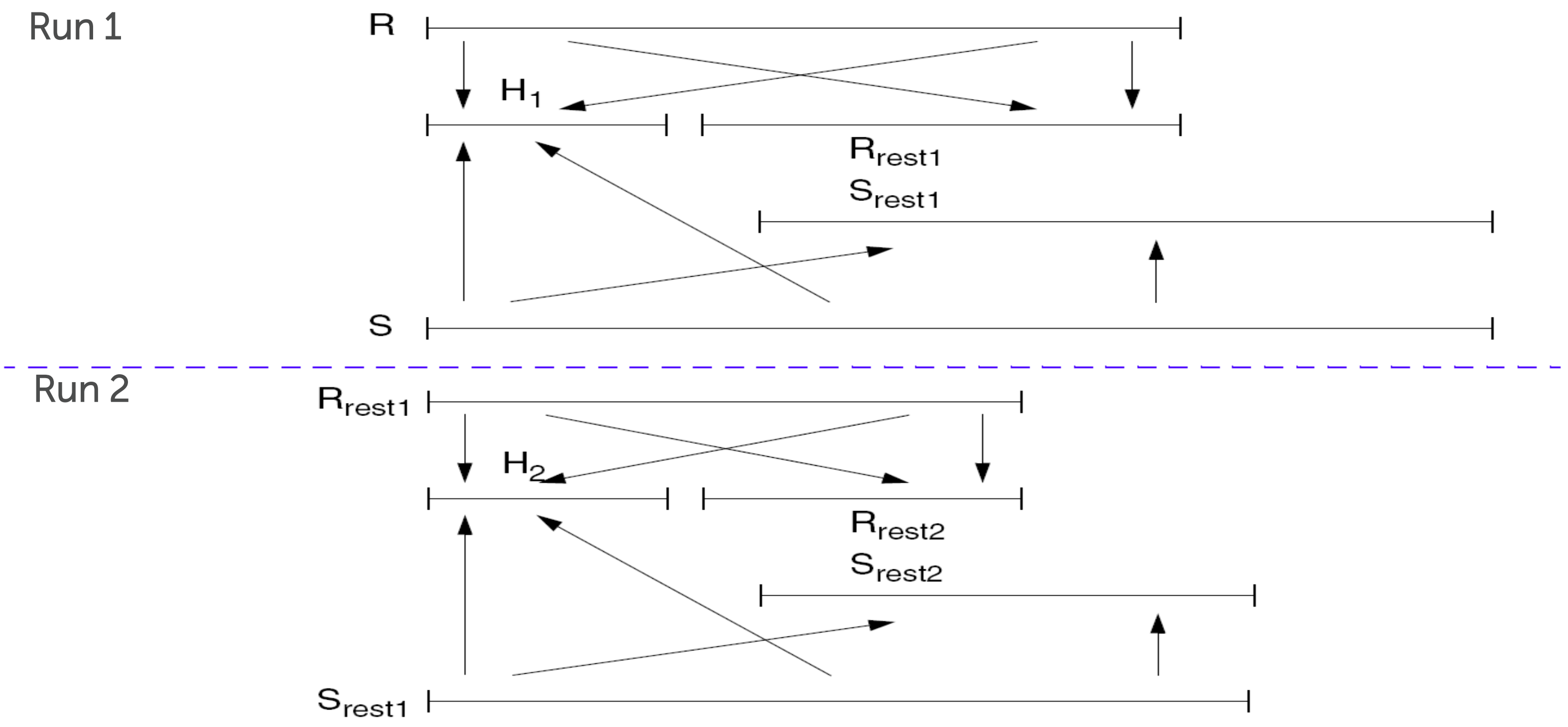
- run scan to S
- check P_S and apply the hash function $h_P()$ to each qualified tuple s
- if $h_P(s.JA)$ falls into the selected range, search a join partner in H_i (probing)
- if successful, form a join tuple and assign it to the result
- otherwise, save it to a temporary intermediate file for "passed" s tuples

STEP 3

- repeat steps 1 and 2 with the previously passed tuples until R is exhausted
  - the check of P_R and P_S is no longer required

Hash Join– Simple Hashing (3)

Illustration

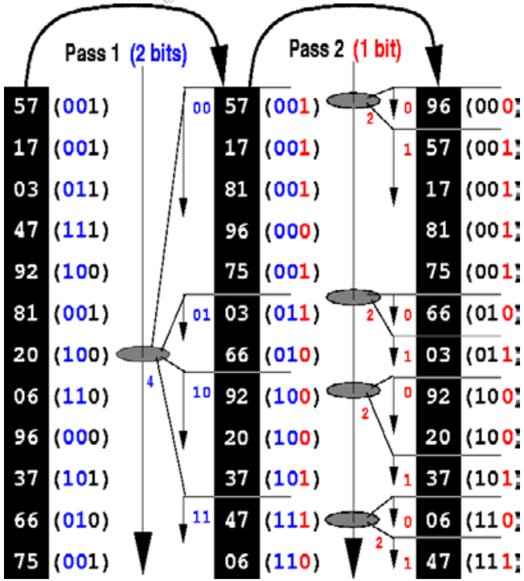


Radix Join

Dresden Database Systems Group

Radix Cluster

- splits a relation into H clusters using p sequential passes
- clustering is done on the lower B bits of a integer hash-value of a column
- each pass clusters tuples on B_p bits, starting with the leftmost bits
- each pass subdivides each cluster into $H_p = 2^{Bp}$ new clusters
- number of clusters $H = \prod H_p$

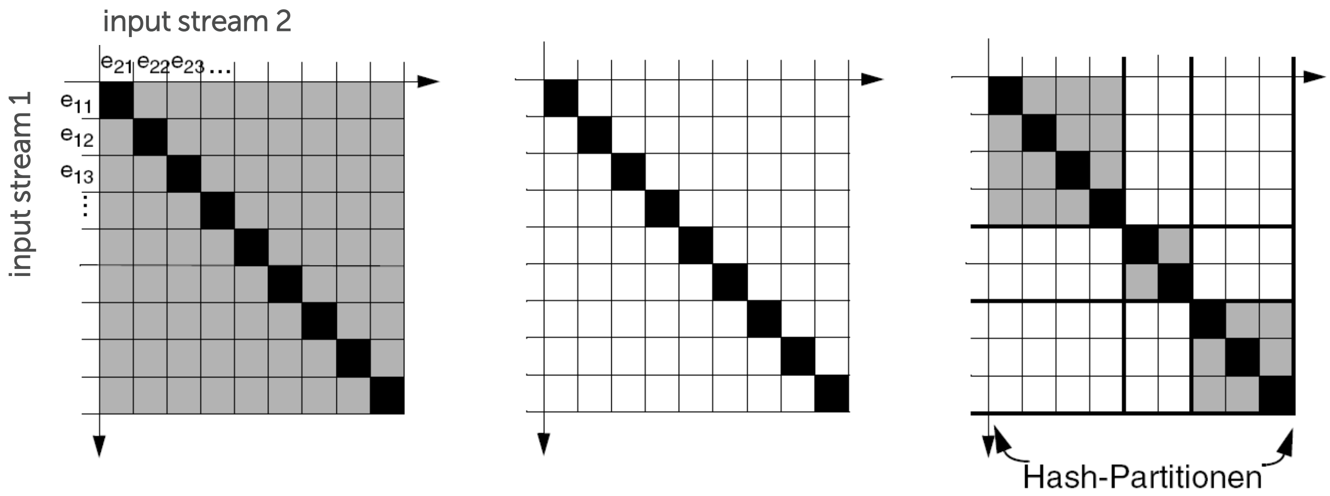


Advantages

- number of randomly accessed regions H_x can be kept low; while still a high overall number of H clusters can be achieved using multiple passes
- avoids cache trashing if H_x is kept smaller than the number of cache lines
- not necessary to store cluster boundaries in additional data structures
- a radix-clustered relation is ordered by radix-bits; for the join a merge step is performed to get pairs of clusters

