

Data Warehousing SoSe 2007

Dr. Jens-Peter Dittrich

jens.dittrich@inf

www.inf.ethz.ch/~jensdi

Institute of Information Systems



Query Processing and Indexing



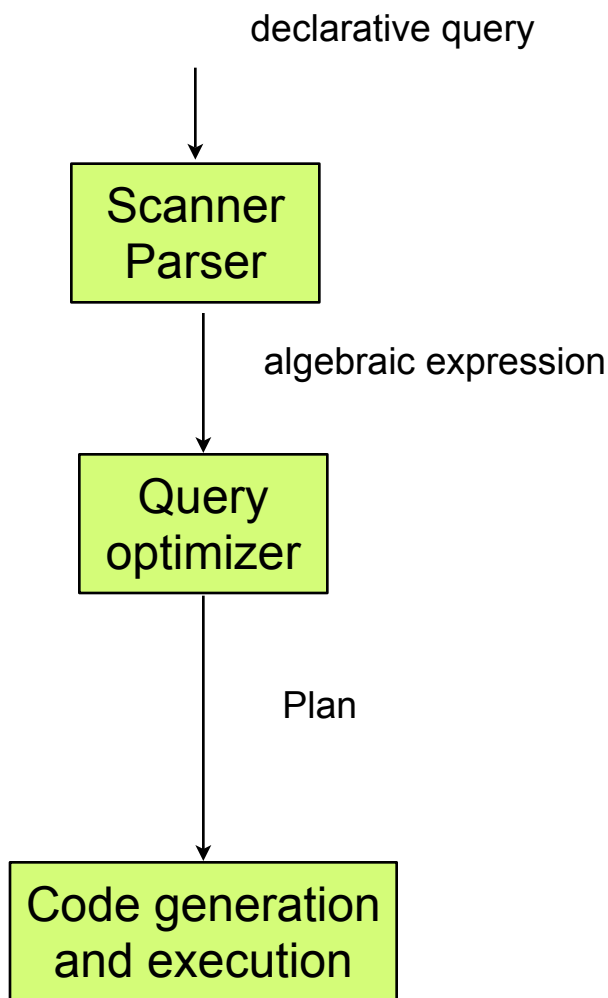
Agenda

- Basics
 - native plans
 - Star Joins
 - native
 - cross product
 - semi-joins
 - partitioning (range, hash)
 - mat views
- O'Neil article (How to index in order to speed-up star-join processing)
 - bitmaps
 - bitmap-operations
 - bloom-filters
 - join indices
 - bitmapped join indices
 - star joins using indices

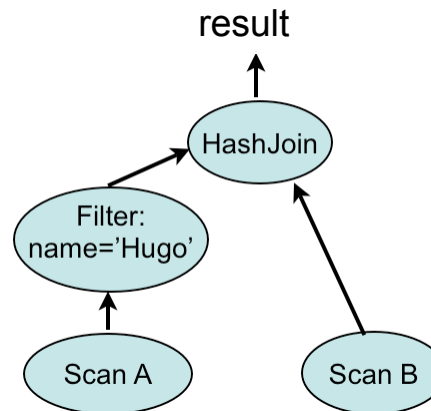
Agenda

- French article (probs with current DBMS architectures)
 - Horizontal partitioning (range or hash)
 - vertical partitioning
 - projection index
 - adjusting pagesize
 - considering cache hit rates
 - compression
- MOLAP (multidimensional OLAP)
 - Dwarf
- Stonebraker article (bigger picture on DBMS architectures)
 - different engines for different problems
- Summary

Motivation & Overview

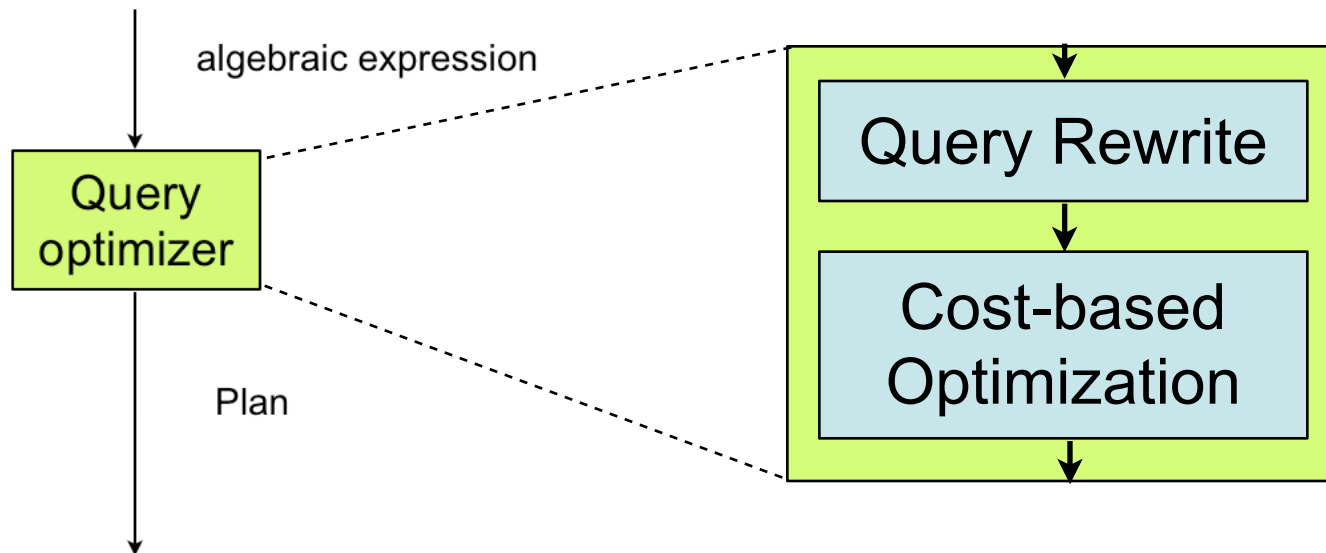


```
SELECT title
FROM A,B
WHERE A.name = 'Hugo' AND A.id = B.dz;
```

$$\Pi_{\text{title}} \left(\sigma_{A.\text{name}='Hugo' \text{ and } A.\text{id}=B.\text{dz}} (A \times B) \right)$$


Query Optimizer

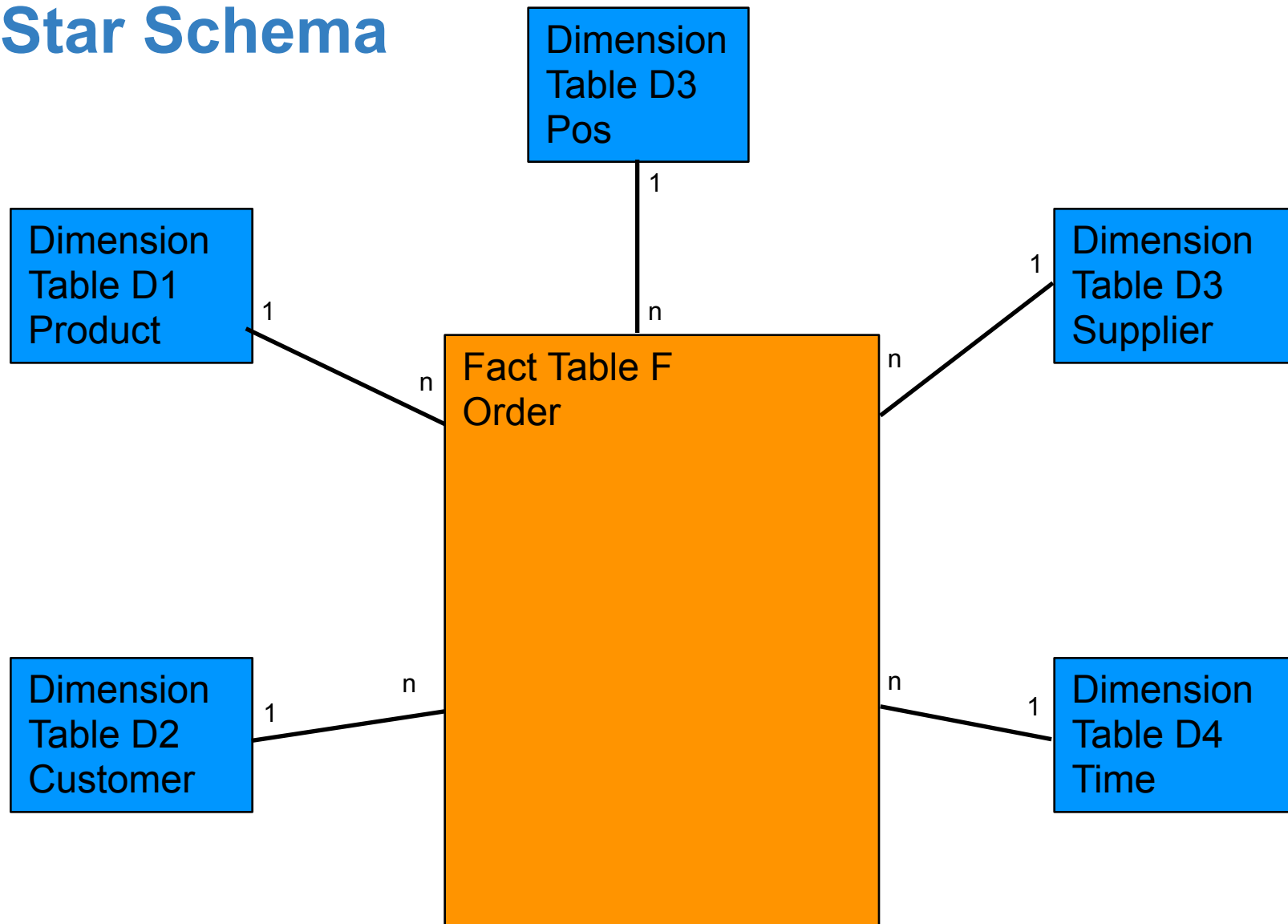
- Query Optimization is divided into two phases



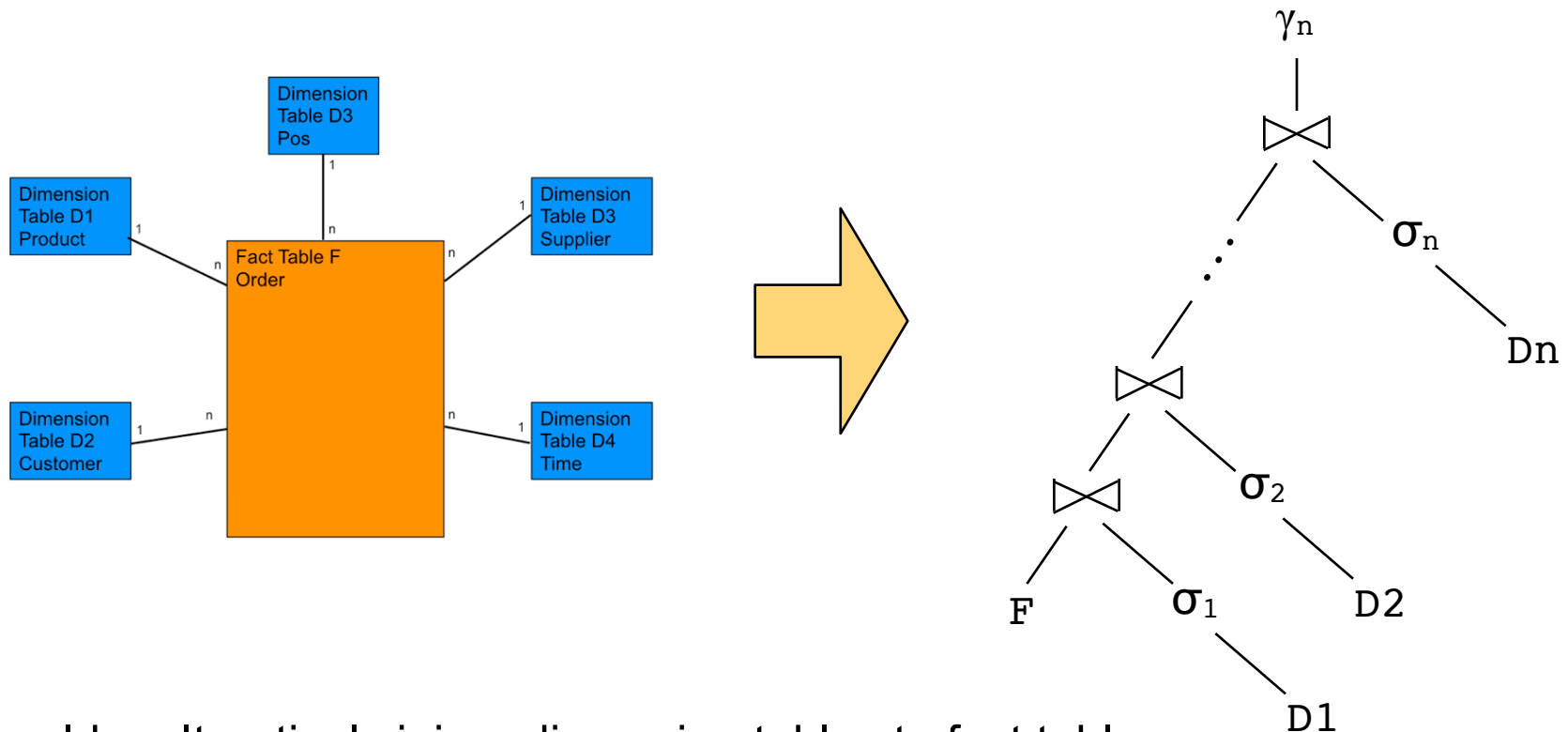
- Query Rewrite
 - rewrite query using rules and heuristics
- Cost-based Optimization
 - Estimate costs of different plans using appropriate cost models
 - Choose plan that has the lowest cost estimate

