Part VI

Storage Structures for Data Warehouses

Storage Structures for Data Warehouses

Relational storage – ROLAP

- Implementation of star or snowflake schema to relations
- Common form of storing DW tables
  (Details: see Lecture "database implementation techniques")
- Features
  - Very large fact tables! → Acceleration of access by partitioning
  - Multidimensional Access → specific cluster and index structures
  - Update characteristic (appending data)

Partitioning

- Independent of and in addition to indexing methods:
  Separating large relations into smaller subrelations (so-called partitions or fragments)
- Size and content of the partitions depends on request and update characteristics
- Originally intended for distributed databases for supporting load distribution on multiple nodes
- Partitioning includes the **logical** structure of relations, the physical distribution is the responsibility of the allocation

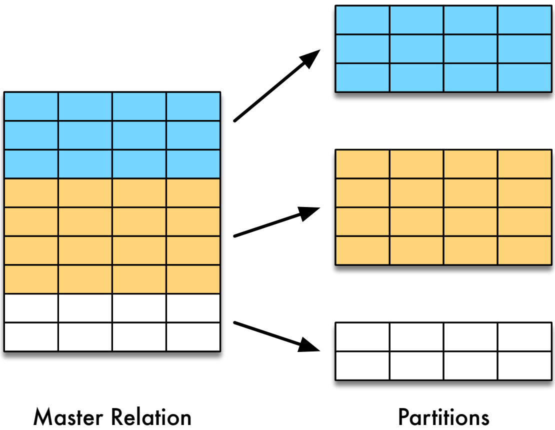
Horizontal Partitioning

- Master relation R is split in multiple pairwise disjoing subrelations $R_1, \ldots, R_n$:

$$R = R_1 \cup \ldots \cup R_n; \quad R_i \cap R_j = \emptyset \text{ fÃ}\frac{1}{4}\text{r } i \neq j$$

- Various forms of splitting:
  - Range partitioning:
    - Each partition is defined by a selection criterion $R_i := \sigma_\varphi(R)$ with φ selection condition (range restriction)
  - Hash partitioning:
    - Hash (applied to the whole or individual tuple attributes) determines to which partition a tuple belongs
    - Tuples with the same hash value (or hash values in a given area) are located in a partition

Horizontal Partitioning (2)



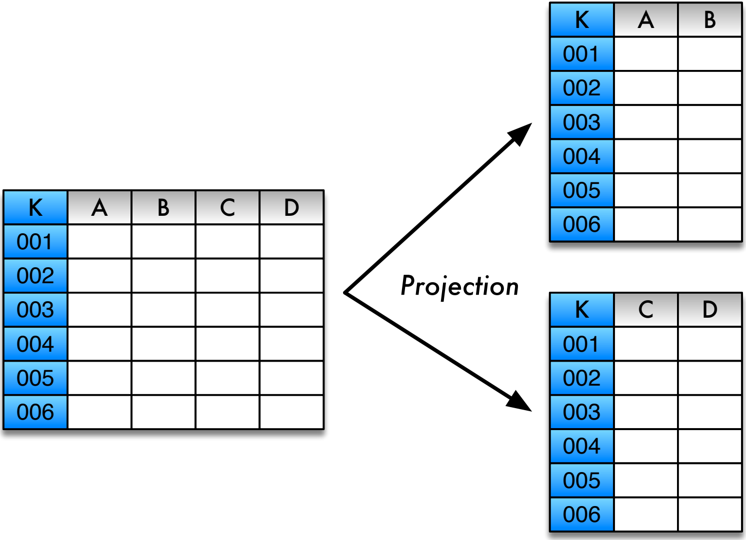
Vertical Partitioning

- Distribution of individual attributes (columns) on partitions
- Partition corresponds to a projection on the master relationship:

$$R_i := \pi_{attrlist}(R)$$

- For reconstructing the master relation, there must be a common attribute in two partitions
  - Normally, the primary key is included in all partitions enthalten

Vertical Partitioning (2)



Partitioning in Oracle

- Range Partitioning

```
CREATE TABLE Sales (
   Date DATE NOT NULL,
   ...)
PARTITION BY RANGE(Date)  (
PARTITION Sales2009
   VALUES LESS THAN (to_date('2010-01-01')),
PARTITION Sales2010
   VALUES LESS THAN (to_date('2011-01-01')),
PARTITION Sales2011
   VALUES LESS THAN (to_date('2012-01-01')));
```

Partitioning in Oracle (2)

- Hash Partitioning

```
CREATE TABLE Sales (
   ProductID INT NOT NULL,
   BranchID INT NOT NULL,
   ...)
PARTITION BY HASH(ProductID, BranchID)
PARTITIONS 5;
```