Comparison of Join Algorithms

Dresden Database Systems Group



Nested-Loop Join

Sort-Merge Join

Hash-Join

element comparison

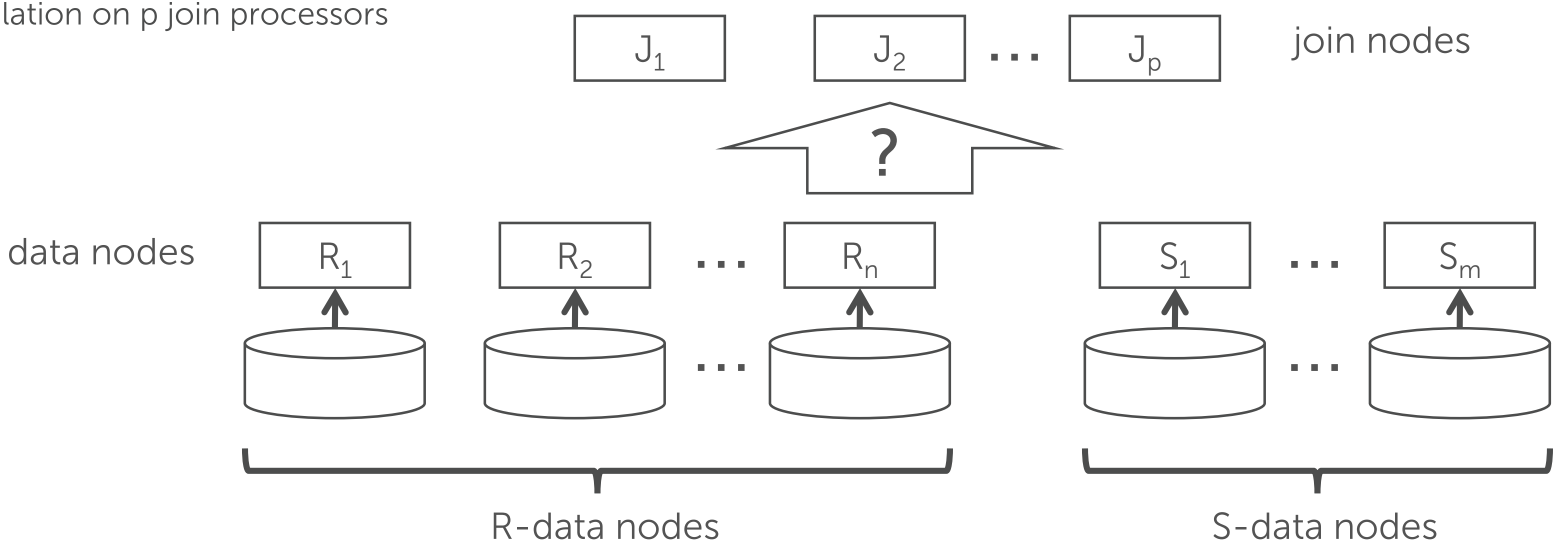
successful element comparison

… more complex join operators

Parallel Join

SCENARIO

- equality join between R and S
  - $R = \cup (R_1, R_2, \ldots, R_n)$
  - $S = \cup (S_1, S_2, \ldots, S_m)$
- S is less than R
- join calculation on p join processors



Parallel Join(2)

Principle of the parallel hash join

- redistribution of the smaller relation S over hash function h () to join attribute
- in join processors, incoming tuples get in main store hash table
- redistribution of the second relation R to the join processors using the hash function
- probing: determine join partners in hash table for incoming tuples

Characteristics

- sequencing of the scan phases
- advantage: Reduction of the redistribution effort for R by using bit-vector filtering possible
- pipeline parallelism in building and probing phase possible
- overflow treatment required, if S-partitions are not fully accommodated in main memory (=> three-level partitioning)

Configurations of parallel joins

Replicated join "broadcast join"

- partitioning with small relations is not worth it. ...
- assign a copy of the smaller join partner to the partitions of the larger join partner
  - advantage: no relation must be partitioned after the join attribute

One-way redistribution join "directed join"

- one of the two join partners is partitioned after the join attribute
- partitions of the other join partner are partitioned newly at runtime after the join attribute
- example
  - order relation is partitioned according to the customer key
  - repartitioning by the attribute O_ORDERKEY

Configurations of parallel joins

Completely redistributive join „repartitioned join“

- both join partners are repartitioned after the join attribute
- high communication costs -> Avoid!

Partition local join (›co-located joins‹)

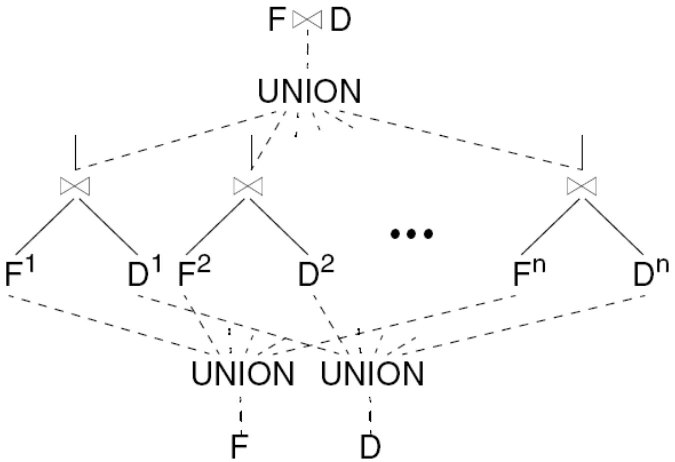
- at the same time, join attribute is a composite attribute for both partners
- maximum parallelism with minimal communication effort between the parallel-running join operators
- Eexample
  - fact table and order relation (ORDERS) partitioned according to L_ORDERKEY or O_ORDERKEY
- partitions local join in the following query

```
SELECT O_ORDERPRIORITY, SUM(L_QUANTITY) AS SUM_QUAN
  FROM TPCD.LINEITEM, TPCD.ORDERS
 WHERE L_ORDERKEY = P_ORDERKEY
 GROUP BY O_ORDERPRIORITY;
```

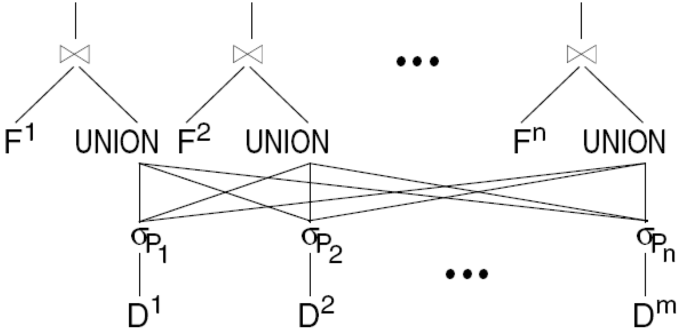
Example: parallel join

Example

- F Fact table
- F^k Partitions of the fact table ($1 \le k \le n$)
- P_k Partitioning predicates of the fact table ($1 \le k \le n$)
- D Dimension table
- D^k Partitions of the dimension table ($1 \le k \le n/m$)



partition local join



one-sided directed join

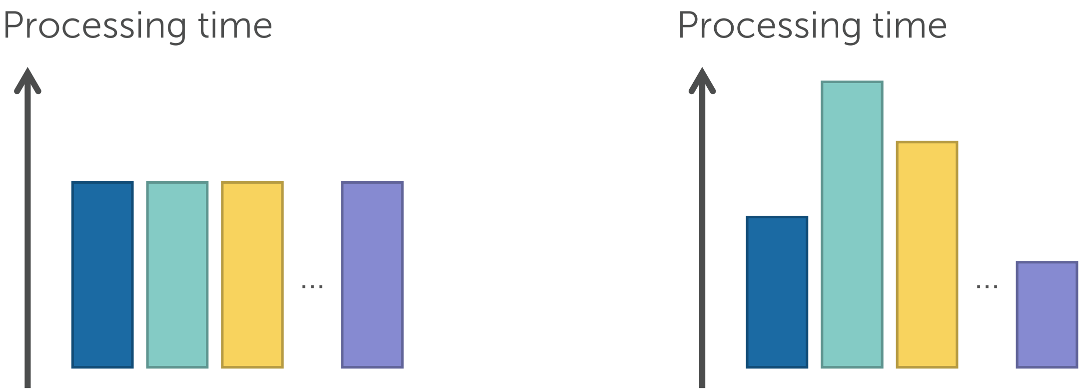
Data Skew

Observation

- unequal processing time of partial operations (execution skew) impairs parallelization

Cause

- execution skew often goes back to data skew:
  differently sized data sets per partial operation due to non-uniform distribution of attribute values und tuples



No skew (optimal parallelization)

Execution skew

Data Skew (2)

Dresden Database Systems Group

Data distribution skew (tuple placement skew)

- different partition sizes
- uneven duration of scanning operations
- treatment:
  best knowledge of the distribution of values for distribution attributes
  - histograms
  - sampling
  - determination during sorting on sort merge joins

Redistribution skew (redistribution skew)

- distribution function leads to different fragment sizes
- treatment:
  such as data distribution skew

Data Skew (3)

Dresden Database Systems Group

SELECTIVITY SKEW

- different hit rates per computer
- (e.g. Area queries regarding distribution attribute for area partitioning)
- treatment:
  hardly treatable, as determined by request and data transfer

JOIN PRODUCT SKEW

- different join selectivity per node
- treatment:
  - estimation of the total size of the join result as well as the resulting value distribution for the join attribute
  - determine area partitioning, which provides a roughly equal partial result for each of the p join processors

Join Operations for Star-Queries

Prerequisites for a Star Join

- fact table is always part of a star query
- join with dimension table reflects an indirect selection via restrictions on the dimension relation ("extension of the fact table by dimensional attributes on the fly …")

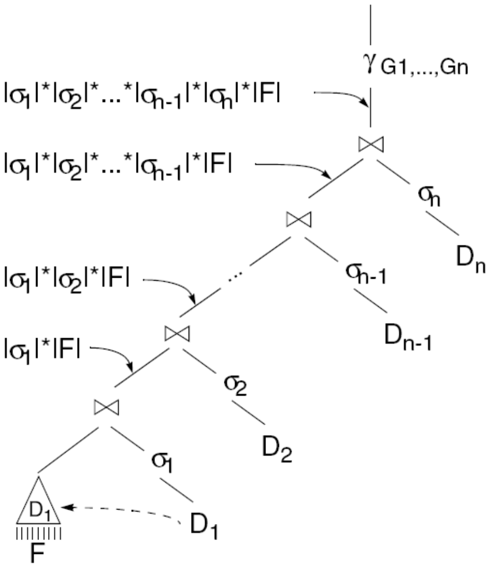
Problems with a Classic Join Operation

- fact table is already included in the first join operation
- dimension tables are successively linked
- size of the data stream starts with |F| and only gradually decreases
- only a single one-dimensional index can only be used in the first join operation