See [Architecture and Implementation of DBMS lecture](#) for details (every HS)

Star Schema

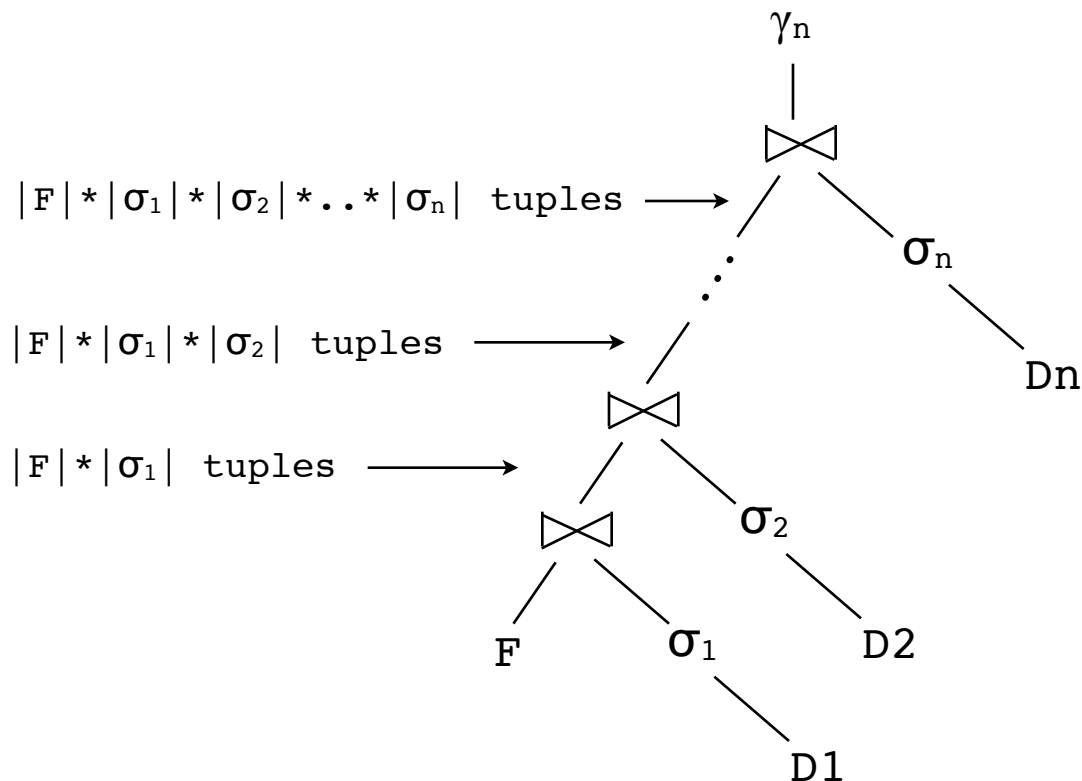


Query Optimization: Naïve Strategy



- Idea: Iteratively join n dimension tables to fact table
- first join may exploit primary index on fact table
- this strategy works well if the first intermediate result, i.e., join of D1 and F is small, following joins should decrease size of intermediate result

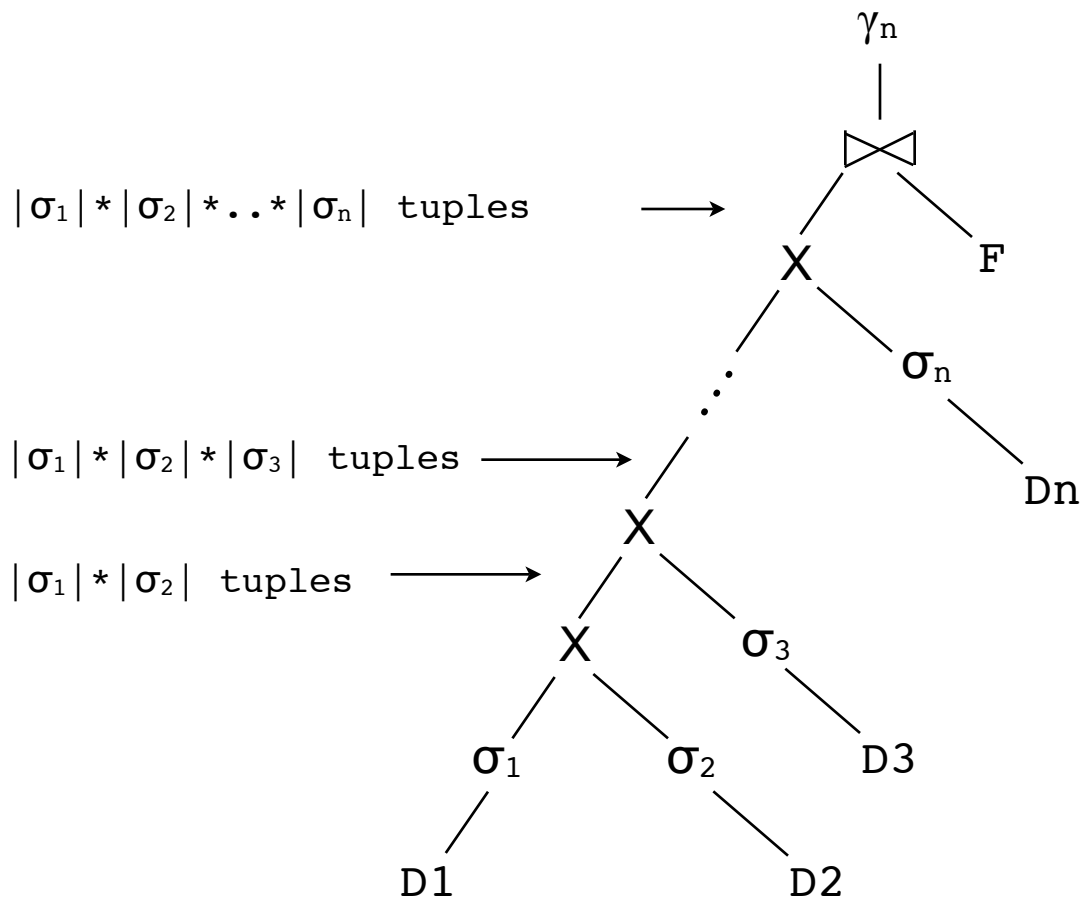
Naïve Strategy: Join Selectivity



- Note: in order to reduce intermediate results, cost based optimizer should reorder dimensions in the plan

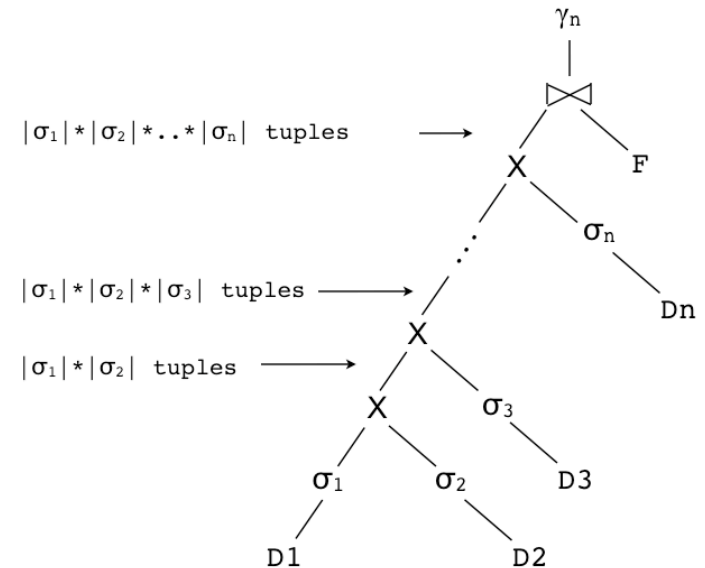
Cross Product Plan

- Idea: create the multidimensional cube and fill that cube with existing values

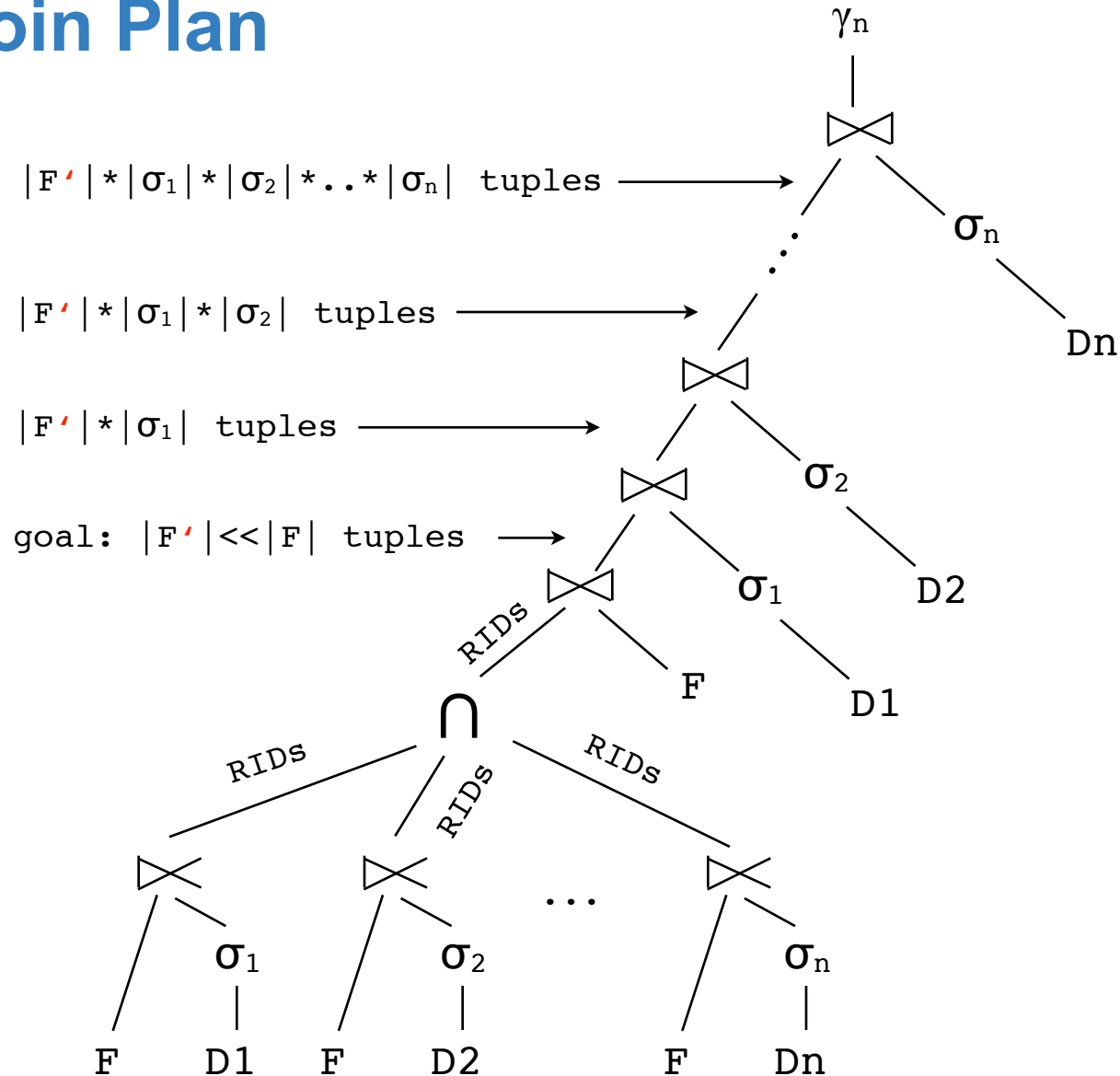


Cross Product Plan

- Idea: create the multidimensional cube and fill that cube with existing values
- this plan may be **very** expensive
 $O(|\sigma_1| * \dots * |\sigma_n|)$, i.e., $O(n^2)$ already for two dimensions
- plan works well if cartesian product on first dimensions is small
- cost based optimizer has to decide whether this plan makes sense
- final join with F may exploit compound index on F
- this plan is also interesting for those cases when client wants to display all cells of the cube anyway no matter whether they contain values or not (CUBE or ROLLUP operators)