Partitioning in Data Warehouses

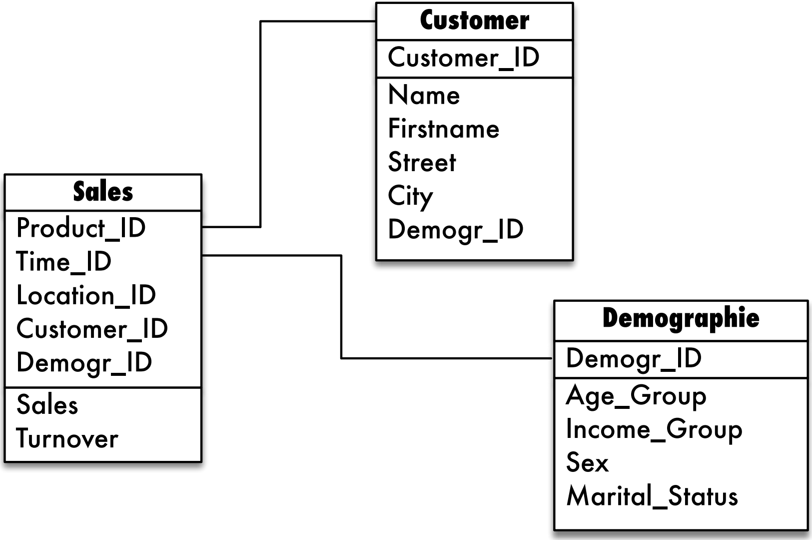
- Horizontal Partitioning (esp. Range Partitioning) allows for example to split large fact tables in more manageable parts
  - Selection criteria for individual partitions should consider the frequently occuring range restrictions in queries
- Vertikal partitioning requires normally expensive join operations to reassemble tuples;
  - Can be used for splitting off rarely queried attributes
  - Reduction of fact or dimension tables that are frequently accessed

Partitioning in Data Warehouses (2)

- Special case of vertical partitioning: mini dimensions
- Occasionally dimension tables become huge in size: e.g. customer table with several million records
  - Many attributes are never or only rarely requested, because they are are uninteresting for evaluations

  Or
  - There are disjoint attribute groups which are only ever used for different applications or different types of evaluations. are needed
- Separation of attributes by vertical partitioning allows then a clear reduction of the individual dimension tables

Mini Dimensions



Special Table Types in DB2

- Append mode tables
  - Optimized for **`insert`** operations.
  - Tuples are appended at the end, without free space on pages on pages
- Range-clustered tables (RCT)
  - For sequence data
- Multidimensional clustered tables (multidimensional clustering tables – MDC)
  - Storage in multiple dimensions cluster-wise

Append-Mode Tables

- Optimized mode for tables to add data.
- Adding is done at the end $\rightarrow$ INSERT optimization
- Leads to multiple page loads at query time
- In DB2 via ALTER TABLE no clustered index may be associated
- In Oracle on load, e.g. bulk loader option

```
ALTER TABLE Order (
   OrderNo int primary key, ...
) APPEND ON
```

Range Clustered Tables

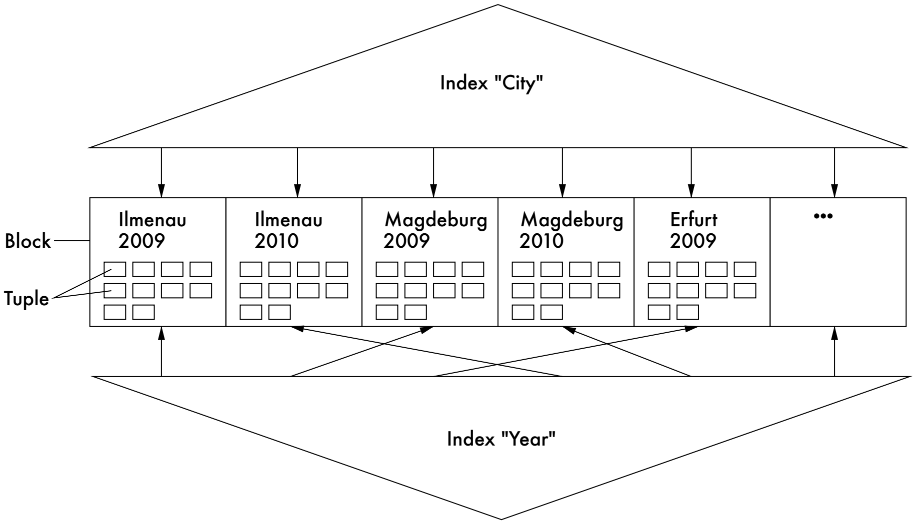
- Use a sequence number (arbitrary attribute) as a logical Rowid to determine the physical memory address
- Pre-allocation of the whole memory space of the table
- Sorting of the tuple via sequence number
- Access via sequence number → no additional index necessary

```
CREATE TABLE Purchase_order (
   OrderNo int primary key, ...
) ORGANIZE BY KEY SEQUENCE
   (OrderNo starting from 1 ending at 10000)
```