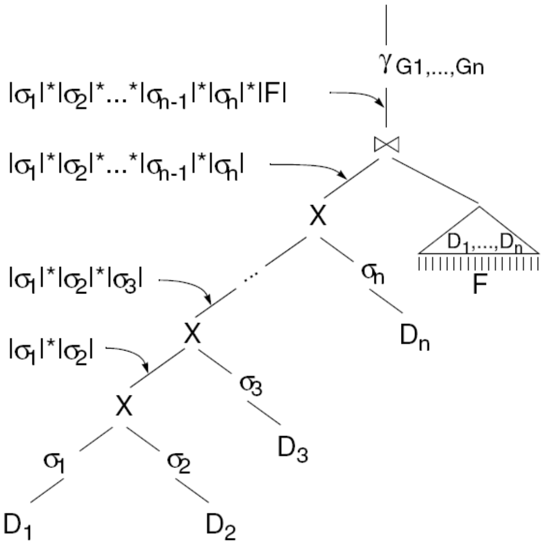
Star Join with Restructuring

STAR-JOIN WITH RESTRUCTURING

- formation of the Cartesian product of the dimension tables
- access to fact table with n-fold index
- disadvantage: Cartesian product can become very large!



a) Operatorengraph einer Star-Query



b) Operatorengraph einer Star-Query nach Bildung des kartesischen Produktes

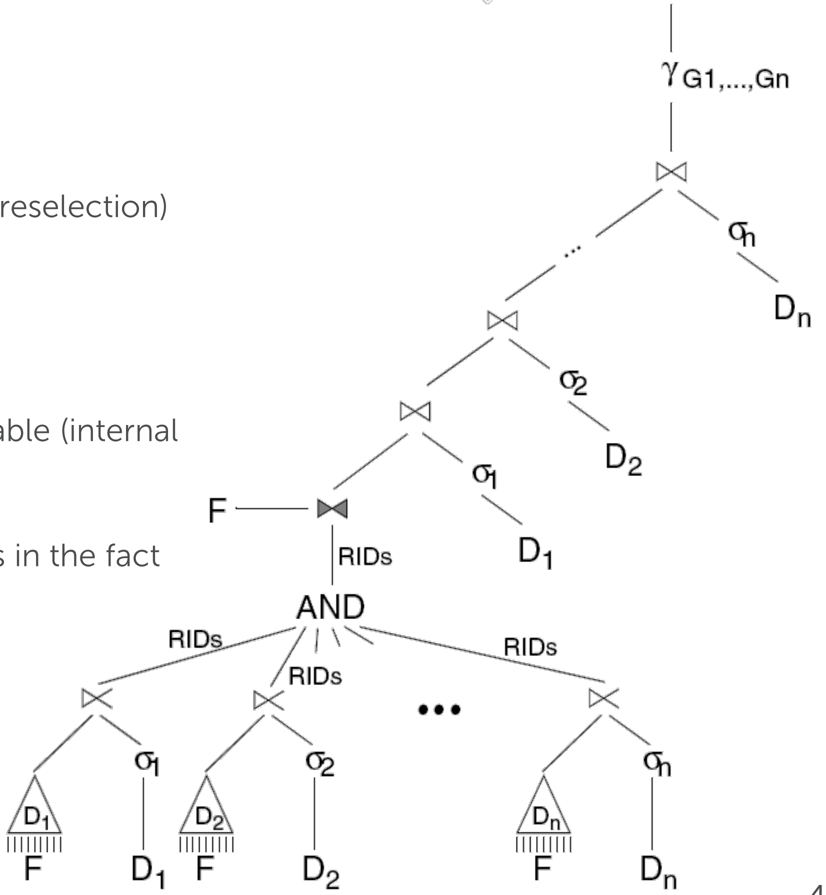
Star Join with Preselection

IDEA

- maintain the regular join strategy
- reduction of the size of the fact table in a preliminary stage (preselection)
- requires: An index on the foreign key attributes of F

REALIZATION

- local selection on the dimension tables
- semi join with the corresponding index structure of the fact table (internal record addresses are retained)
- intersection of all partial results
- resulting record addresses exactly identify the required entries in the fact table



$\bowtie$ wertebasierter Gleichheitsverbund

$\ltimes$ Semi-Verbund

$\bowtie$ RID-basierter ›Verbund‹ (FETCH-Operator)

Star Join with Fuzzy Preselection

Dresden Database Systems Group

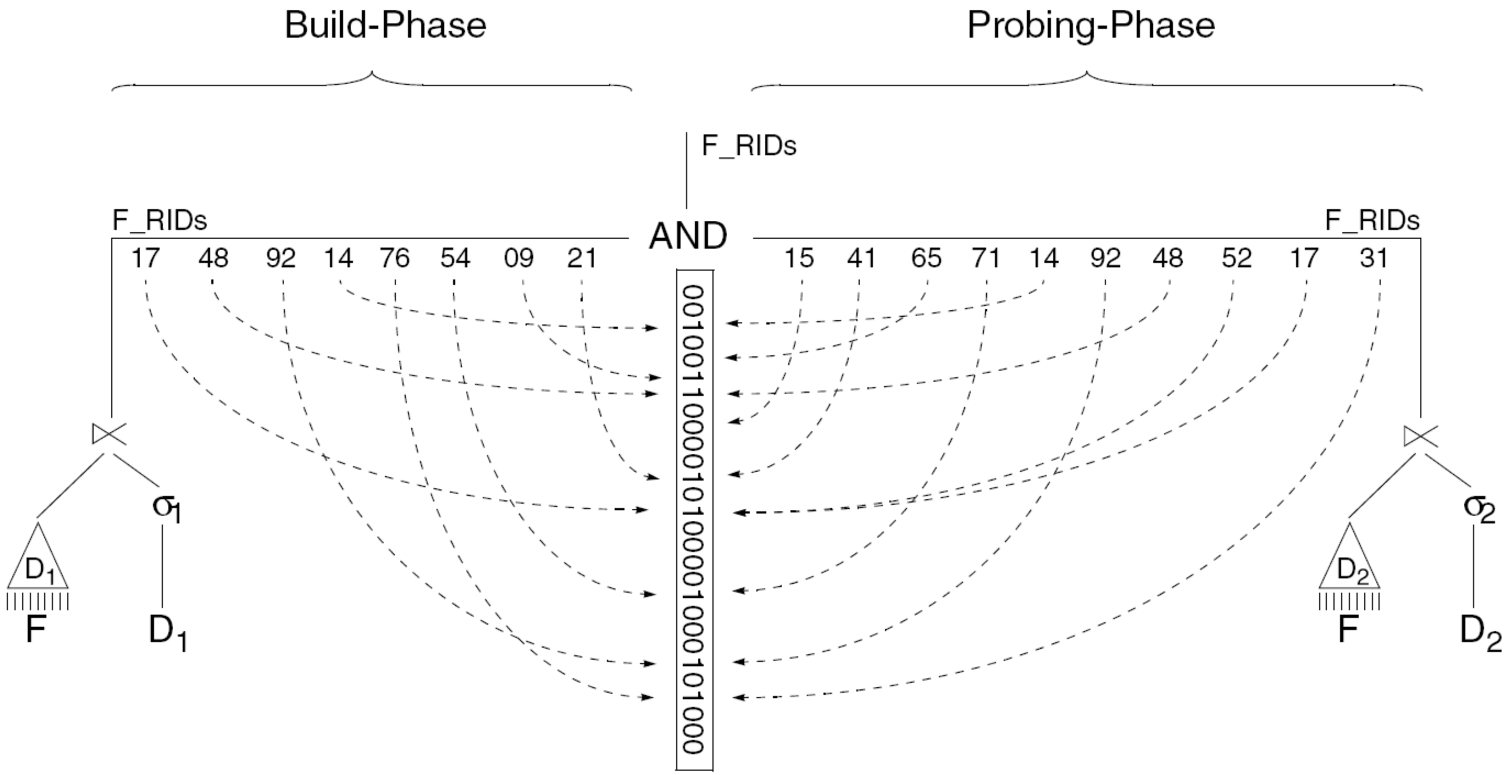
Evaluation of Preselection

- sdvantage: The fact table has a final size before the join operation
- disadvantage: extremely complex intersecting sets of the RID lists

Introduction of a Fuzziness: Bloom Filter Technology

- generation of a bit list initialized with 0 in main memory
- build phase
  - hash function determines the position of the bit list for each RID entry: set to 1
- probing phase
  - dumping of RID, if corresponding bit list entry is 0
  - otherwise: inclusion in the result (fuzzy!)
- example
  - build Phase:
    h(RID 14)=3, h(RID 21)=12
  - probing Phase:
    h(RID 14)=3 --> correct result, h(RID 65)=4 --> correct dumping,
    h(RID 31)=25 --> Erroneously taking over in result
  - quality depending on the size of the bit list; Correction by real join operations