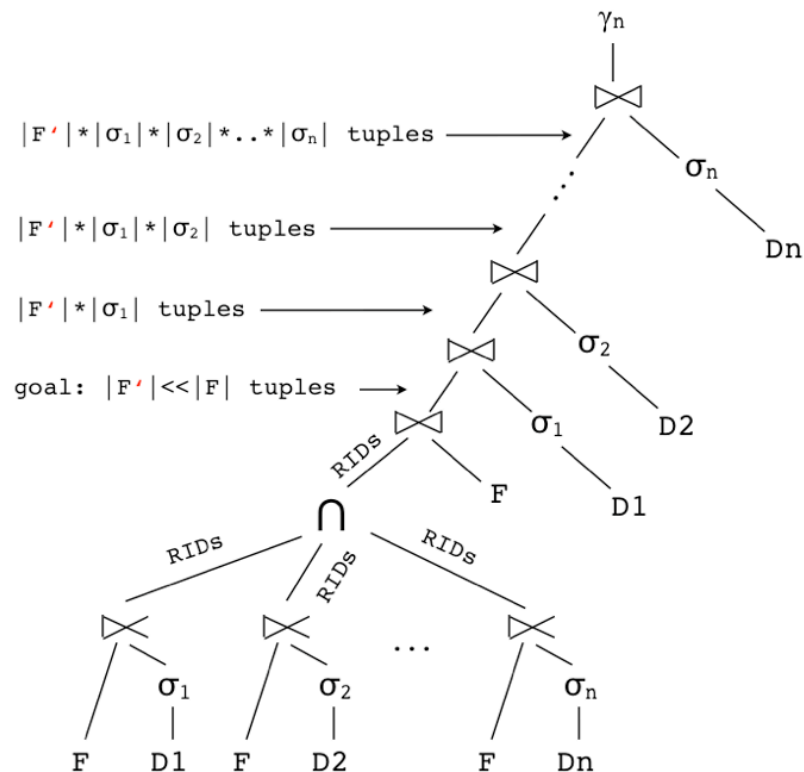


Semi-Join Plan



Semi-Join Plan

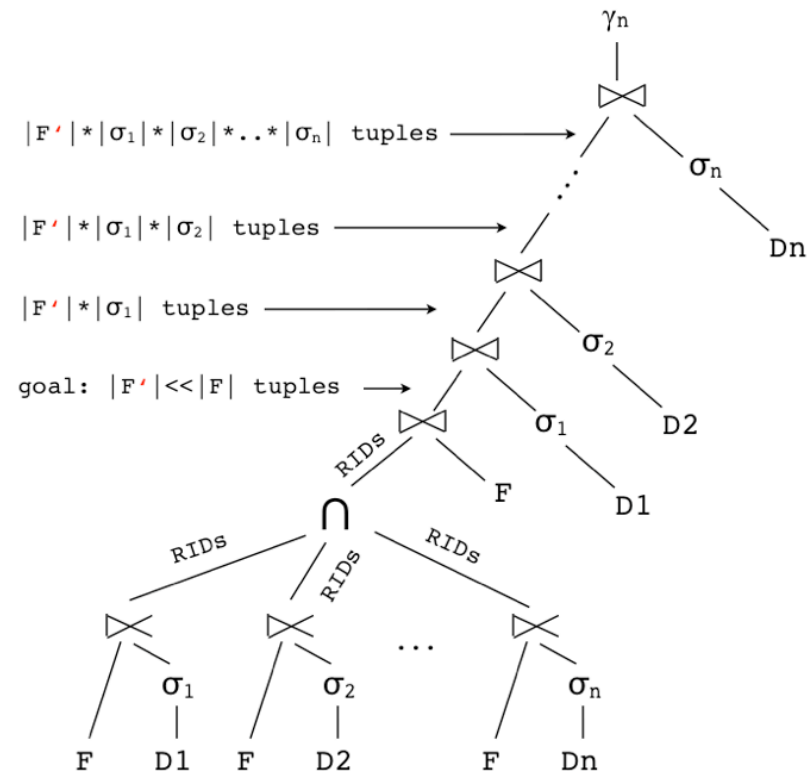
- Idea: reduce size of fact table by preselecting those rows from F that may participate in the join
- preselection is performed using semi-joins of F and D_i
- RID lists of the n semi-joins are then intersected
- final RID list is joined with F
- RID list may be used to enhance the join of the RID list with F by directly accessing tuples in F
- Again: performing n semi-joins may be expensive depending on the selectivity of the selections
- cost based optimizer has to decide whether this plan makes sense



The Story so far

- Different options for star joins

- naive strategy
- Reordering dimensions
- cross products
- semi-joins



- Open issues...

- How can we improve the performance of the semi-joins?
- How can we improve the performance of the selections?
- How can we improve the performance of the join operations?

... and their Solutions

- **How can we improve the performance of the join operations?**
 - materialized views
 - join index (special case of materialized views)
 - partitioning
- **How can we improve the performance of the semi-joins?**
 - bitmap join index
- **How can we improve the performance of the selections?**
 - secondary index: selection bitmap
 - combination of selection bitmap and bitmap join index

OLTP vs. OLAP vs. IR

	OLTP	OLAP	IR
Data Access	read/write	read-mostly	read-mostly
Freshness	up-to-date	stale	stale
Query Access	key-oriented	value-oriented	value-oriented
Indexing	update efficient	query efficient	query efficient
Data	structured	structured	unstructured
Query optimization	heuristic	cost-based	cost-based
Parallelism	inter-query	intra-query	intra-query
Precision&Recall	1	1	≤ 1
Data volume	small-medium	huge	huge

Indexing Constraints

- OLAP allows for much more expensive indexing techniques
- number of indexes is only limited by
 - available disk space
 - time window available to load and index new data into the DWH
- DBA should select appropriate indexes based on query patterns
 - boss's favorite queries should be tuned first...
 - to be on the safe side:
precompute boss's favorite queries at indexing time
(this is Offline Analytical Processing, in other words: Reporting)
- Claim: using the techniques presented in this lecture a DWH can be configured to have interactive (< 5 sec) response times for **all** queries
- Claim: you have to know very well how to use your weapons

Materialized Views

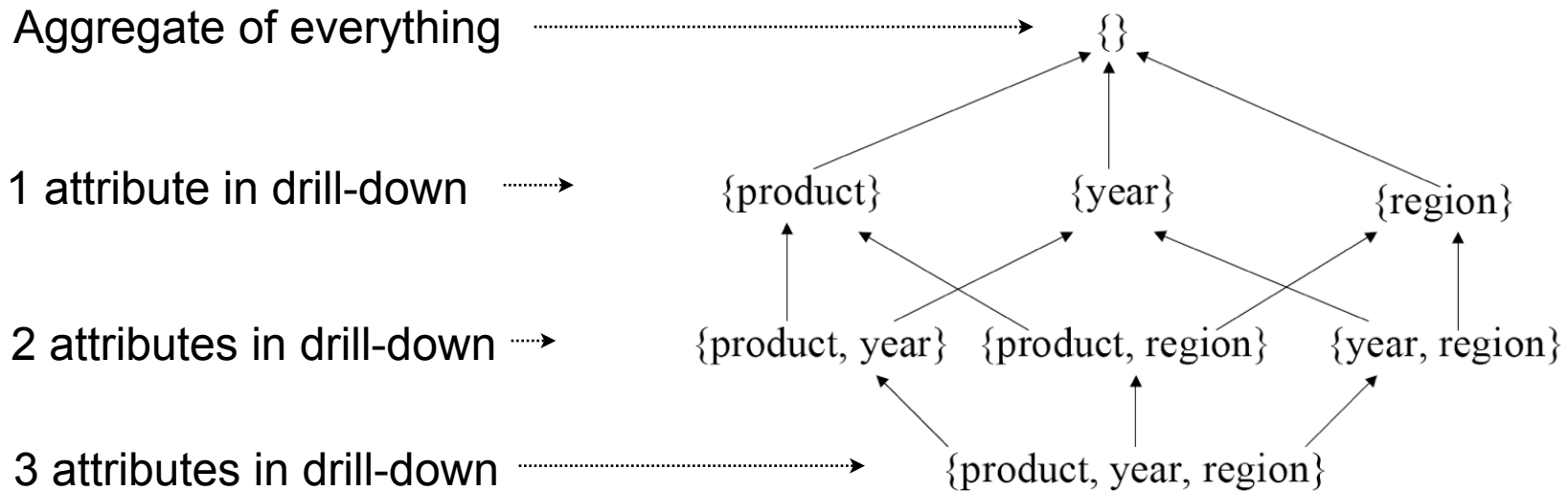
- Idea: materialize (precompute) the result to a join and/or aggregate
- a materialized view is the precomputed result to a query
- the query optimizer may choose to pick one of the materialized views to speed-up query processing
- Trade-off
 - additional speedup for query processing
 - versus
 - additional costs for storing and maintaining materialized views

- Oracle SQL dialect:

```
CREATE MATERIALIZED VIEW matview42
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
```

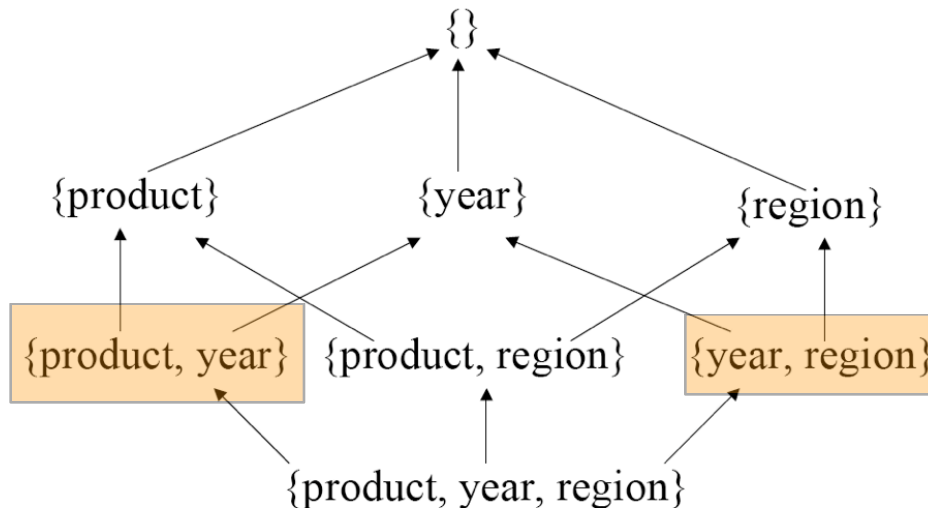
- Problem: how to chose which materialized views to create?

View Lattice



- Derivability: arrows indicate how a view can be derived from other views
- Note: DWH implementations may have hundreds (!) of attributes
- Number of aggregates for n functional independent attributes = 2^n
- For functional dependent attributes polynomial number of aggregates
- => Not all aggregates may be materialized!