MDC Tables

- Tables usually clustered max. by one index
- Scan over other index in worst case: 1 page access per tuple
- MDC:
  - Tuples with same values concerning several attributes (dimensions) store on the same page or in the same extent
  - Indexing via block indexes (sparse indexes)

MDC Tables and Block Indexes



Creation of an MDC Table

```
CREATE TABLE Sales (
    Turnover number,
    Year int,
    City varchar(20),
    ...
) ORGANIZE BY DIMENSIONS (City, Year)
```

Multidimensional Storage

Multidimensional Storage – MOLAP

- Use different data structures for data cube and dimension
- Storage of the cube as array
- Ordering of the dimension for addressing the cube cells necessary
- Often proprietary structures (and systems)

Data Structures for Dimensions

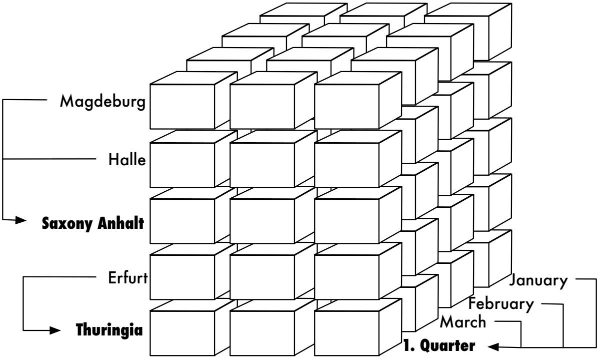
- Finite, ordered list of dimension values.
- Dimension values: simple unstructured data types (String, integer, date)
- Order of dimension values (internal integer 2 or 4 bytes)
  $\rightarrow$ Finiteness of the value list

Data Structure for Cube

- For n dimensions: n-dimensional space.
- m_i dimension values of dimension i: Division of the cube into m parallel planes
- By finiteness of the list of dimension values: finite, list of planes of the same size per dimension
- Cell of an n-dimensional cube is uniquely defined over n-tuples of dimension values.
- cell can hold one or more metrics of a previously defined data type defined before
- In case of multiple key figures: Alternative $\rightarrow$ multiple data cubes

Classification Hierarchies

- Dimension values include all expressions of the dimension: Elements (leaves) and nodes of higher classification levels.
- Nodes of higher levels form further levels



Calculation of Aggregations

- Real time:
  - On request of cells representing values of a higher aggregated classification level $\rightarrow$ Calculation from Detailed data
  - High timeliness, but high overhead
  - Possibly caching
- Pre-calculation:
  - After taking over the detail data $\rightarrow$ calculation and entering the aggregation values into corresponding cells
  - Recalculation necessary after each data transfer
  - High query speed, but increase of cube size and runtime overhead
- Solution: incremental precalculation

Other Data Structures

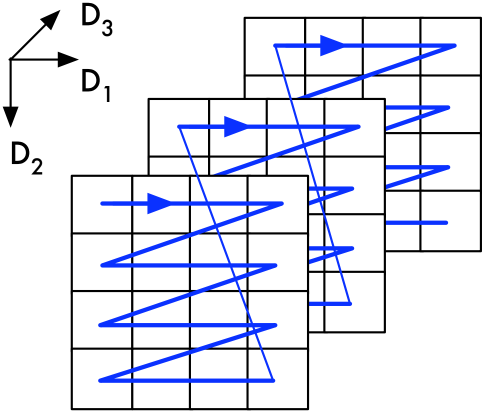
- Attributes
  - Classifying features of a dimension.
  - Identification of subsets of dimension values. (e.g. ”product color”)
  - Not intended for precalculation
- Virtual cube
  - Includes derived data ("‘profit"’, "‘percentage turnover")
  - Derived from other cubes by applying calculation functions $\approx$ Views in relational model
- Partial cube
  - Combination of multiple planes of a cube $\rightarrow$ virtual

Array Storage

- Storing the cube as an n-dimensionales Array
  $\rightarrow$ Linearizing to a one-dimensional list
- Indexes if the arrays
  $\rightarrow$ Coordinates of the cube cells (Dimensions D_i)
- Index calculation for cell z with coordinates $x_1 ... x_n$

$$Index(z) = x_1 + (x_2 - 1)|D_1| + (x_3 - 1)|D_1||D_2| + \cdots + (x_n - 1)|D_1| \cdots |D_{n-1}|$$

Linearization Order



Array Storage: Problems

- Number of disc accesses in case of unadvantageous linearization orders
  - Order of the dimensions needs to be considered while defining the cube
- Caching required for reduction
- Storage of sparse cubes