Star Join with Fuzzy Preselection



Join in Distributed Systems

PROBLEM

- request in node K, which requires a join between (partial) relations R at node KR and (partial) relation S at node KS
- definition of the execution node: K, KR or KS

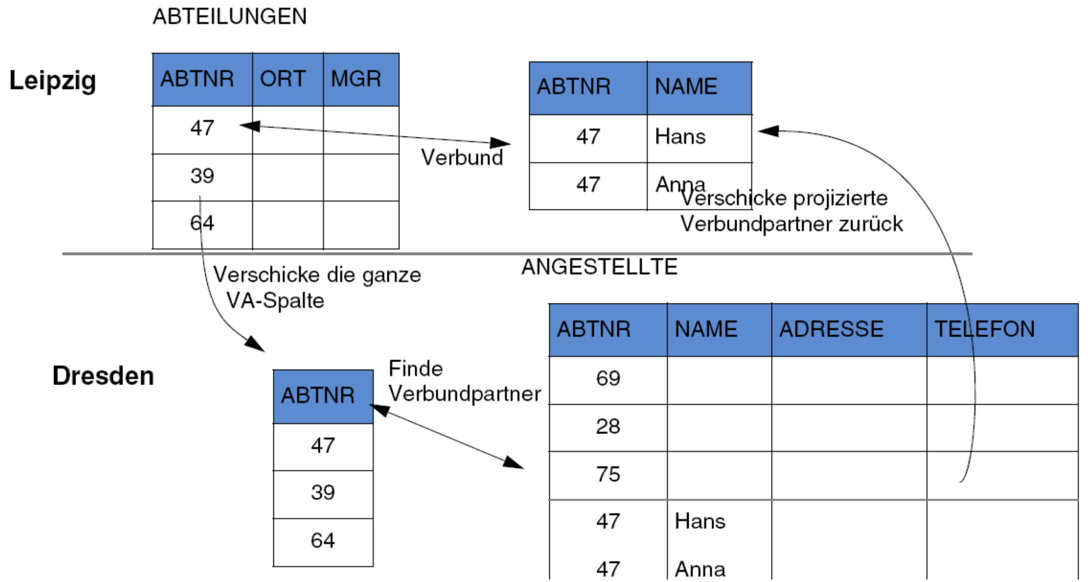
DETERMINATION OF THE EVALUATION STRATEGY

- „Ship Whole“:
  transmit participating relations completely to a node and perform local join computation
  - minimum number of messages
  - very high transfer volume
- „Fetch as Needed“:
  request associated tuples of the second relation for every join value of the first relation
  - high number of messages
  - only relevant tuples are considered
- compromise solution:
  semi join or extensions such as bit-vector-grouping (hash-filter-join)

Join in Distributed Systems (2)

Semi Join

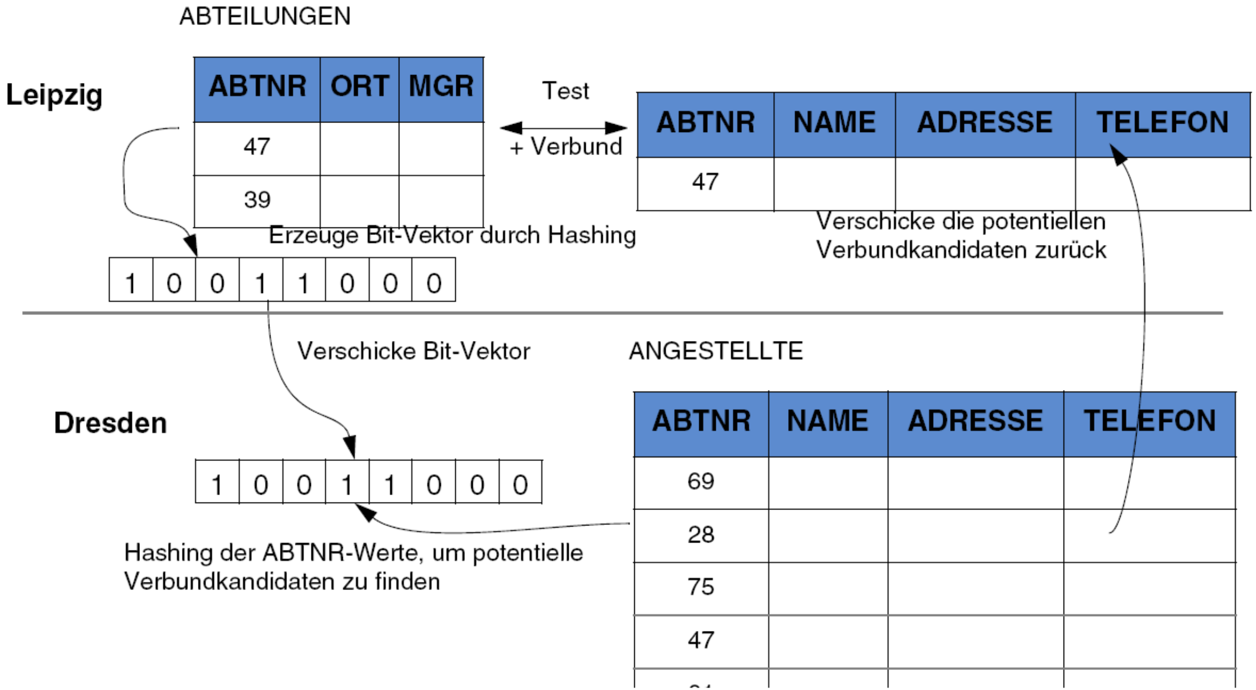
- send a list of VA from R to node S
- determine the compound partners in S and return to the node of R
- implementation of the join



Join in Distributed Systems (3)

Bit Vector Join

- similar to semi join, only sending a hash function created bit vector (bloom filter)
- returning a superset of the join partners in S



Invisible Joins

Dresden Database Systems Group

Observation

- many data warehouses model data using star/snowflake schemes
- joins of one (fact) table with many dimension tables is common
- with late materialization values from dimension table group-by columns need to be extracted in out-of-position order

Invisible Join

- late materialized join, minimizes the values that need to be extracted out-of-order
- makes sure that the table that can be accessed in position order is the fact table for each join
- rewrites joins into predicates on the foreign key columns in the fact table
- position lists from the fact table are then intersected (in-position order)
- reduces the amount of data that must be accessed out of order from the dimension tables

Invisible Join

Dresden Database Systems Group

Phase 1

- for each predicate dimension table keys are extracted which satisfy the predicate
- keys are used to build a hash table

Apply "region = 'Asia'" On Customer Table

custkey	region	nation	...
1	ASIA	CHINA	...
2	EUROPE	FRANCE	...
3	ASIA	INDIA	...

→ **Hash Table Containing Keys 1 and 3**

Apply "region = 'Asia'" On Supplier Table

suppkey	region	nation	...
1	ASIA	RUSSIA	...
2	EUROPE	SPAIN	...

→ **Hash Table Containing Key 1**

Apply "year in [1992,1997]" On Date Table

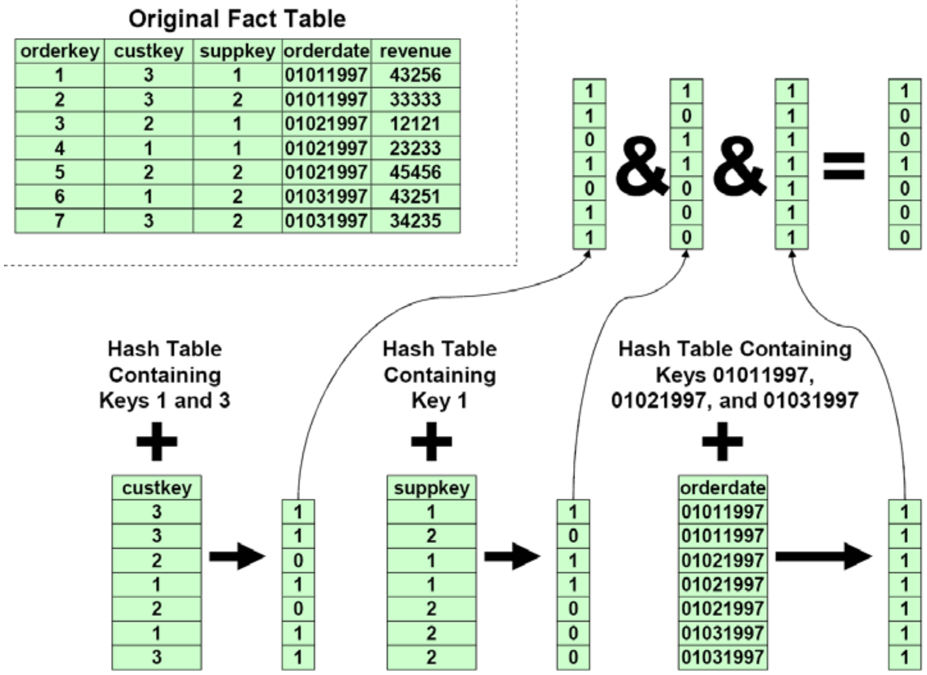
dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