Query Answering using Mat. Views Example



- Lets assume we maintain two materialized views (orange boxes)
- All queries referencing aggregates {product}, {year} or {} can be computed using materialized view {product, year}
- All queries referencing aggregates {region}, {year} or {} can be rewritten using materialized view {year, region}

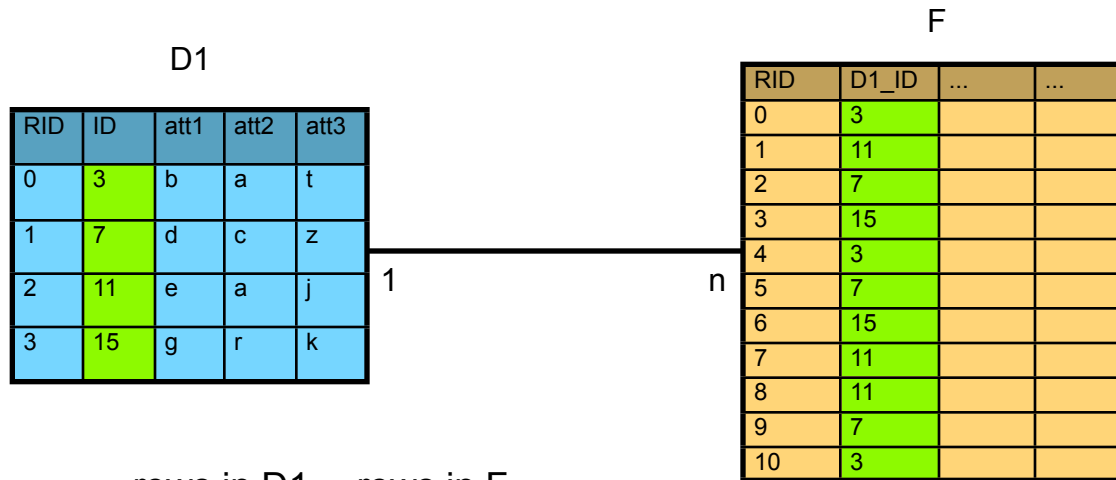
Materialized View Updates

- What happens when new data is loaded into the DWH?
 - depending on the number of materialized views and the view lattice many materialized views may have to be recomputed or updated
- Recompute
 - existing mat view updates may be combined to provide for more efficient recomputation (multi-query optimization MQO)
 - MQO core idea: for two mat views MV1 and MV2 try to find an aggregate MV3 such that both MV1 and MV2 can be derived from MV3
- Update
 - compute view delta
 - merge delta with existing mat view
 - only allowed for certain types, i.e., additive aggregation functions
- Details: see book by Lehner

Join Index

- special case of a materialized view
- For two tables R and S a join index may have three different types
 1. join value $\rightarrow (\{R.RID\}, \{S.RID\})$ i.e., all rows of R and S that have the specified join column value
 2. $R.RID \rightarrow \{S.RID\}$, i.e., all rows (RIDs) of S that join with row R.RID
 3. $R.attribute \rightarrow \{S.RID\}$, i.e., all rows of S that join with those rows in R having the specified attribute value (not the join key)
- If more than two tables are involved, these indices are called domain indices.
- Literature: O'Neil and Graefe 95

Join Index Example: Types 1&2



rows in D1 rows in F



1. join value -> $(\{D1.RID\}, \{F.RID\})$

- Index:

3 -> $(\{0\}, \{0,4,10\})$

7 -> $(\{1\}, \{2,5,9\})$

11 -> $(\{2\}, \{1,7,8\})$

15 -> $(\{3\}, \{3,6\})$

2. D1.RID -> $\{F.RID\}$

- Index:

0 -> $\{0,4,10\}$

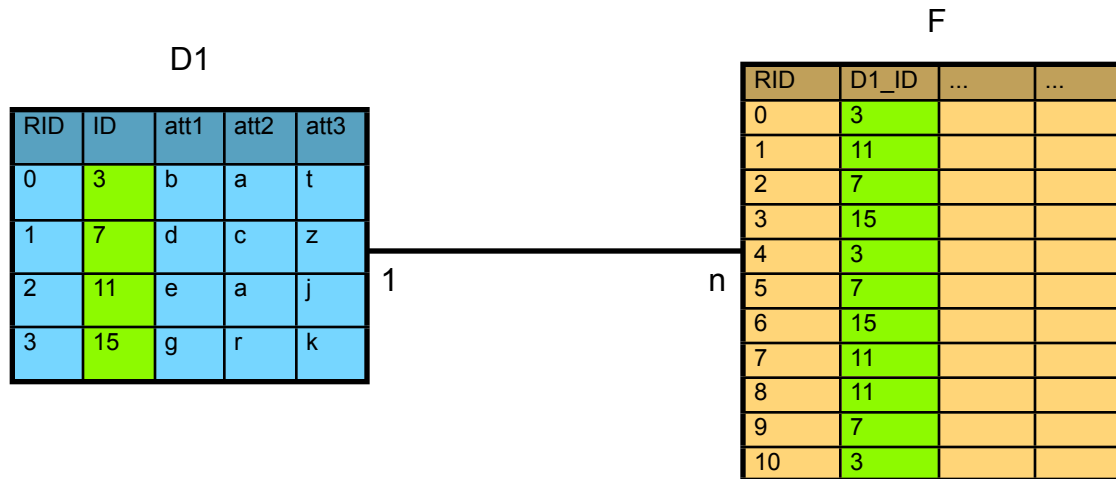
1 -> $\{2,5,9\}$

2 -> $\{1,7,8\}$

3 -> $\{3,6\}$

Can be derived from Type 1

Join Index Example: Type 3



3. D1.att2 -> {F.RID}

- Index:

a -> {0,1,4,7,8,10}

c -> {2,5,9}

r -> {3,6}

Important: we need one bit list per column value. As column values do not have to be unique the total number of bit lists is typically less than |D1|.

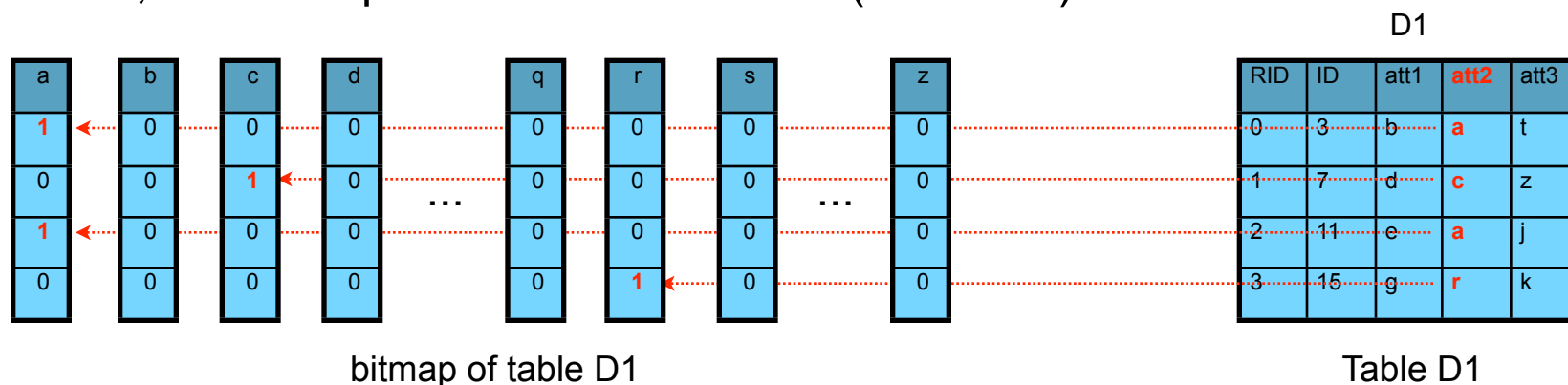
Can be derived by joining inversion on att2->RID with Type 1

Join Index Implementation

- How to represent result RID lists? Two possibilities:
 1. as lists of RIDs
 - uncompressed
value -> [int,int,int, terminator] or:
value -> [number of RIDS, int,int,int]
 - using some sort of compression
value -> [int, diff to previous value, diff to previous value]
or any other suitable techniques
 - again many similarities to IR technology
 - see [ArchDBMS lecture](#) for details
 2. as bitmaps
 - works if total number of RIDs is bounded and the number of attribute values is “low“
 - will be discussed in the following

Value Bitmap

- Lets assume that the domain of att2 is {a,...,z}.
- Lets assume that the number of RIDs is bounded by n.
- Then, the bitmap on att2 is defined as (here n=3)



- This means, for each attribute value v of attribute att2 we define a separate bit list BL_v of length $n+1$.
- If row RID has value v for attribute att2 then $BL_v[RID] := 1$, otherwise $BL_v[RID] := 0$.
- Size of the bitmap: $|\{a, \dots, z\}| * (n+1) = 26 * 4 = 104 \text{ Bits} = 13 \text{ Bytes}$

Bitmap Operations

- Why are bitmaps a good idea?
- Assume we have multiple bitmaps for different attributes of a table D1
- AND
condition $D1.att1 == 42 \text{ AND } D1.att4 == 47$
computation: lookup bit lists in both bitmaps and compute boolean **and**
- OR
condition $D1.att1 == 42 \text{ OR } D1.att1 == 43$
computation: lookup both bit lists in bitmap for att1 and compute boolean **or**
- NOT
condition $D1.att1 != 42$
computation: lookup bit list in bitmap for att1 and compute boolean **not**
- etc.
- bit operations are very efficient as they combine 32/64 bits in a single CPU operation

Bitmap Compression

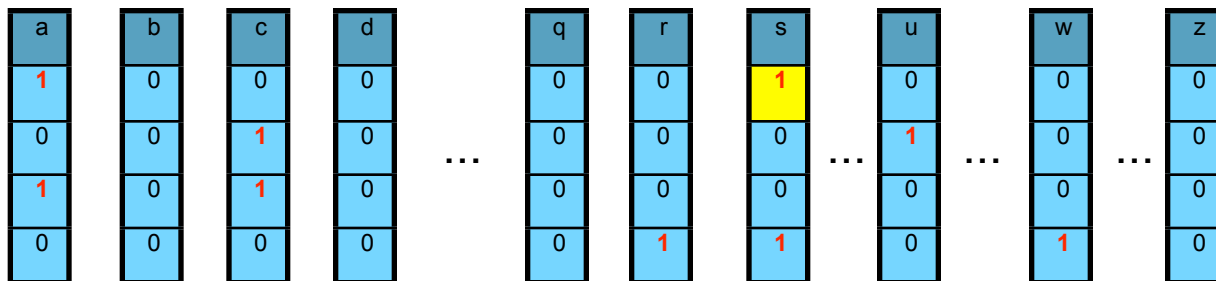
- Many entries of a bit map may be zero
- Therefore compression techniques may be applied
- At query time bitmap operations then have to decompress the bitmaps
- Techniques
 - run length encoding RLE
 - approximative bitmaps
 - coarse bit maps: similarities to kd-tries (see [ArchDBMS lecture](#))
 - bloom-filter: also applicable for distributed join processing to reduce load of shipped list for semi-join
(see: Comp. Surv. article by Donald Kossmann, 2001)

Bloom-Filter

- Idea: for an RID domain of size $n+1$ use a bitmap of size $m \ll n+1$
- Apply a hash-function to compute the position in the reduced bit map, i.e., $\text{position} := \text{hash}(\text{RID}) \% m$
- Standard bit list (see above):
If RID has value v for attribute $\text{att2} \Leftrightarrow \text{BL}_v[\text{RID}] := 1$.
- Bloom filter bit list
If RID has value v for attribute $\text{att2} \Rightarrow \text{BL}_v[\text{RID}] := 1$.
- This means that
 $\text{BL}_v[\text{RID}] == 1 \not\Rightarrow$ RID has value v for attribute att2
(RID **may** have value v for attribute att2)
 $\text{BL}_v[\text{RID}] == 0 \Rightarrow$ RID does not have value v for attribute att2
- For this reason bloom-filters compute a superset of the exact query.
- False drops have to be eliminated in consecutive steps of query processing.

Bloom-Filter Example

D1



m=4

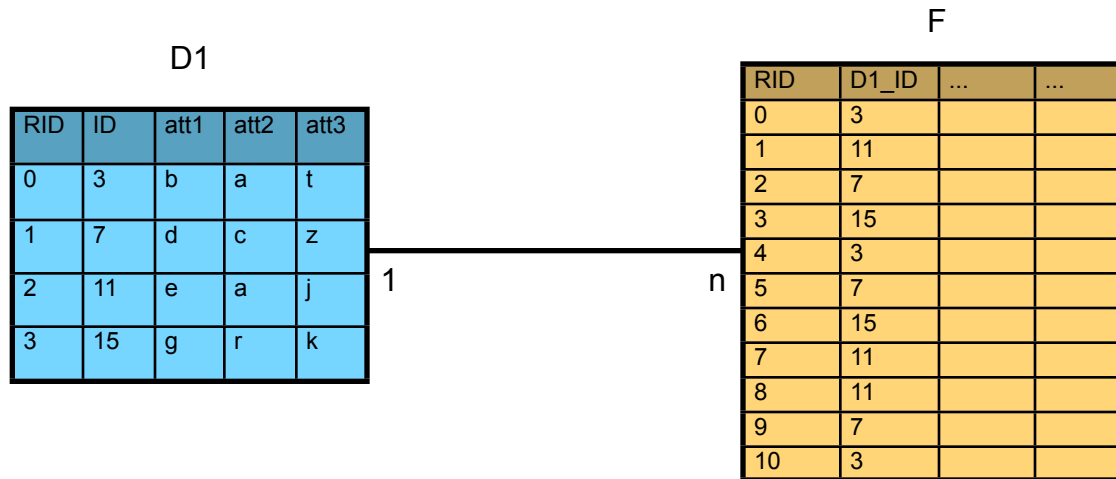
RID	ID	att1	att2	att3
0	3	b	a	t
1	7	d	c	z
2	11	e	a	j
3	15	g	r	k
4	s	...
5			u	
6			c	
7			w	
8			s	

- Simple hash function: $\text{hash}(\text{RID}) := \text{RID} \% m$
(please do not use this in practice, see introductory lectures, book by Widmayer et.al., or Cormen et.al. for good hash functions)
- RIDs 4 and 8 map to the same bit $\text{BL}_{s'}[\text{hash}(\text{RID})] := 1$
- From, e.g., $\text{BL}_{r'}[0] == 0$ we know that: no tuple with RID having hash $(\text{RID}) == 0$ may have the value of att2 set to 'r'
- However from $\text{BL}_{s'}[0] == 1$ we cannot determine which of the RIDs 4,8 or any multiple of 4 is set to 1

Switching Bitmap Representations

- A small note:
depending on the cardinality of an attribute value, RID list representations may be switched
- For example
 - attribute value frequent: bitmaps may be a good choice
 - attribute value rare: RID lists may be better
- A clever index implementation may switch between the two representations at any time.
- This observation holds for B⁺-trees as well as for inverted lists.
- Therefore, an ordered RID list can be regarded as a way to compress a bitmap (and vice-versa).