Storage Consumption

	Array	**Relational** (Star-Schema)
Storage Coordinates	Implicit (Linearization)	Explicit (redundant)
Empty Cells	Take space	Take no space
New classif. nodes	complete reorganization *Strong growth in storage consumption*	new row in dimension table *Almost no growth in storage consumption*
Storage consumption	$b \cdot \prod_{i=1}^{n} d_i$	$b \cdot M \cdot (n+1)$

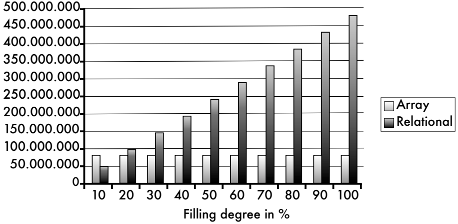
M: Number of facts, i.e., $M = \delta \cdot \prod_{i=1}^{n} d_i$ (fill degree δ)

Comparison Storage Consumption

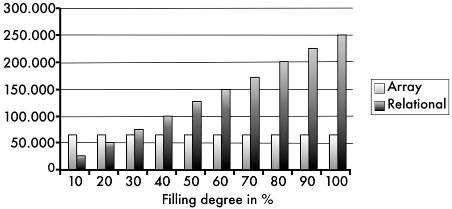
- Factors
  - Fill degree
  - k: Number of nodes
  - n: Number of dimensions
- Already at small fill degrees array storage is more efficient
- Performance depends on many factors
  - Indexing
  - Sequential reads
  - ...

Storage Consumption Comparison (2)

Storage Consumption by Filling Degree, b=8, k=100, n=5



Storage Consumption by Filling Degree, b=8, k=20, n=3



Limits of Multidimensional Storage

- Scalability issues due to sparse data spaces.
- Partial one-sided optimization with respect to read operations
- Ordering of dimension values necessary (due to array storage)
  - Complicates changes to dimensions
- No standard for multidimensional DBMSs
- Special knowledge required

Hybrid Storage – HOLAP

- Combining the advantages of both worlds
  - Relational (scalability, standard)
  - Multidimensional (analytical power, direct OLAP support)
- Storage:
  - Relational database: detailed data
  - Multidimensional database: aggregated data
  - Multidimensional storage structures as an "“intelligent”" Cache for frequently requested data cubes
- Transparent access via multidimensional query system

Storage Variants

Storage Variants

- Goal:
  - Optimization for read operations, spec. OLAP queries (aggregations)
  - Fast loading of actually needed data into main memory for calculation
- Aspects:
  - Partitioning: remove empty areas
  - Compression: avoid storing null values and redundant data redundant data
  - indexing *(next chapter)*:
    - ★ Of data blocks (grid files, R+ trees, two-levels).
    - ★ In a data block (array/relational storage of the cells, RLE, bitmap)
- Overall: preserving the spatial neighborhood relationship of the cells in secondary storage (multidimensional clustering).

Partitioning of Data Cubes

- Goal:
  - Removal of empty areas from cube
  - Optimized storage for access patterns: frequently accessed areas in a few blocks
- Criteria:
  - Type of partitioning
  - Control: optimization of partitioning for application
  - Tool support: for control of partitioning

Partition Type

- Divisionpartition of a cube into non-overlapping areas. (multidimensional intervals)
- General form: multidimensional tiling
  - Given: n-dimensional array with dimension values

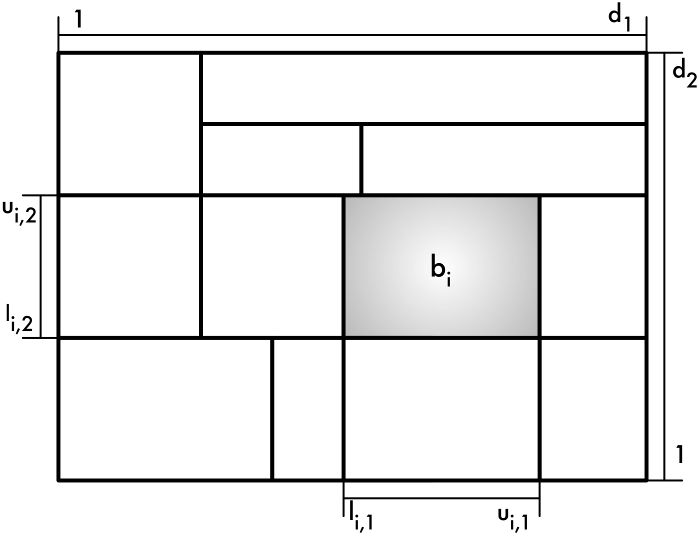
$$D = [1 : d_1, ..., 1 : d_n]$$

  - Tiling: set of sub-arrays corresponding to ranges $b_1, ..., b_m$ of dimension values