→ **Hash Table Containing Keys 01011997, 01021997, and 01031997**

Invisible Join

Phase 2

- hash table is used to extract the positions of records in the fact table that satisfy the corresponding predicate
- each value in the foreign key column of the fact table is probed into the hash table
- results in a list of all positions in the foreign key column that satisfy the predicate
- lists from all of the predicates are intersected to generate a list of satisfying positions P in the fact table



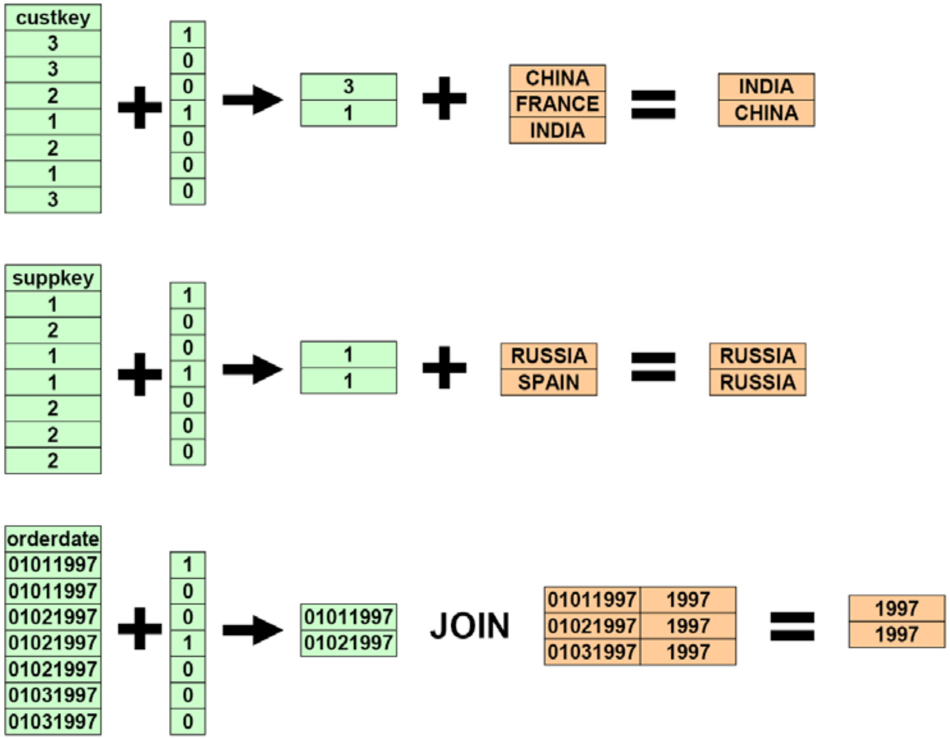
orderkey	custkey	suppkey	orderdate	revenue
1	3	1	01011997	43256
2	3	2	01011997	33333
3	2	1	01021997	12121
4	1	1	01021997	23233
5	2	2	01021997	45456
6	1	2	01031997	43251
7	3	2	01031997	34235

Invisible Join

Dresden Database Systems Group

PHASE 3

- for each referenced dimension column in the query, corresponding foreign key values from the fact table are extracted and looked up in the dimension table
- if dimension table key is a sorted list of identifiers starting from 1, then foreign key represents the position of the tuple in the dimension table



Jive/Flash-Join

Invisible Join

- still accessing table out of order

Jive/Flash-Join

- instead of probing projected columns from inner table out of order:
  - sort join index
  - probe projected columns in order
  - sort result using an added column

LM vs EM tradeoffs

- LM has the extra sorts (EM accesses all columns in order)
  - (Radix Sort can be used)
- LM only has to fit join columns into memory (EM needs join columns and all projected columns)
- LM only has to materialize relevant columns

Database Cracking

Database Cracking

Dresden Database Systems Group

Problem

- non-discriminative index structures (assume uniform distribution)
- high maintenance overhead during updates
- index selection is a weak compromise amongst many plausible plans

➔ Offline index selection fails due to dynamic workload changes

➔ Online index selection (on logical level) fails due to compromise between queries, too

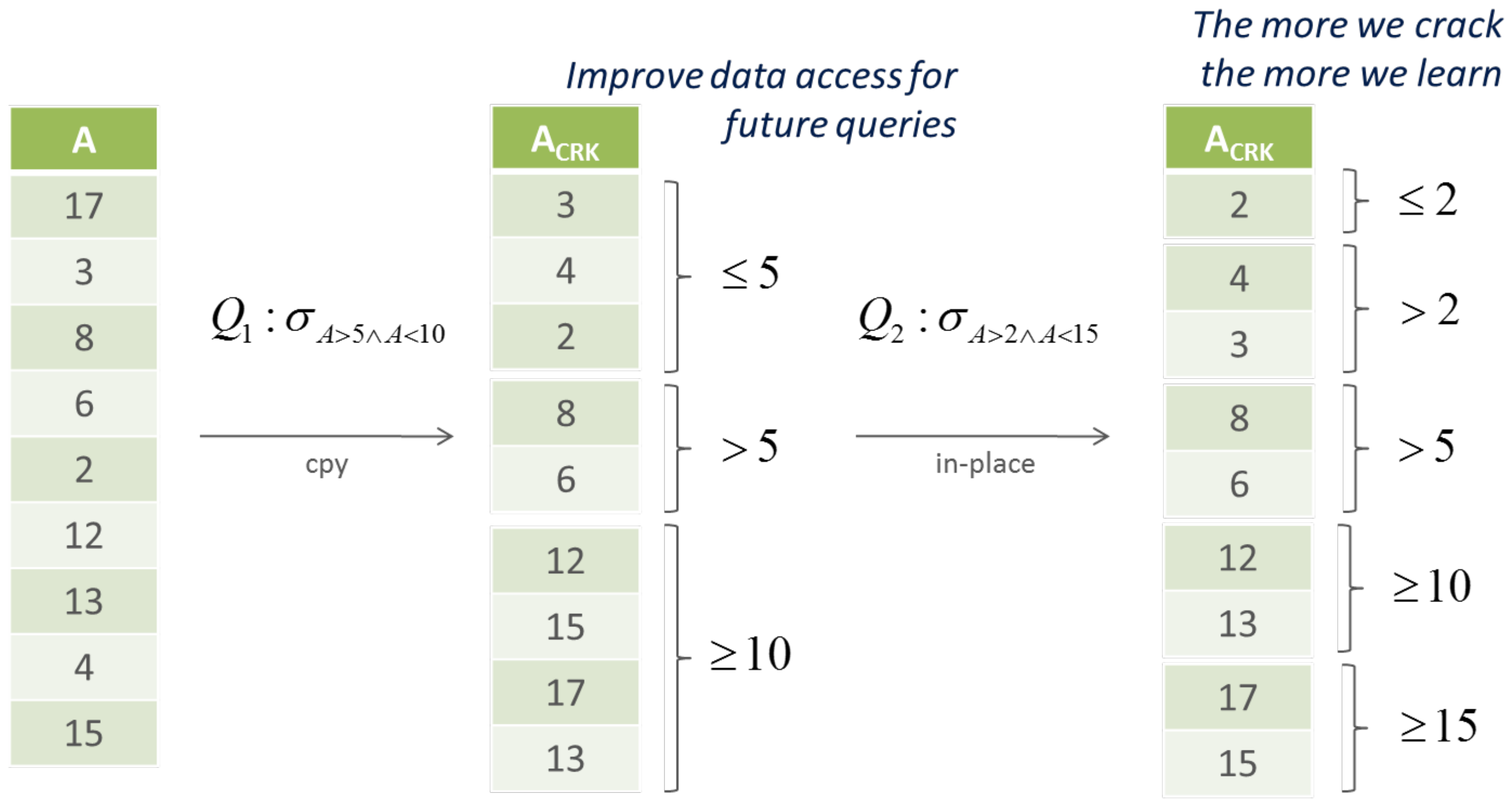
Idea

- continuously adapt the database organization
- each query triggers physical re-organization of the db

Cracking

- cracking … Split a column into smaller pieces
- index maintenance as part of query processing
- physical reorganization for self-organized behavior

Example of Database Cracking



Database Cracking

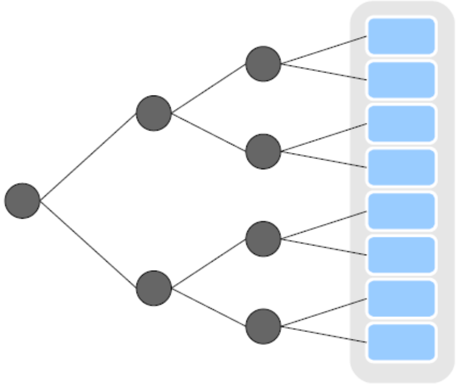
CRACKING ALGORITHMS

- algorithm **`CrackInTwo(c,posL,posH,med,inc)`**
  - split a slice into two pieces using single-sides predicates ($A\ \theta\ med$)
- algorithm **`CrackInThree(c,posL,posH,low,high,incL,incH)`**
  - split a slice into three pieces using double-sided predicated (low $\theta_1\ A\ \theta_2\ high$)
- three-piece cracking is semantically equivalent to two subsequent two-piece crackings
  (low $\theta_1\ A\ \theta_2\ high$) = (low $\theta_1\ A$) & ($A\ \theta_2\ high$)
- both algorithms are single-pass algorithms

Database Cracking

Ordered Pieces of Cracker Column

- the first time a range query is posed on an attribute A, a cracking DBMS makes a copy of column A, called the cracker column of A
- a cracker column is continuously physically re-organized based on queries that need to touch attribute such as the result is in a contiguous space
- for each cracker column, there is a cracker index
  - AVL-tree
  - B-tree



→ How does cracking compare to a sort-based strategy?