Bitmap Join Index Example: Type 3



3. D1.att2 -> {F.RID}

- Logical Index:

a -> {0,1,4,7,8,10}

c -> {2,5,9}

r -> {3,6}

2. D1.RID -> {F.RID}

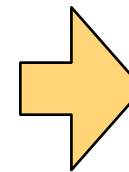
- Bitmap Join Index:

0	1	2	3
1	0	0	0
0	0	1	0
0	1	0	0
0	0	0	1
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0
0	0	1	0
0	1	0	0
0	0	1	0
1	0	0	0

3. D1.att2 -> {F.RID}

- Bitmap Join Index:

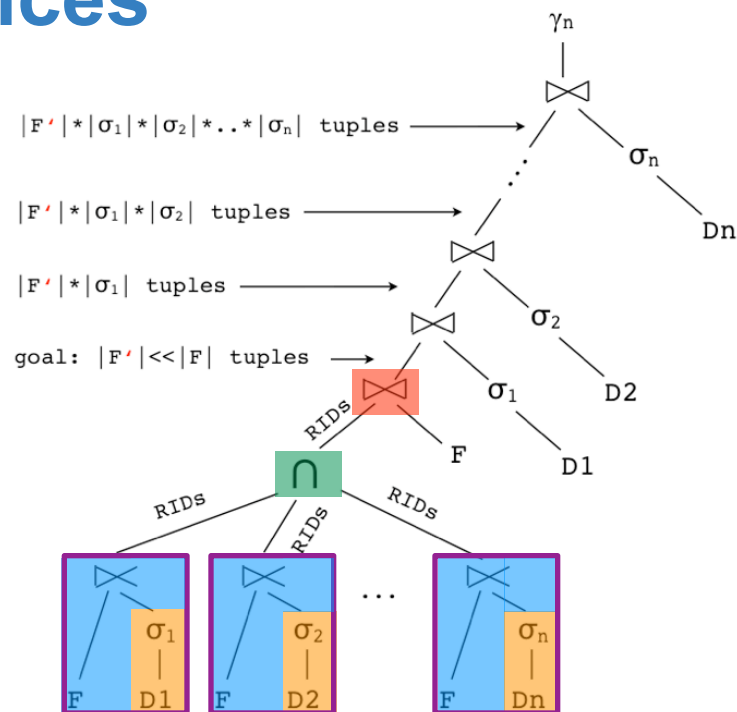
a	c	r
1	0	0
1	0	0
0	1	0
0	0	1
1	0	0
0	1	0
0	0	1
1	0	0
1	0	0
0	1	0
1	0	0



Can be derived by joining inversion
on att2->RID with Type 1

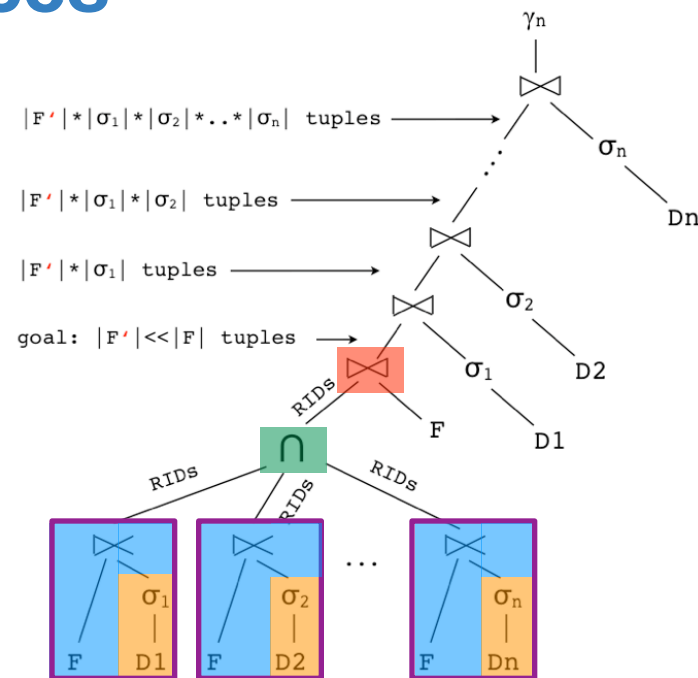
Star Join using Bitmap Indices

- Remember the semi-join plan?
- Now we can use the same plan, but:
 - replace selections by **value bitmaps**
 - replace semi-joins by **Type 2 bitmap join indexes**
 - alternatively: replace selections and semi-join by **Type 3 bitmap join indexes**
- If more than one attribute is selected for a dimension D_i , their bitmaps are combined using a boolean OR.
- The intersection is implemented by a **large merge (AND)** of the respective bitmaps.
- The result from that merge operation determines the tuples from F that have to be read to perform the actual join.
- The qualifying tuples can again be fetched from F using an indexed **nested-loops join** accessing the index on F .



Star Join using Bitmap Indices

- This plan will be very efficient if the bitmaps help to reduce the number of tuples that have to be considered from F.
- In addition, the selectivity on the dimension tables should be high.
- Example
 - assume that each predicate has a selectivity of 0.01 (=1%)
 - if all N predicates are statistically independent of each other, we estimate an overall selectivity of 10^{-2N}
 - for N=3 (three predicates) and a fact table containing 10^8 tuples it follows that approximately only 100 RIDs `survive` the **bitmap operation**.
 - **fetching** 100 rows independently using an index nested-loops join accessing the index on F should be faster than reading the entire table F
- However: we need a cost based optimizer to make this decision.



How can a Star Join be improved further?

- Horizontal partitioning (range or hash)
- vertical partitioning
- projection index
- adjusting pagesize
- considering cache hit rates
- compression
- multi-dimensional index

Horizontal Partitioning

- Idea: split fact table F into disjoint horizontal partitions, i.e, pieces have the same schema as F , sum of all pieces equals F
- if query contains a predicate that allows to skip certain partitions of F , less tuples from F have to be considered for query processing
- partitioning allows to add or remove data to/from F in a single operation
- partitioning allows to distribute data on multiple machines to allow for concurrent computation of queries

Horizontal Partitioning: by Range

- Idea: for an attribute $F.att_i$, split F into partitions F_i such that all elements in F_i belong to a certain range
- Example: Partition by time
 - Assume F contains an attribute 'year'
 - create partitions F_{year} such that F_{year} contains tuples of that year
 - F_{2004} , F_{2005} , F_{2006} , F_{2007} , etc.
 - Any query specifying a time attribute may be restricted to consider only data in some of the partitions.
 - ```
SELECT ...
FROM ...
WHERE ... F.year=2006 ...
```

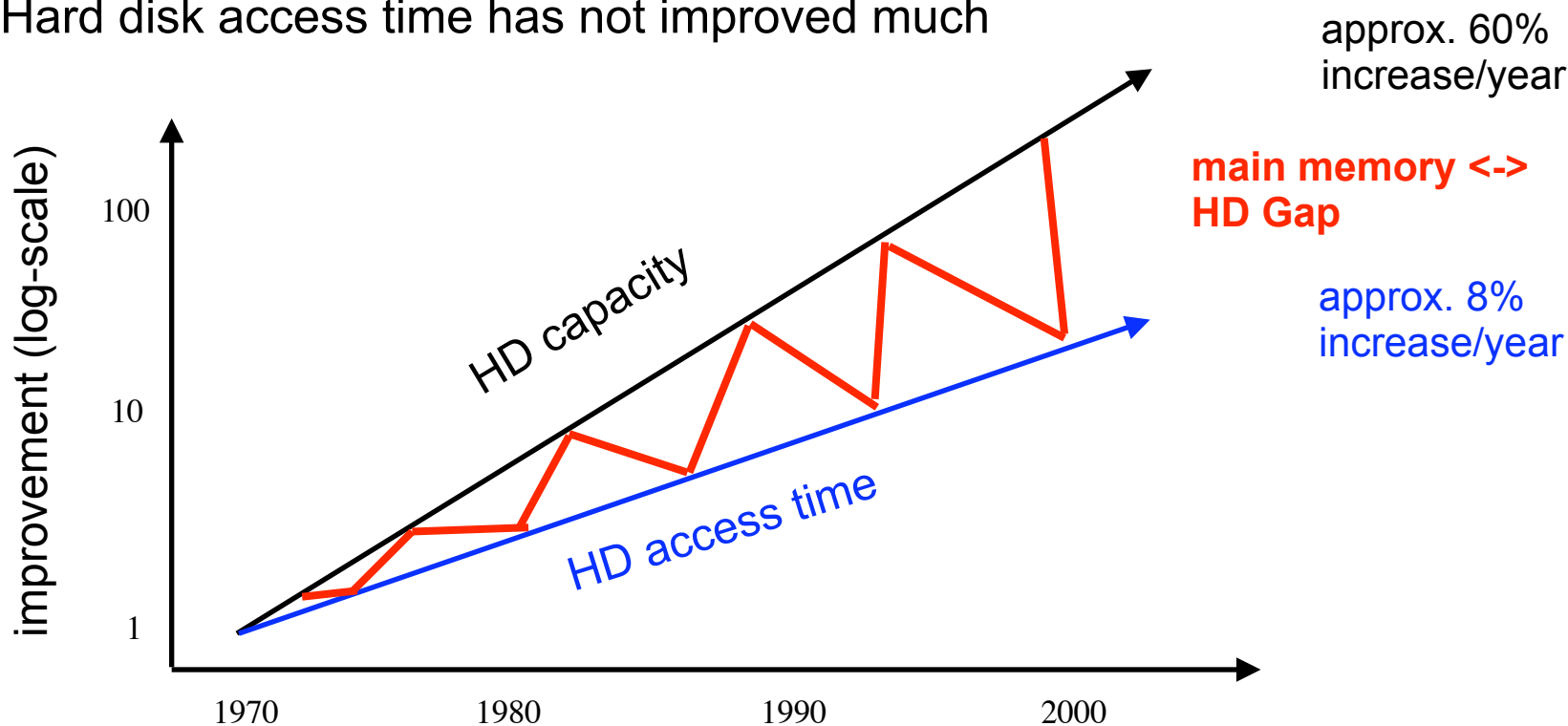
This query only has to consider data from partition  $F_{2006}$
- Oracle SQL dialect: `PARTITION BY RANGE(year)`

# Horizontal Partitioning: by Hash

- Idea: for an attribute  $F.att_i$ , split  $F$  into partitions  $F_i$  such that all elements in  $F_i$  belong to a certain hash bucket
- $i$ th hash bucket is determined by computing  
partition := hash( <some key columns> ) % number of partitions
- Example: Partition by hash
  - Assume  $F$  contains an attribute 'year'
  - create four partitions  $F_i$  such that  $F_i$  contains tuples of hash(year)
  - $F_0, F_1, F_2, F_3$
- Oracle SQL dialect: `PARTITION BY HASH(year) PARTITIONS 4;`
- Horizontal partitioning also useful for intra-query parallelism