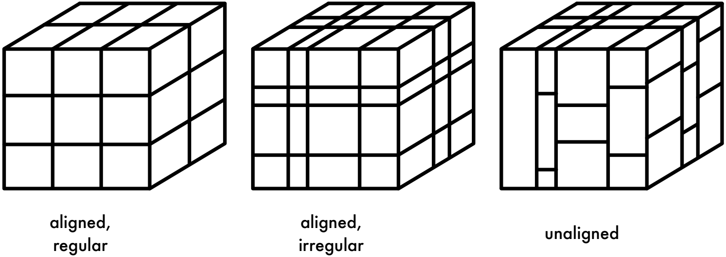
$$b_1 = [l_{1,1} : u_{1,1}, ..., l_{1,n} : u_{1,n}], ..., b_m = [l_{m,1} : u_{m,1}, ..., l_{m,n} : u_{m,n}],$$

    such that $b_i \cap b_j = \emptyset$ for $i \neq j$ and $b_i \subseteq D,\ i, j = 1, ..., m$ and each occupied cell belongs to a sub-array

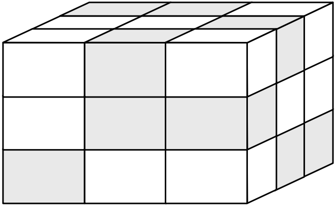
Multidimensional Tiling



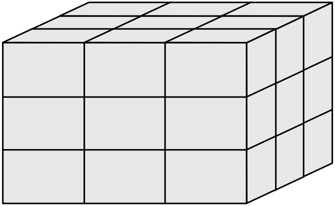
Tiling Orientation



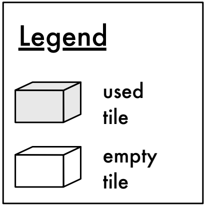
Space Occupation of Tiling



sparsely populated



densely populated



Control of Partitioning

- Automatic partitioning:
  - Automatically find the partitioning for optimal execution of operations
  - Use of the fill level of the areas
  - Use of access statistics
- Importance of certain dimensions/dimension combinations
  - Special treatment of the time dimension
  - Partitioning by time series (special formats for series of values, e.g. daily, weekly, etc.)
- Two-level storage
  - Only storage of used combinations of sparse Dimensions
- Partition specification of the user
  - Direct specification of each range
  - dimension partitions

Cell Storage

- Use a specific storage format for each data block
- Support of different storage formats (depending on fill level)
- From certain fill level: array storage more efficient than relational storage
  - Reason: storage of coordinates as key necessary with relational storage necessary

Minimum Fill Level for Optimal Storage

- Above a computable minimum fill level is array storage is better than relational storage
- Minimum fill level δ is maximum δ such that holds:

$$Ix_{rel} + \delta \prod_{i=1}^{n} L_i \cdot \left(s_c + \sum_{j=1}^{n} s_j \right) < Ix_{arr} + \prod_{i=1}^{n} L_i \cdot s_c$$

- L_i: Length of the sub-array in dimension i
- s_c: Memory size of the cells (space consumption of all parameters of a cell)
- s_j: Memory size of dimension attributes j
- Ix_{rel}: Storage size of the indexing (relational storage)
- Ix_{arr}: Memory size of indexing (array storage).

Minimum Fill Level: Example

- Assumption: Ix_{rel} and Ix_{arr} equal, $s_j = s_c = 8$
- 2 dimensions:

$$\delta \prod_{j=1}^{2} L_i \cdot 24 < \prod_{j=1}^{2} L_i \cdot 8$$

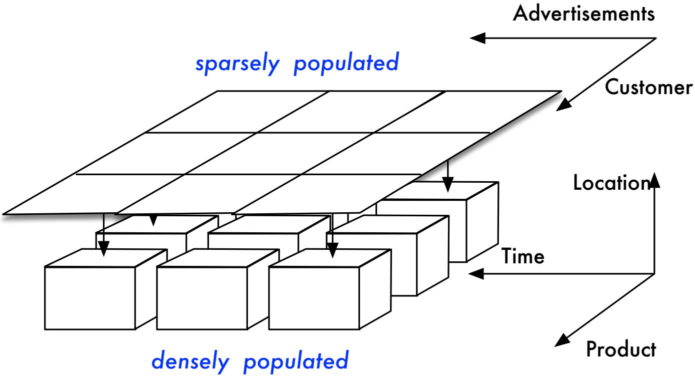
- Array storage more efficient from fill level 0.33
- For three dimensions: 0.25

$\Rightarrow$ Filling degree decreases with increasing number of dimensions

Two-Level Data Structure

- Upper level indexes data blocks that are stored on lower level stored on lower level
- Lower level array containing all possible combinations of dimension values
- Cells of the array:
  - Pointer to data block containing data values for corresponding dimension value of the densely populated dimensions
  - **NULL** for empty range