→ How does cracking compare to traditional indices?

Cracking vs. Indices and Sorting

SORTING

- sort the data upfront and then perform fast binary search operations
- preconditions
  - which data is interesting for the user/queries ?
  - which single (combination of) attributes is primarily requested ?
  - time and resources to create the physical order before any query arrives
  - no updates or long sufficient time between updates and queries

  ➔ If all those information and resources exist sorting is superior

INDICES

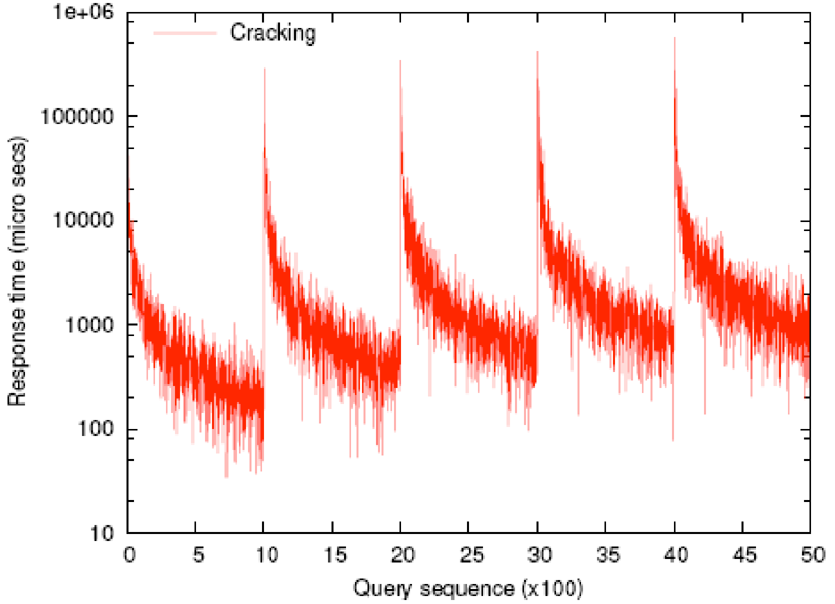
- similar arguments as for sorting
- initial costs for creation and maintenance costs for index update

	Index	Cracking
Creation	Initial effort	Pay-as-you-go
Maintenance	updates	queries

Naive Insertion Approach

Forget Algorithm (FO)

- when a query request a value range such that one or more tuples are contained in the pending insert column:
  1) Delete (forget) the cracker index
  2) Simply append all pending insertions
  3) Cracker index rebuilt from scratch



Problems

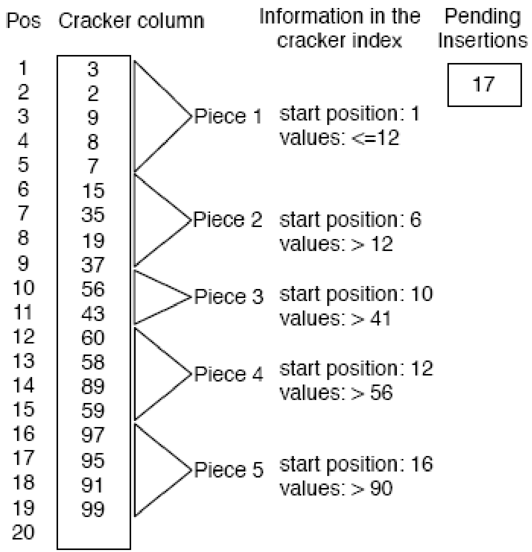
- a number of queries (after FO) suffer a high cost
- large parts of the cracker column is built periodically
- no predictability in terms of response time

In-Place Insertion

Dresden Database Systems Group

Ideal cracker index maintenance

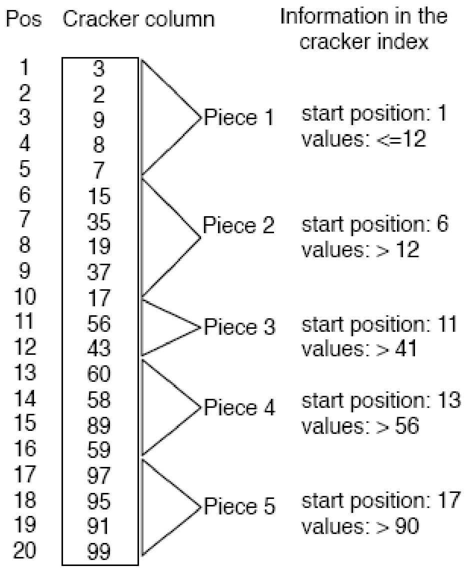
- inserts without loosing any information from the cracker index
- merge-like maintenance algorithms (in-place)



(a) Before the insertion

$$Q_1 : \sigma_{A>5 \wedge A<50}$$

Problem:
moving tuples in different positions of the cracker column
➔ High costs (10 tuples)



(b) After inserting value 17.

TECHNISCHE UNIVERSITÄT DRESDEN

Shuffling Technique

Core Idea

- inside each piece tuples have no specific order
- exchange as less tuples as possible

Details

- piece *p* holds k tuples
- pending inserts *z*
- if *k<= z*
  - moving p completely with all k tuples
- if k>z
  - take z tuples from the beginning of p and move it to the end of p

Finally, we moved only 3 tuples

...	...
9	37
10	17
11	43
12	56
13	58
14	89
15	59
16	60
17	95
18	91
19	99
20	97

17

$pos:10$

>41

$pos:11$

>41

$pos:12$

>56

$pos:13$

>56

$pos:16$

>90

$pos:17$

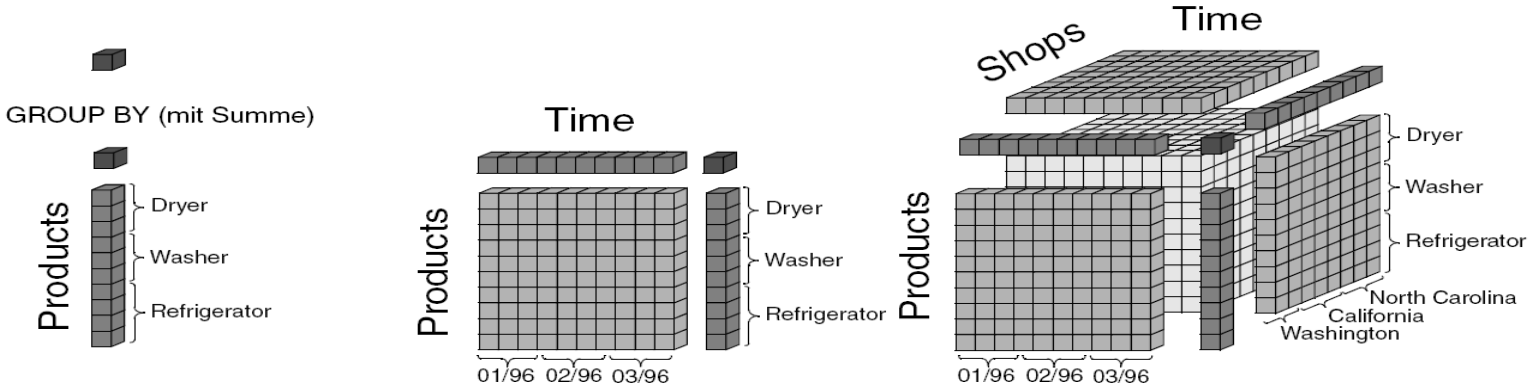
>90

Multidimensional Grouping

CUBE

CUBE-OPERATOR: EXTENSION FOR THE GROUP BY CLAUSE:

- abbreviation for the enumeration of all 2^n possible grouping combinations, i.e.
  `GROUP BY CUBE(A,B) is equivalent to GROUP BY GROUPINGS SETS ((A,B),(A),(B),())`
- nulldimensional data cubes: CUBE()
  a single aggregation value / cell (non-existent group-by clause)
- one-dimensional data cubes: CUBE(Article)
  a value line and a cell
- two-dimensional data cubes: CUBE(Article, Day)
  an area, two rows of values, a cell (cross table with "totals")



CUBE (2)

Dresden Database Systems Group

Example

```
SELECT L_SHIPMODE, L_SHIPINSTRUCT,
       SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
       GROUPING(L_SHIPMODE) AS GRP_SHIPMODE,
       GROUPING(L_SHIPINSTRUCT) AS GRP_SHIPINSTRUCT
  FROM TPCD.LINEITEM  GROUP BY CUBE(L_SHIPMODE, L_SHIPINSTRUCT);
```