# Hardware Development: Hard disks

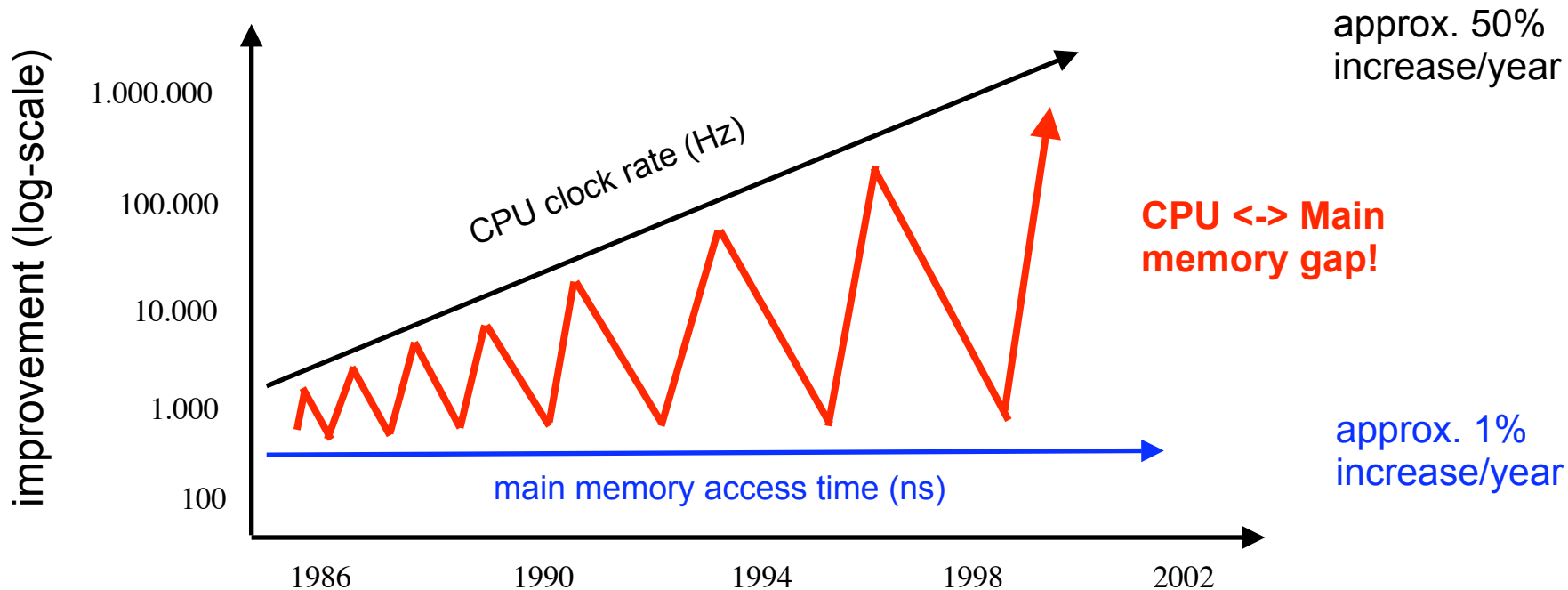
- \$/MB ratio has improved considerably over the years
- Hard disk access time has not improved much



HD capacity vs. HD access time

# Hardware Development: CPU and Main Memory

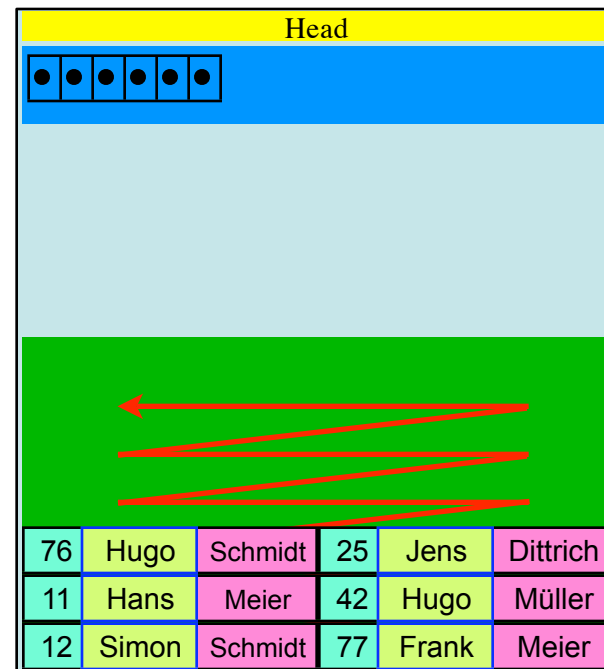
- \$/GHz ratio has improved considerably over the years
- main memory access time has not improved much



CPU clock rate vs. main memory access time

# Standard n-ary Storage Model (NSM)

| Key | fname | lname    |
|-----|-------|----------|
| 77  | Frank | Meier    |
| 12  | Simon | Schmidt  |
| 42  | Hugo  | Müller   |
| 11  | Hans  | Meier    |
| 25  | Jens  | Dittrich |
| 76  | Hugo  | Schmidt  |
|     |       |          |



- records are stored sequentially
- all attributes of a record are stored together

Question: what happens in the memory hierarchy?

# Storage Hierarchy

## Capacity, Access time

Bytes  
1-5 ns

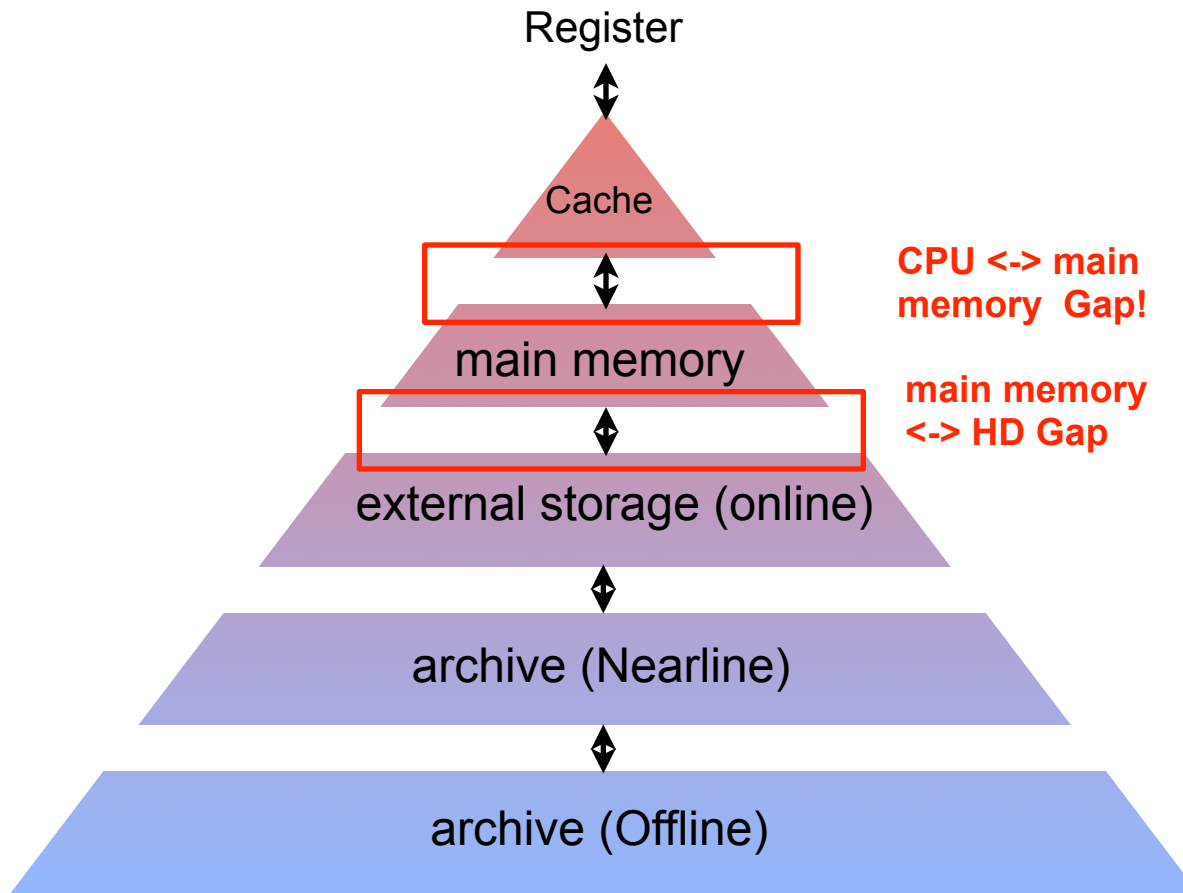
K-M Bytes  
2-20 ns

M-G Bytes  
50-100 ns

G Bytes  
ms

T Bytes  
sec

T Bytes  
sec-min



## Control, Granularity

Program/Compiler  
1-8 Bytes

Cache-Controller  
8-128 Bytes

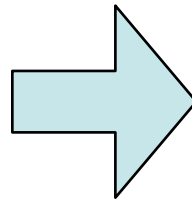
Operating system  
1-16 K Bytes

User/Operator  
M Bytes (Files)

User/Operator  
M Bytes (Files)

# Decomposition Storage Model (DSM)

| RID | Key | fname | Iname    |
|-----|-----|-------|----------|
| 1   | 77  | Frank | Meier    |
| 2   | 12  | Simon | Schmidt  |
| 3   | 42  | Hugo  | Müller   |
| 4   | 11  | Hans  | Meier    |
| 5   | 25  | Jens  | Dittrich |
| 6   | 76  | Hugo  | Schmidt  |
|     |     |       |          |



| RID | Key |
|-----|-----|
| 1   | 77  |
| 2   | 12  |
| 3   | 42  |
| 4   | 11  |
| 5   | 25  |
| 6   | 76  |
|     |     |

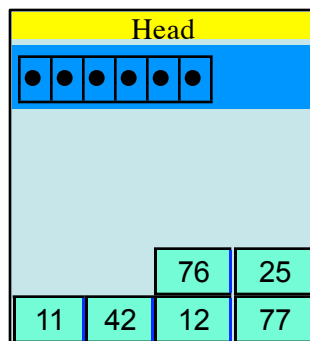
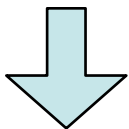
| RID | fname |
|-----|-------|
| 1   | Frank |
| 2   | Simon |
| 3   | Hugo  |
| 4   | Hans  |
| 5   | Jens  |
| 6   | Hugo  |
|     |       |

| RID | Iname    |
|-----|----------|
| 1   | Meier    |
| 2   | Schmidt  |
| 3   | Müller   |
| 4   | Meier    |
| 5   | Dittrich |
| 6   | Schmidt  |
|     |          |

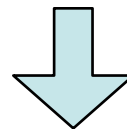
- split table into set of two-column tables
- alternatively: split table into one-column table storing the RID implicitly (array-like representation)

# Decomposition Storage Model (DSM)

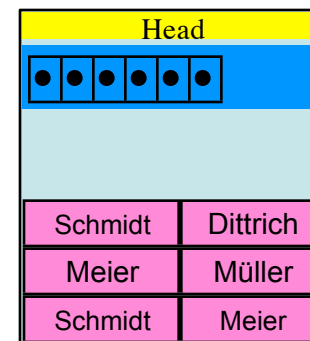
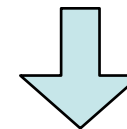
| RID | Key |
|-----|-----|
| 1   | 77  |
| 2   | 12  |
| 3   | 42  |
| 4   | 11  |
| 5   | 25  |
| 6   | 76  |
|     |     |



| RID | fname |
|-----|-------|
| 1   | Frank |
| 2   | Simon |
| 3   | Hugo  |
| 4   | Hans  |
| 5   | Jens  |
| 6   | Hugo  |
|     |       |



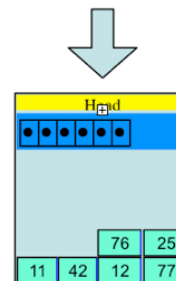
| RID | Iname    |
|-----|----------|
| 1   | Meier    |
| 2   | Schmidt  |
| 3   | Müller   |
| 4   | Meier    |
| 5   | Dittrich |
| 6   | Schmidt  |
|     |          |



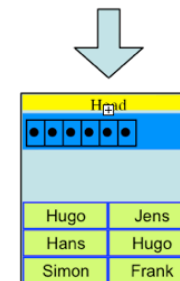
# Decomposition Storage Model (DSM)

- optimized for accessing few attributes
- Advantage: very efficient when only few attributes need to be accessed
- Disadvantage: inefficient when many attributes are accessed
- Disadvantage: rows distributed to many pages

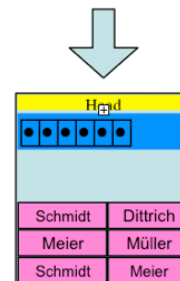
| RID | Key |
|-----|-----|
| 1   | 77  |
| 2   | 12  |
| 3   | 42  |
| 4   | 11  |
| 5   | 25  |
| 6   | 76  |



| RID | fname |
|-----|-------|
| 1   | Frank |
| 2   | Simon |
| 3   | Hugo  |
| 4   | Hans  |
| 5   | Jens  |
| 6   | Hugo  |



| RID | lname    |
|-----|----------|
| 1   | Meier    |
| 2   | Schmidt  |
| 3   | Müller   |
| 4   | Meier    |
| 5   | Dittrich |
| 6   | Schmidt  |