Two-Level Data Structure



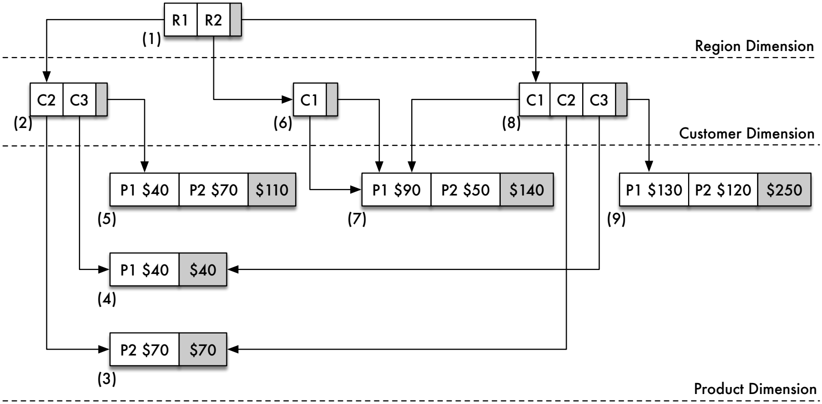
Dwarf – Shrink petacube

- Highly compressed data structure.
- Prefix and suffix redundancies are unified
  - Prefix suitable for dense areas
  - Suffix suitable for sparse areas
- 1 petabyte cube with 25 dimensions $\rightarrow$ 2.3 GB dwarf
- Suitable for distribution in mobile networks

Example

region	**customer**	**product**	**price**
R1	C2	P2	70
R1	C3	P1	40
R2	C1	P1	90
R2	C1	P2	50

Dwarf: Example



Further Multidimensional Storage Structures

- Cube Forests und Hierarchically Cube Forests [Johnson & Shasha 1996, 1997]
- CubeTree [Roussopoulos & Kotidis & Roussopoulos 1997]
- CubiST [Fu & Hammer 2000]
- Condensed Cube [Wang & Lu & Feng & Yu 2002]
- Quotient Cube [Lakshmanan & Pei & Han 2002]
- m-Dwarf [Michalarias & Omelchenko & Lenz 2009]

Further Multidimensional Storage Structures (2)

Based on Iceberg Cubes

- Bottom-Up Cube [Beyer & Ramakrishnan 1999]
- H-Cubing [Han & Pei & Dong & Wang 2001]
- Star Cubing [Xin & Han & Li & Wah 2003]

Summary

- Relational vs. multidimensional storage
- Relational extensions
  - Partitioning
  - Special table types
- Special types of multidimensional storage
  - Array storage
  - Tiling
  - Handling of thinly populated cubes
- Hybrid shapes

Column-Oriented Storage

Motivation for Column-Oriented Data Management

- requests serve analysis of data (long transactions).
- dataset stable, i.e. few/no updates
- import of data (often) via ETL process
- Single values often uninteresting (cf. application fields DWH)
- Frequently create and process aggregated values
  - AVG(), SUM(), COUNT()
  - GROUP BY (CUBE)
  - CUBE operations
- For aggregate functions (e.g. AVG()) single columns are interesting.
- Also groupings (and CUBE) intuitive column-by-column processing

Data Explosion

- Data volumes in OLAP are constantly growing $\hookrightarrow$ Data hardly manageable in centralized systems
- Historization of data increases data volume additionally
- OLAP queries are very memory and compute intensive $\hookrightarrow$ Load distribution necessary
- For aggregations (OLAP) vertical partitioning/fragmentation is useful $\hookrightarrow$ exploit already existing partitioning of column stores
- Current systems use compression techniques for data volume reduction

OLAP: Row Store

- Historically: used in On-line Transactional Processing (OLTP) with short transactions, e.g. posting transactions.
- Mapping of tuples in DBMS, i.e. tuples stored sequentially.
- processing of whole tuples for aggregate functions $\hookrightarrow$ I/O overhead.
- **total:** Tuple-oriented physical storage unfavorable for OLAP.