L_SHIPMODE	L_SHIPINSTRUCT	SUM_CHARGE	GRP_SHIPMODE	GRP_SHIPINSTRUCT
AIR	COLLECT COD	761624782,329	0	0
AIR	DELIVER IN PERSON	754419485,270	0	0
AIR	NONE	763523815,451	0	0
AIR	TAKE BACK RETURN	769723164,447	0	0
AIR	-	3049291247,498	0	1
MAIL	COLLECT COD	760829060,943	0	0
MAIL	DELIVER IN PERSON	765447918,010	0	0
MAIL	NONE	756568001,823	0	0
MAIL	TAKE BACK RETURN	767495428,550	0	0
MAIL	-	3050340409,327	0	1
TRUCK	COLLECT COD	760705270,142	0	0
...				
-	COLLECT COD	5334791395,658	1	0
-	NONE	5340588680,414	1	0
-	TAKE BACK RETURN	5364491060,031	1	0
-	-	21356601173,078	1	1

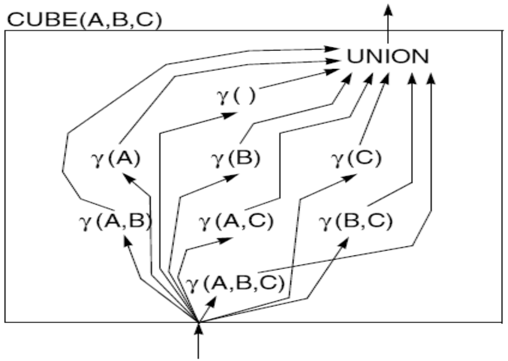
CUBE – Implementation

Problem

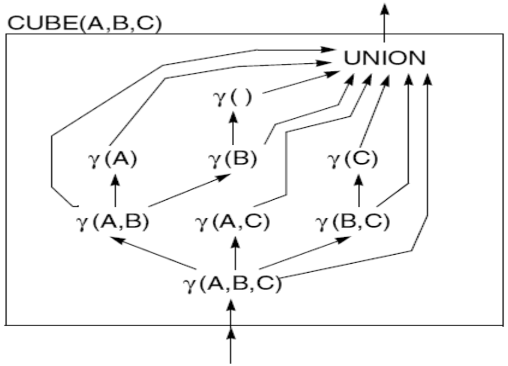
- CUBE() results in 2^n grouping combinations

Naive variation

- 2^n-fold execution of the core query and computation of the union
- improvement by exploiting direct derivability



a) Naiver Ansatz



b) Ausnützen der direkten Ableitbarkeit

CUBE – Implementation (2)

Dresden Database Systems Group

Problem

- data must be reordered to calculate different grouping combinations
- example
  - data is sorted by ABC
  - γ(AB) or γ(AC) can be calculated without sorting
  - γ(BC) requires sorting according to BCA

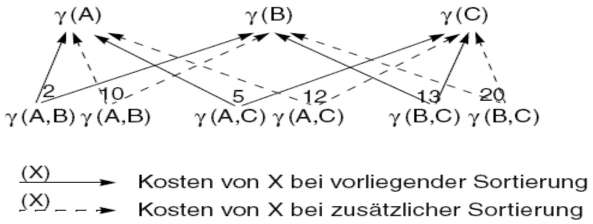
Selection of direct predecessors nodes with sorting order

- sorting the data with minimal costs → conflict:
  - smallest predecessor („smallest parent“)
  - favorable sorting order („share sort“)
- per level within the aggregation grid with n attributes
  - an operator instance with costs without sorting
  - N-1 copies of the operator with cost of sorting
  - find the minimum cost matching on the bipartite graph

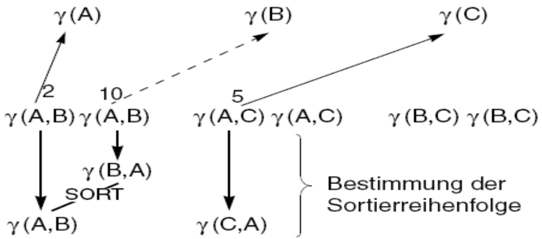
CUBE – Implementation (3)

Dresden Database Systems Group

CUBE implementation with pipelining



a) Erweitertes Aggregationsgitter



b) Ergebnis nach Paarbildung

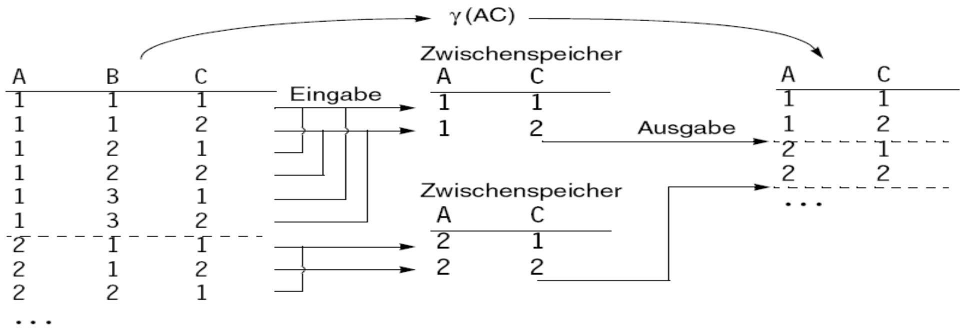
Extension

- relaxation of complete pipelining (introduce staging buffer)
- intermediate storage in compliance with partial sort order

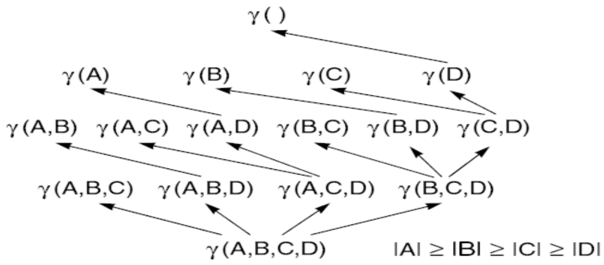
CUBE – Implementation (4)

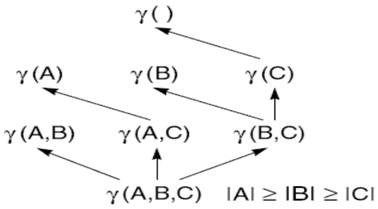
Calculation over partial sorting order

- selection of predecessors with partial sort order



- derivation tree while preserving the partial sorting order





CUBE – Implementation (5)

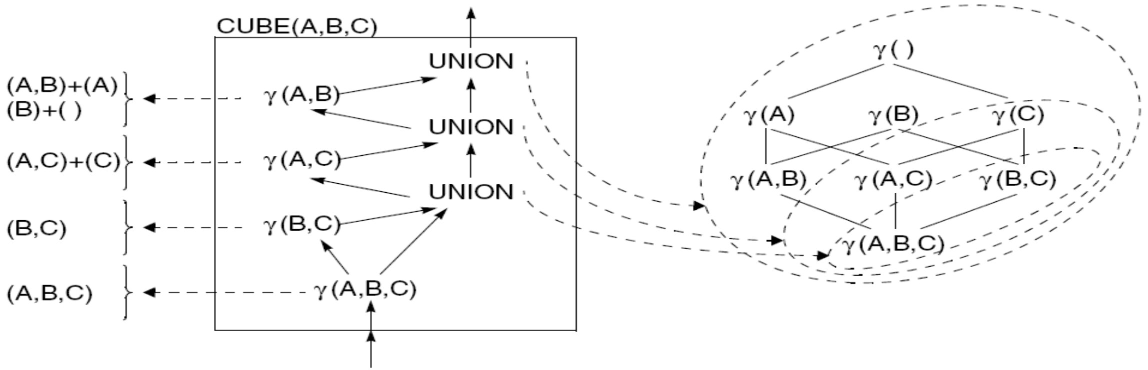
```
Algorithm: PartitionedCube
Input:  R          // Relation with tuples to be grouped
        G1,...,Gn  // Grouping attributes from the GROUP BY clause
        AGG(), A   // Aggregation function and attribute A to be aggregated
Begin
  If (Size(R) < MaxMemorySize)
    // If R fits in the main memory, a CUBE() is calculated based on the direct derivability
    C := ComputeMainMemoryCube(R, G1,...,Gn, A, AGG());
  Else
    // Selection of a grouping attribute according to which the data is partitioned horizontally into m fragments.
    k := PickSplitPosition(G1,...,Gn);
    (R1,...,Rm) := PartitionTableByAttr(R, k); // where: m ≤ |Gk|
    // Individual calculation of m partial cubes
    For i = 1 To m
      Ci := PartitionedCube(Ri, G1,...,Gn, AGG(), A);
      C:= C + Ci;
    End For
    // Calculate subtotal subtotals by reducing the set of grouping attributes by the partition attribute.
    C' := PartitionedCube(R, G1,..., Gk-1, Gk+1,...,Gn, AGG(), A);
    C := C + C';
  End If
  Return(C);
End
```

CUBE – Implementation (6)

INTERSECTION STACKING

- core difference to the partial quantity approach
  - direct derivability in the partial quantity approach („subset stacking“)
  - intersection stacking