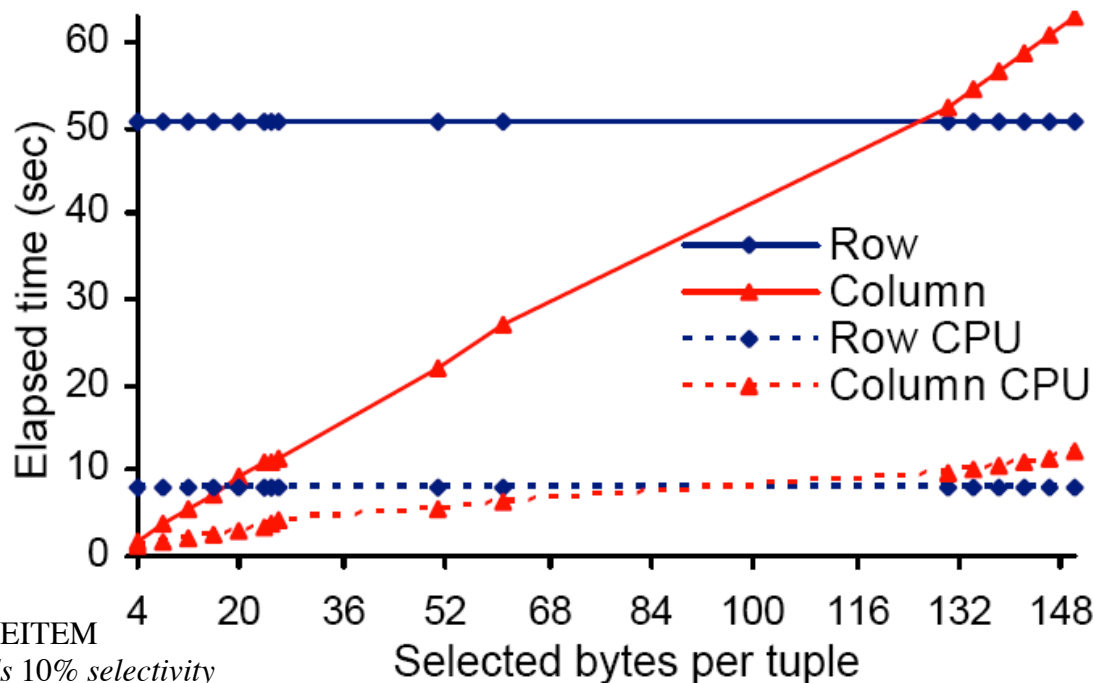
## Literature

- Don S. Batory: On Searching Transposed Files. ACM Trans. Database Syst. 1979.
- George P. Copeland, Setrag Khoshafian: A Decomposition Storage Model. SIGMOD 1985

# Column Stores

- Several Products
  - Sybase IQ (since early 90ies)
  - Applix
  - MonetDB (main memory)
  - SAP BI Accelerator (main memory)
  - Vertica
  - ...
- Will become more and more important given current hardware trends
- Student in 1995: What is a tape?
- Student in 2010: What is a hard disk?

# Some Recent Study on Column Stores



- Tuple width: 150 bytes, 16 attributes, 9.5GB table
- Literature: Performance Tradeoffs in Read-Optimized Databases, Harizopoulos et.al, VLDB 2006

# Projection Index

- Idea
  - stick with NSM
  - **but:** create additional **redundant** projections on single attributes
  - projections represent vertical partitions of DSM
- Projection Index may be used if all values have to be read, otherwise the NSM representation will be used
- Literature: O'Neil and Quass 97

# Projection Index Details

- assume `foo` is a column of F
- the projection index then consists of the sequence of column values from `foo` ordered by RID (holes may exist for unused row numbers)
- if column `foo` is 4 bytes in length, then we can fit 1000 values from `foo` on a single 4 KByte disk page
- for any given row number  $n = m(r)$  in F we can access the proper disk page  $p$  and slot  $s$  to retrieve the appropriate `foo` value with a simple calculation:

$$p = n / 1000$$

$$s = n \% 1000$$

- the row number of a given page  $p$  and slot  $s$  is calculated as:

$$n = 1000 * p + s$$

# Bit-sliced Index Example

Fact table

| F  | foo |
|----|-----|
| 3  |     |
| 1  |     |
| 4  |     |
| 10 |     |
| 0  |     |
| 1  |     |
| 5  |     |
| 11 |     |
| 5  |     |
| 12 |     |
| 6  |     |

binary representation

| foo        |
|------------|
| 0...000011 |
| 0...000001 |
| 0...000100 |
| 0...001010 |
| 0...000000 |
| 0...000001 |
| 0...000101 |
| 0...001011 |
| 0...000011 |
| 0...001010 |
| 0...000110 |

bit-sliced index

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |

- for this example only 1/8 of the original space for a PI is required
- for FTS on vertical partition this translates roughly into **times 8** performance improvements!
- Assuming 100 million rows in projection index this is 50 MB versus 400 MB to read (versus possibly dozens of gigabytes for reading F)

# Bit-sliced Index

- Assumptions
  - `foo` is a column of  $F$  of a countable number type
  - domain of numbers is limited, all numbers in  $1, \dots, N$
- To represent  $1, \dots, N$  numbers we need  $n = \lceil \log_2 N \rceil$  bits
- Idea
  - for column `foo` define a bit list  $D(\text{RID}, i)$  for each bit of each value
  - $D(\text{RID}, 0) = \text{true} \Leftrightarrow 2^0$  bit of value `foo` of row  $\text{RID}$  is set
  - $D(\text{RID}, 1) = \text{true} \Leftrightarrow 2^1$  bit of value `foo` of row  $\text{RID}$  is set
  - ...
  - $D(\text{RID}, i) = \text{true} \Leftrightarrow 2^i$  bit of value `foo` of row  $\text{RID}$  is set
  - ...
  - $D(\text{RID}, n) = \text{true} \Leftrightarrow 2^n$  bit of value `foo` of row  $\text{RID}$  is set

# Compressed Vertical Partitions

Fact table

| F | foo |
|---|-----|
|   | 3   |
|   | 1   |
|   | 4   |
|   | 10  |
|   | 0   |
|   | 1   |
|   | 5   |
|   | 11  |
|   | 5   |
|   | 12  |
|   | 6   |

binary representation

| foo        |
|------------|
| 0...000011 |
| 0...000001 |
| 0...000100 |
| 0...001010 |
| 0...000000 |
| 0...000001 |
| 0...000101 |
| 0...001011 |
| 0...000011 |
| 0...001010 |
| 0...000110 |

compressed vertical partition

| foo  |
|------|
| 0011 |
| 0001 |
| 0100 |
| 1010 |
| 0000 |
| 0001 |
| 0101 |
| 1011 |
| 0011 |
| 1010 |
| 0110 |

- Alternative: instead of cutting a vertical partition into bit-slices compress it directly into a single compressed vertical partition
- same effect as bit-sliced partitions
- many different encoding schemes exist for compressing (see Managing Gigabytes book)

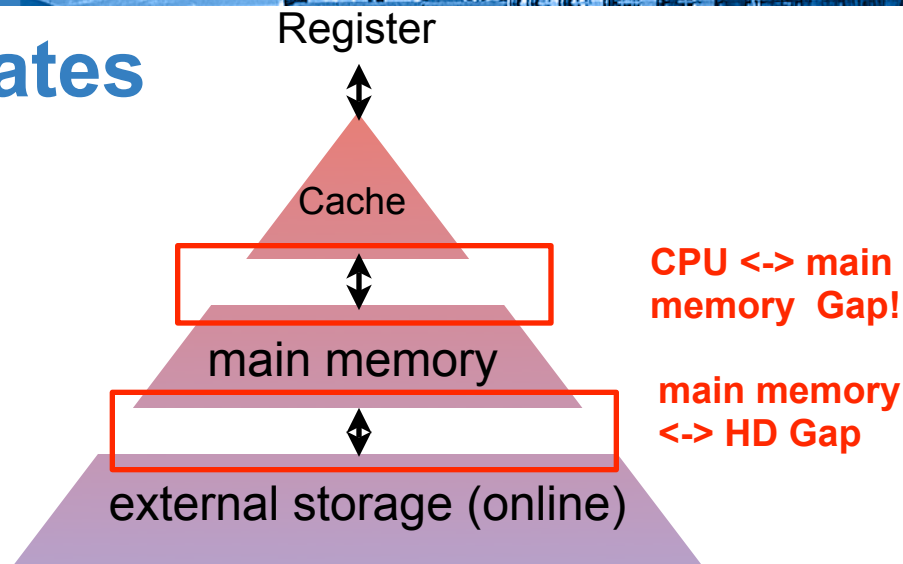
# Pagesize

- OLTP works best with small page sizes
  - space- and update-efficient system
  - may require lots of seek operations
  - high data fragmentation
- OLAP works best with large page sizes
  - query-efficient system
  - OLAP queries may require large amounts of data to be read to answer a query
  - avoid seek operations whenever possible
  - low data fragmentation

# Cache Hit Rates

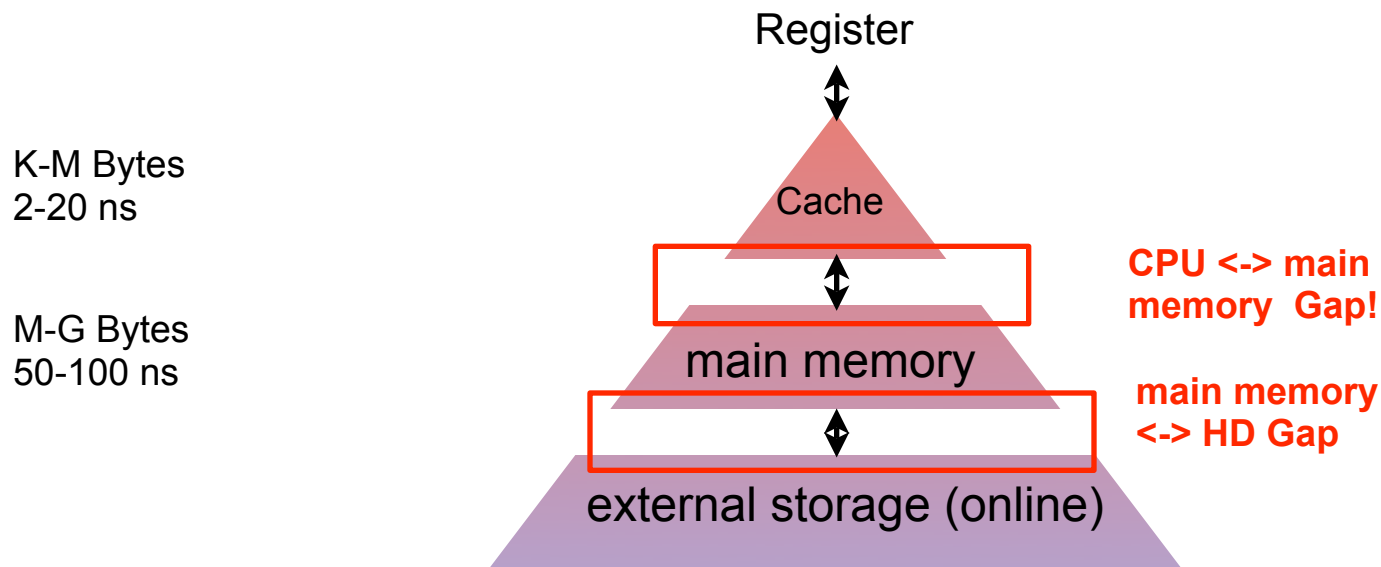
K-M Bytes  
2-20 ns

M-G Bytes  
50-100 ns



- Lesson learned so far: Vertical partitions + compression lowers I/O-cost on the hard disk(s) considerably
- Now: vertical partitions also lowers **I/O-cost in main memory!**
- I/O-cost in main memory?
  - data not in cache, cache miss: read it from main memory
  - access to main memory is much slower than accessing cache
  - CPU has to wait until data is loaded into the cache
  - also: many CPUs are able to prefetch (either automatically guessing access patterns or through pragma directives)

# Cache Hit Rates: Goals



- Ideal world: only `relevant` data should go into the cache
- `relevant` data: data that has to be touched by the CPU
- everything else just waists cache space or stalls CPU
- Algorithms should be designed to **not** waist cache space
- Performance gains may be tremendous (factors)
- Products: Sybase IQ, Applix TM1, SAP BI Accelerator, ...