Product	City	Turnover	Year
Merlot	Magdeburg	4325	2010
Guinness	Magdeburg	2341	2010
Merlot	Ilmenau	5543	2010
Pinot Noir	Ilmenau	4944	2010

OLAP: Column Store

- New architecture needed for OLAP
  $\hookrightarrow$ Column Stores
- Tuples partitioned by columns, i.e. values of a column stored sequentially (and sorted)
- aggregate functions work directly on columns $\hookrightarrow$ only needed data read in
- **total:** Column stores can handle aggregations much more effectively

Product
Merlot
Guinness
Merlot
Pinot Noir

Location
Magdeburg
Magdeburg
Ilmenau
Ilmenau

Turnover
4325
2341
5543
4944

Year
2019
2019
2019
2019

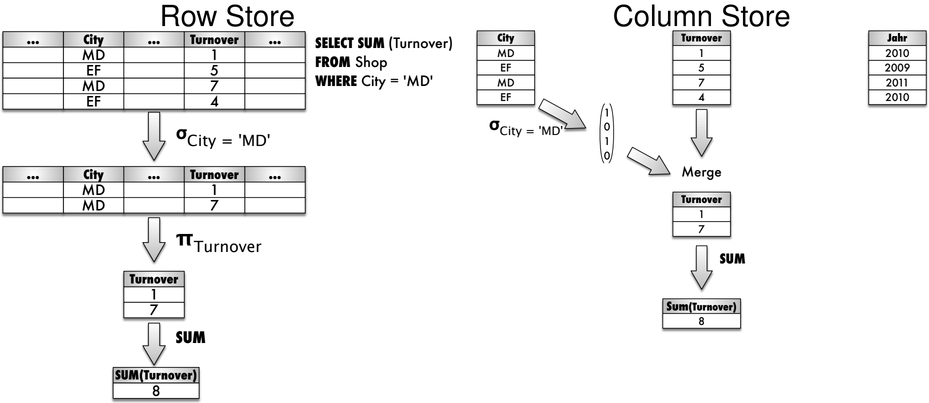
Compression

- row stores
  - Compression relation- or partition-wise
  - Different data types $\hookrightarrow$ Trade off necessary for compression technique selection
  - Common compression ratios 2:1 to 5:1
- Column Stores
  - Compression per column possible
  - use of different techniques, e.g. Run Length Encoding (RLE), dictionary encoding (Lempel-Ziv)
  - Selection of best compression technique per column, i.e. for each data type
  - Common compression ratios 10:1 to 40:1
- Column stores reduce I/O overhead **and** reduce data volume sometimes significantly $\hookrightarrow$ better utilization of main memory

Query Processing

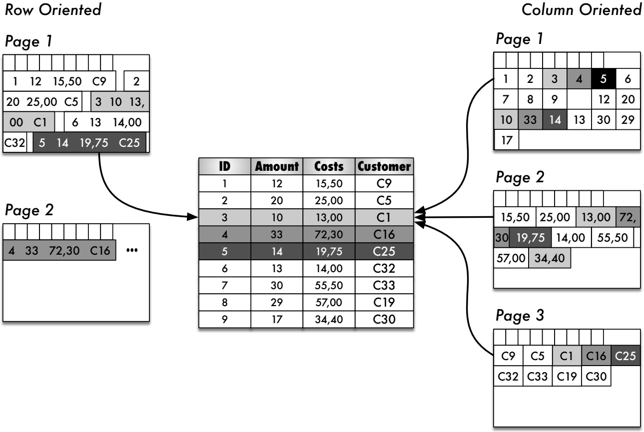
- Column stores are also based on relational data model $\hookrightarrow$ Using relational algebra and its operations
- Logical query plan as for row stores.
- architecture specific execution transparent
- Bit operations inherently supported (cf. bitmap join index).
- Column-wise compression allows processing without decompression
  - Lossless compression techniques (best known: Lempel-Ziv and derivatives) $\hookrightarrow$ same (uncompressed) values have same compressed representation
  - I.e. comparison value is converted to compressed representation if necessary before query execution $\hookrightarrow$ well suited for non-vector based joins
  - Order preserving techniques like RLE $\hookrightarrow$ values directly comparable if necessary or like lossless compression
  - I.e. aggregate functions like MIN/MAX or SUM can process compressed data

Anfrageplanausführung Column vs. Row Store



Tuple Reconstruction

- Operator is called **SPC** (Scan, Predicate, Construct) vor full tuple reconstruction
- **Merge** is a k-tuple reconstruction (with columns $VAL_1 \ldots VAL_k$)