- implementation of the CUBE() operator with intersection stacking



ROLLUP

ROLLUP-Operator: Combination of dimension hierarchies

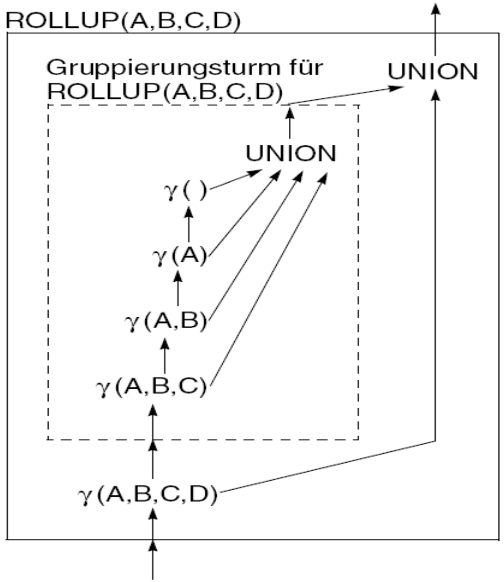
GROUP BY ROLLUP$(A_1,A_2,\ldots,A_n)$, **ROLLUP**$(B_1,B_2,\ldots,B_m)$

$$\left.\begin{array}{l} (\ ) \\ (A_1) \\ \ldots \\ (A_1,\ A_2,\ldots,A_{n-1}) \\ (A_1,\ A_2,\ldots,A_{n-1},A_n) \end{array}\right\} \times \left\{\begin{array}{l} (\ ) \\ (B_1) \\ \ldots \\ (B_1,\ B_2,\ldots,B_{m-1}) \\ (B_1,\ B_2,\ldots,B_{m-1},B_m) \end{array}\right.$$

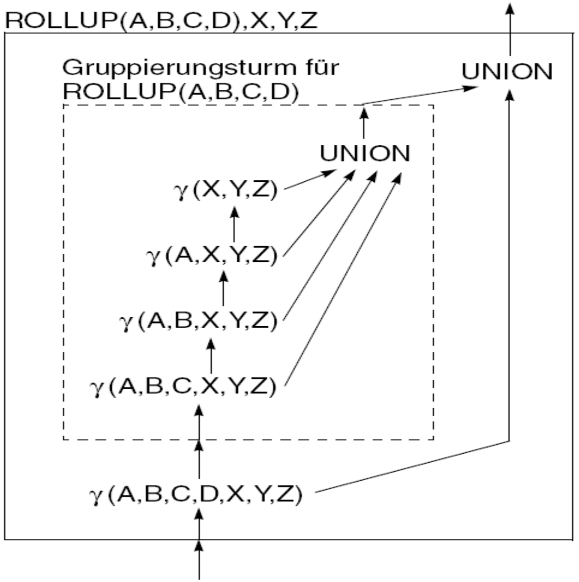
Example

```
SELECT ...
  FROM TPCD.LINEITEM,                                 -- Fact table
       TPCD.ORDERS, TPCD.CUSTOMER, TPCD.NATION N1    -- Order dimension
       TPCD.SUPPLIER, TPCD.NATION N2                  -- Delivery dimension
 WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY
   AND C_NATIONKEY = N1.N_NATIONKEY
   AND L_SUPPKEY = S_SUPPKEY AND S_NATIONKEY = N2.N_NATIONKEY
 GROUP BY ROLLUP(N1.N_REGIONKEY, N1.N_NATIONKEY, C_CUSTKEY, O_ORDERKEY),
          ROLLUP(N2.N_REGIONKEY, N2.N_NATIONKEY, S_SUPPKEY);
```

ROLLUP – Implementation



a) Einzelner Gruppierungsturm

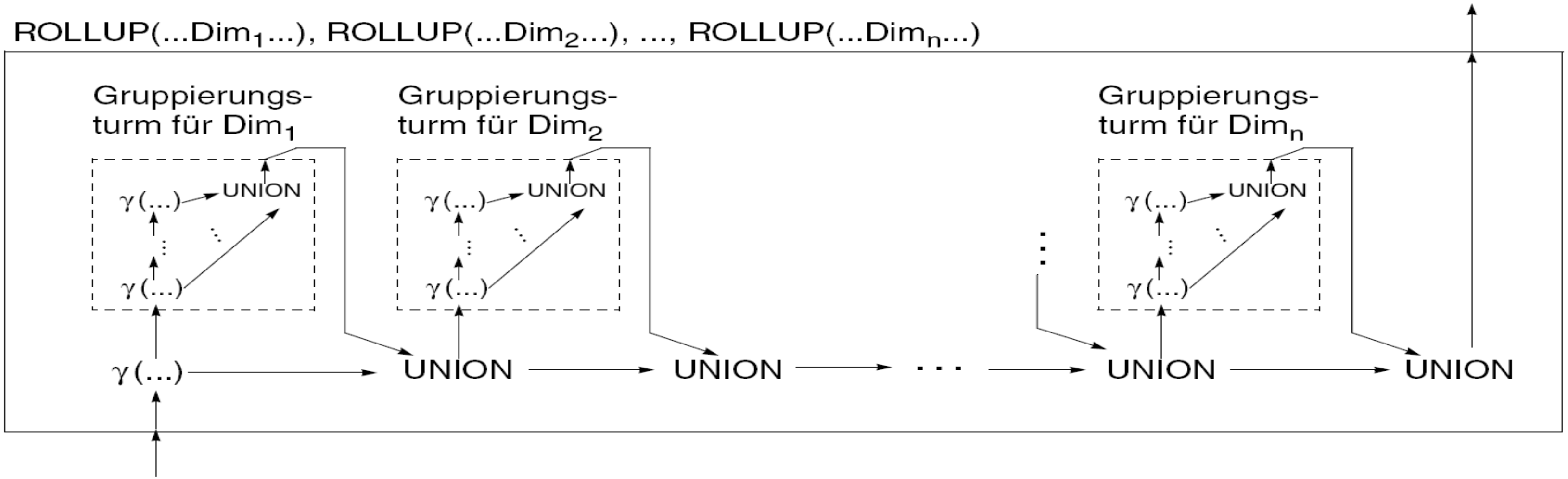


b) Gruppierungsturm mit weiteren Gruppierungsattributen

ROLLUP – Implementation (2)

CHAIN OF GROUPING TOWERS FOR HIERARCHICAL DATA CUBES

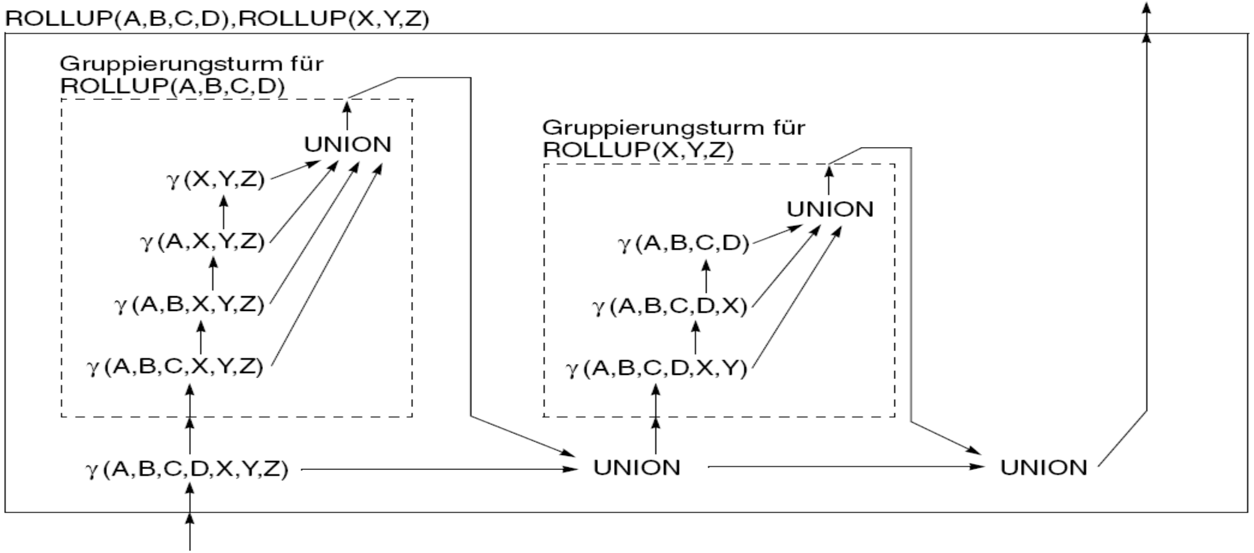
- successive evaluation of ROLLUP () constructs
- "pass" the partial results to the total score



ROLLUP – Implementation (3)

Dresden Database Systems Group

Example



Extension

- partitioning by a grouping attribute
- calculation of the partial cubes plus the groupings over the sums of the partial cubes

Summary

Unary operators

- selection and projection
- grouping
- aggregation
- sorting

Join Operators (Binary Operators)

- join operators
  - nested-Loop join
  - sort-Merge join
  - hash join
- parallelization of join operators
- handling of data skew

Multidimensional Grouping Operators

- CUBE()
- hierarchical data cubes with ROLLUP()

Literature

- Härder, T. & Rahm, E. Datenbanksysteme: Konzepte und Techniken der Implementierung. Springer-Verlag, 1999
- Saake, G.; Heuer, A. & Sattler, K.-U. Datenbanken: Implementierungstechniken. MITP-Verlag, 2005
- Hellerstein, J. M.; Stonebraker, M. & Hamilton, J. R. Architecture of a Database System. Foundations and Trends in Databases, 2007, 1, 141-259
- Graefe, G. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 1993, 25, 73-170