# Cache Hit Rates: Approaches

- 2 approaches to optimize cache hit rates
  - cache-oblivious algorithms
    - no detailed knowledge (obliviousness) about memory hierarchy and size of memory levels
    - generic approach
    - not tailored for a specific architecture (see lecture by Prof. Widmayer)
  - cache-aware algorithms
    - detailed knowledge (awareness) about memory hierarchy and size of memory levels
    - specialized approach
    - not tailored for a specific architecture (see hardware lecture by Prof. Thiele)

# MOLAP: Multidimensional OLAP

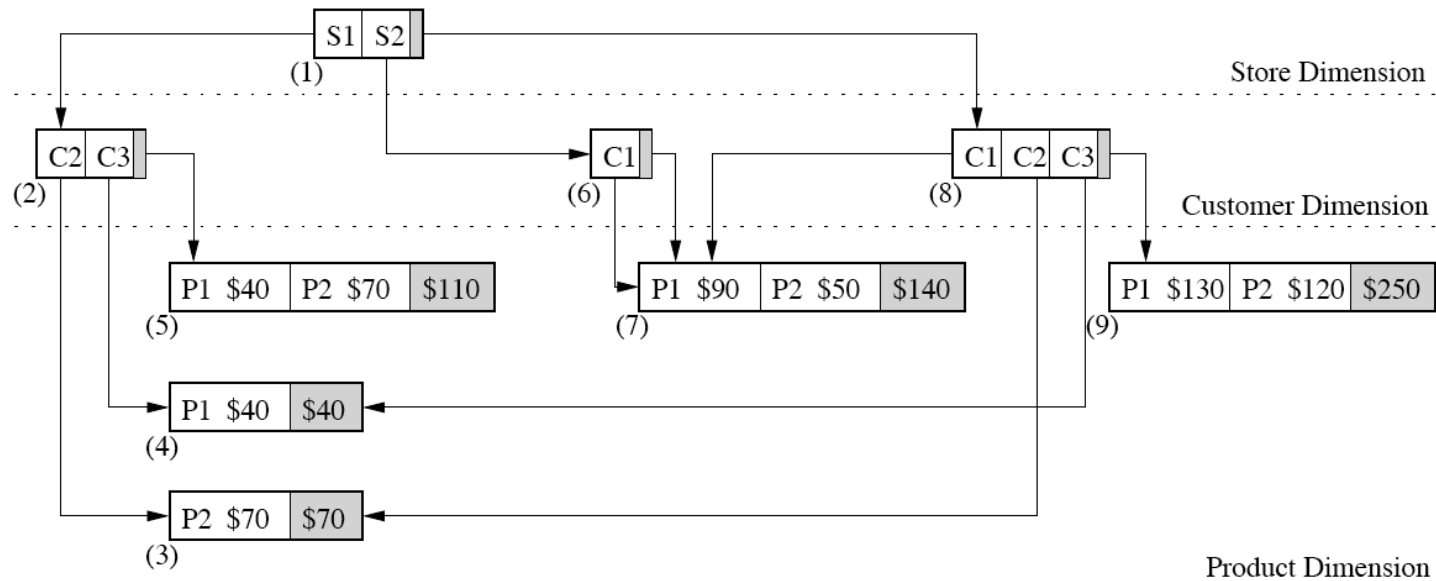
- Non-relational approach to OLAP
- Core Idea
  - Do not use DBMS for query processing
  - invent clever index structures that
    - materialize (almost) all aggregates
    - store aggregates in a highly compressed manner
  - at query time just look-up aggregates values or do light-weight post-aggregation

# MOLAP Example: Dwarf

## Fact table

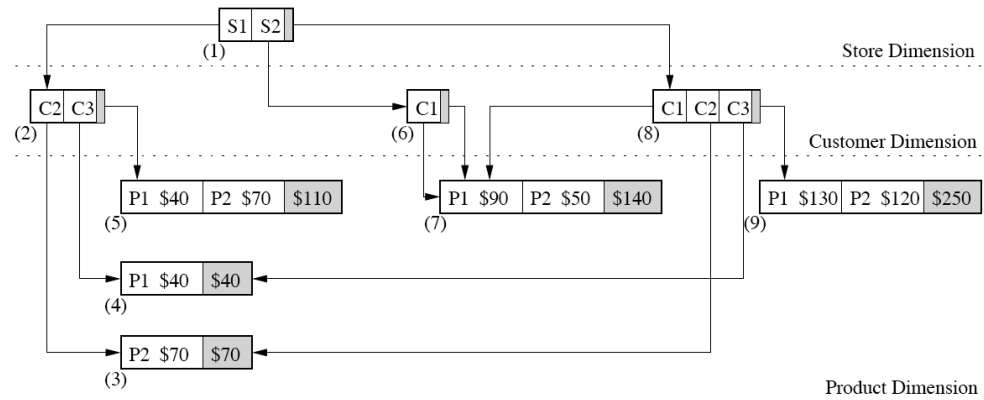
| Store | Customer | Product | Price |
|-------|----------|---------|-------|
| S1    | C2       | P2      | \$70  |
| S1    | C3       | P1      | \$40  |
| S2    | C1       | P1      | \$90  |
| S2    | C1       | P2      | \$50  |

## Dwarf Index



- Literature: Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, Yannis Kotidis: Dwarf: shrinking the PetaCube. SIGMOD Conference 2002: 464-475

# Dwarf: Worst Case Analysis



- Assume each dimension has cardinality  $N$ . Then each level has a fan-out of  $N+1$ .
- The number of nodes is then given by  $(N+1)^{D-2} + 1$  ( $D \geq 3$ ).
- The number of leaves is given by  $(N+1)^{D-1}$  ( $D \geq 1$ ), i.e., number of leaves dominates
- $(N+1)^{D-1}$  may be huge for large cardinalities and high dimensions
- Example:  $N=1000$ ,  $D=6$ , number of leaves =  $10^{15}$ .
- Therefore, this approach is not **guaranteed** to scale well.
- (However approach may work for certain scenarios)

# MOLAP: Discussion

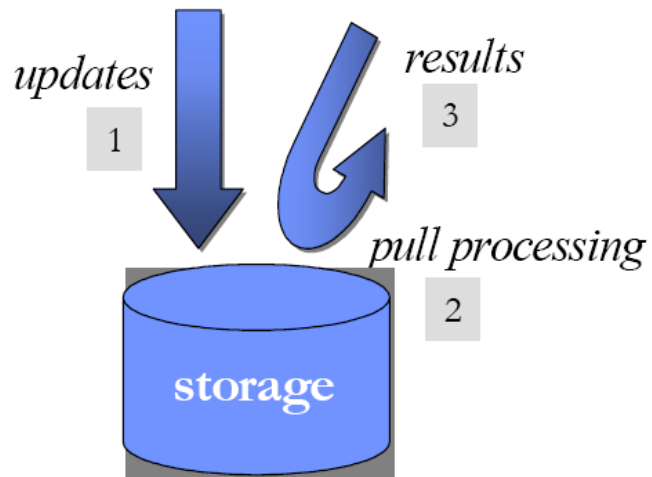
- Fight in industry MOLAP versus ROLAP
- Some companies offer MOLAP based tools: Essbase, MS Analysis Services
- companies do not tell people how they actually do it...
- reports on problems with many dimensions, high cardinalities and unexpectedly long indexing times
- therefore some companies favor HOLAP: Hybrid OLAP, i.e., part of the data is in ROLAP another part in MOLAP
- Recommendation
  - do not use it
  - given the current hardware characteristics it is questionable whether MOLAP has a business case if the techniques taught in this lecture are well applied

# Big picture: One size **does not fit**

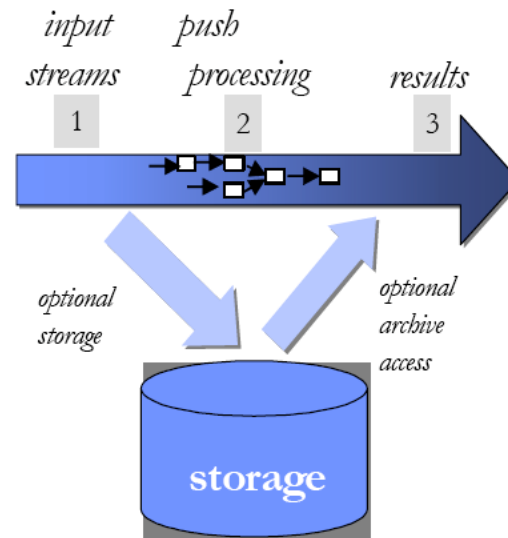
- Last 25 years of DBMS can be summed up in a single phrase:  
“One size fits all“
- DBMS vendors started with an OLTP centric relational architecture and extended that over the years to be able to cope with other applications.
- The idea was **one** architecture for **every** data processing application.
- This idea does not work. Vendors use separate engines.
- Examples include
  - data warehousing
  - stream processing
  - scientific databases
  - text search
  - personal information management
- Literature: Michael Stonebraker, Ugur Çetintemel: "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). ICDE 2005: 2-11

# Big picture: One size **does not fit**

- Example: stream processing

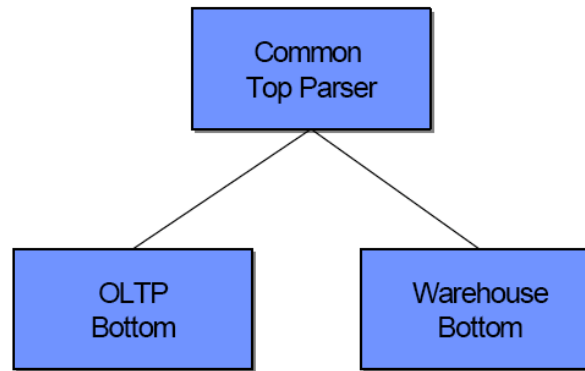


Outbound processing



Inbound processing

# Big picture: One size **does not fit**



- Current approach of vendors: hide multiple engines behind a common frontend
- Cons
  - illusion of a single system is a marketing fiction
  - vendor is in fact selling multiple systems
  - pressure from both markets to include market specific features
  - common front-end strategy impractical, e.g., for stream processing systems

# Big picture: One size **does not fit**

- Current Prediction on Engines
  - OLTP
    - pull-based operator model
    - row store
    - read/write
    - transactional
    - structured data
  - OLAP
    - pull-based operator model
    - column store
    - read mostly
    - structured data
    - long history on different engines

# Big picture: One size **does not fit**

- Current Prediction on Engines
  - IR
    - pull-based operator model
    - inverted lists (stored as files or in main memory or compressed or ...)
    - read only
    - unstructured data
  - Streams
    - push-based operator graphs
    - write mostly!
    - structured data

# Big picture: One size **does not fit**

- Summary
  - hard to find one code line that works well for all applications
  - current OLTP engines bloated with add-on functionality that does not always perform well
  - in future they may be domain specific database engines (it is already happening)
- Some (personal) criticism on this
  - At CIDR 2007 there was some evidence that some of the engines may be merged in future
  - DB/IR is a good example
    - IR and DB techniques are converging
    - part of this is explored in the iMeMex project (see [imemex.org](http://imemex.org))
    - iMeMex: merge of DB, IR, **and** Information Integration

# Summary

- Several methods to speed-up OLAP queries were presented
  - star join plans
  - bitmaps
  - join-indexes
  - mat views
  - partitioning
  - vertical and horizontal partitioning
  - compression
  - cache awareness and obliviousness
  - page size
  - data layout
  - multidimensional databases
  - „one size fits all“ approach etc.
- Applying all these techniques well gives excellent OLAP query performance even on TB data warehouses.
- Not all vendors provide all features.

# Literature

- Don S. Batory: On Searching Transposed Files. ACM Trans. Database Syst. 1979.
- George P. Copeland, Setrag Khoshafian: A Decomposition Storage Model. SIGMOD Conference 1985: 268-279
- Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, Marios Skounakis: Weaving Relations for Cache Performance. VLDB 2001.
- Patrick E. O'Neil, Goetz Graefe: Multi-Table Joins Through Bitmapped Join Indices. SIGMOD Record 24(3): 8-11 (1995)
- Patrick E. O'Neil, Dallon Quass: Improved Query Performance with Variant Indexes. SIGMOD Conference 1997: 38-4
- Chee Yong Chan, Yannis E. Ioannidis: Bitmap Index Design and Evaluation. SIGMOD Conference 1998: 355-366
- Clark D. French: "One Size Fits All" Database Architectures Do Not Work for DDS. SIGMOD Conference 1995: 449-450
- Clark D. French: Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. ICDE 1997: 194-198
- Michael Stonebraker, Ugur Çetintemel: "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). ICDE 2005: 2-11

# Literature

- Ian H. Witten, Alistair Moffat, Timothy C. Bell: Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. 1999, Morgan Kaufmann
- Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, Yannis Kotidis: Dwarf: shrinking the PetaCube. SIGMOD Conference 2002: 464-475