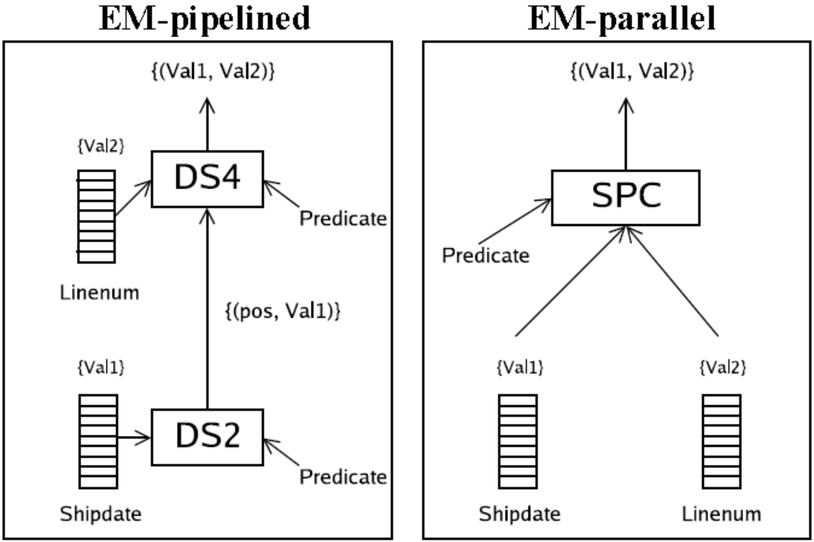


ID	Amount	Costs	Customer
1	12	15,50	C9
2	20	25,00	C5
3	10	13,00	C1
4	33	72,30	C16
5	14	19,75	C25
6	13	14,00	C32
7	30	55,50	C33
8	29	57,00	C19
9	17	34,40	C30

Materialization Time Point

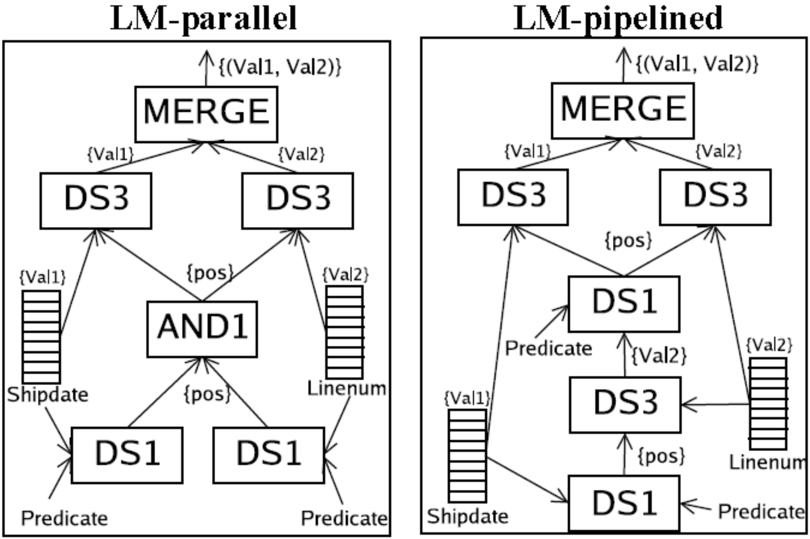
- Early Materialization (EM)
  - Request processing very close to row stores
  - Aggregate functions on single columns
  - Tuple reconstruction as soon as tuple used
  - Mostly used for tuple oriented query processing
- Late materialization (LM)
  - Work on columns as long as possible
  - Multiple access to base tables and/or intermediate results
  - Consequence: Query plan no longer a tree
  - But: Simultaneous processing on compressed and uncompressed data possible
  - Necessary for (effective) column-oriented query processing

Early Materialization (EM)



Taken from "Query execution in column-oriented database systems", PhD thesis by D. Abadi

Late Materialization (LM)



Taken from "Query execution in column-oriented database systems", PhD thesis by D. Abadi

Disadvantages

- Tuple reconstruction incurs costs
- costs for insert operation due to tuple partitioning
- Updates need tuple reconstruction
- Consequence: insert- and update-in-place not possible
- But: Updates and inserts in OLAP/DWH application rarely or only by ETL

Solutions

- Tuple-oriented query processing and early materialization.
- C-Store/Vertica
  - Read-optimized (RS) and write-optimized storage (WS).
  - Different overlapping projections in RS
  - Inserts and updates only in WS
  - Tuple mover transfers data from WS to RS at low load (offline)
- SybaseIQ (first commercial column store)
  - Similar to C-Store approach
  - System divided into read and write or read/write nodes
  - Adjustment in the background at times of low load
- redundancy
  - data column- and row-oriented in main memory
  - Database redundant as column and row store
  - virtualization of the data cube
- ...

Systems

- Commercial
  - SybaseIQ
  - Vertica
  - Infobright ICE
  - Tenbase (web-based)
  - BigTable (Google, not relational)
  - ...
- Free
  - Infobright ICE Community Edition
  - LucidDB
  - MonetDBX100 (Ingres/Vectorwise)
  - C-Store (requires old gcc)
  - Hbase (Apache), Hypertable, Cassandra (Facebook) all BigTable derivatives
  - ...
- In contrast to row stores, column store implementations differ greatly among themselves

Summary

- Row Stores not optimal for OLAP and DWH applications.
- Column Stores better suited for aggregate functions
- Column Stores reduce data volume partly drastically $\hookrightarrow$ less I/O, better utilization of main memory
- Representation of tuples generates costs in column stores (tuple reconstruction)
- Column stores show weaknesses in inserts and updates
- Many and very different implementations for column stores