Data Warehousing

Implementation and Performance Aspects

Spring Term 2020

Dr. Andreas Geppert
geppert@acm.org

Outline of the Course

- Introduction
- DWH Architecture
- DWH-Design and multi-dimensional data models
- Extract, Transform, Load (ETL)
- Metadata
- Data Quality
- Analytic Applications and Business Intelligence
- **Implementation and Performance**
- Privacy and Security

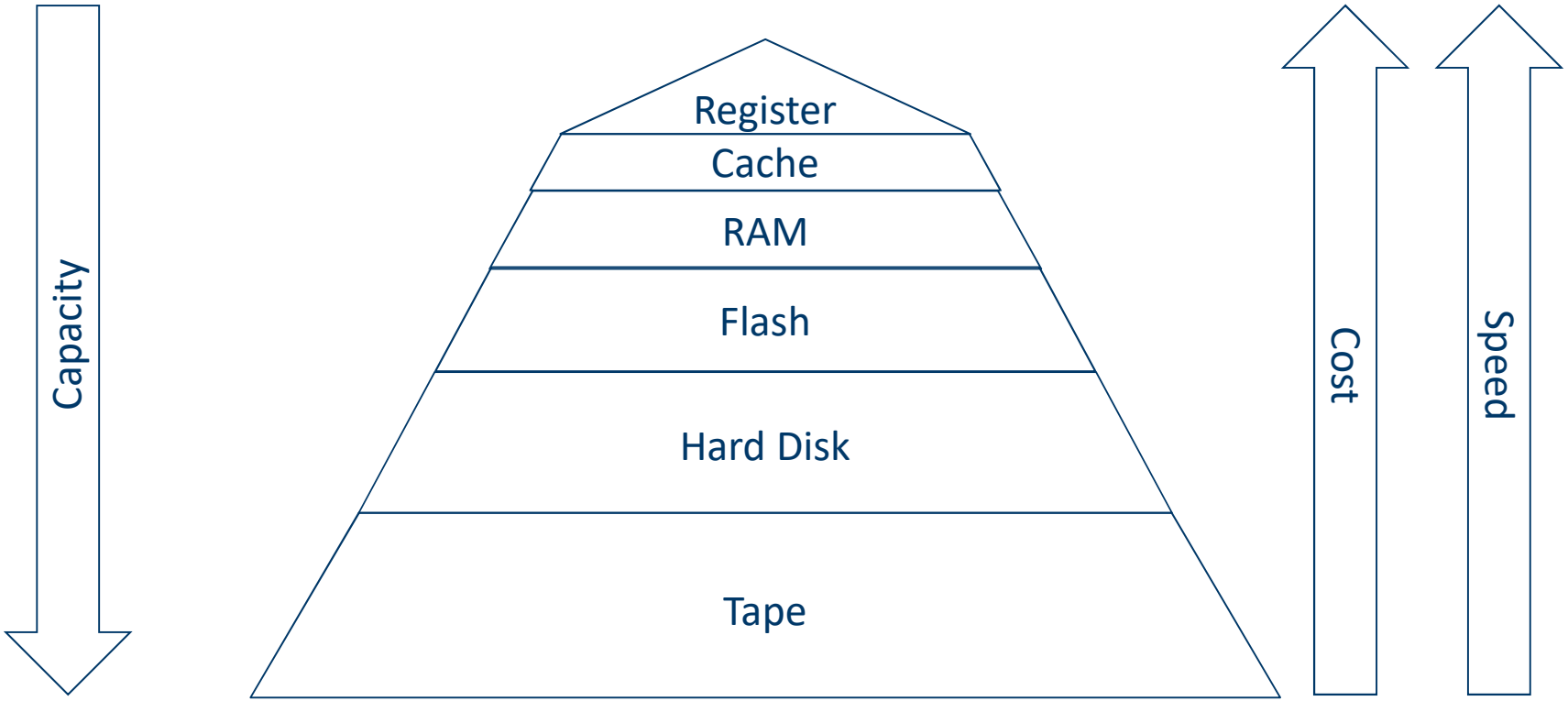
Outline

1. Introduction
2. Indexing
3. Star Joins and Join Indexes
4. Partitioning
5. Column Stores
6. Main-memory Databases
7. Materialized Views

Introduction

- Relational Implementation (e.g. Star Schema)
- Analytic queries
  - Range queries
    - "... for all products in product family food", "... For the last three years"
  - Multi-dimensionality: symmetric access to all dimensions
  - Short response times
  - Analytic queries against very large tables
- Queries are very complex
  - Many selection criteria
  - Multi-table joins
- Required:
  - adequate query execution techniques
  - Tuning measures
- See "Implementation of database systems"

Storage Hierarchy

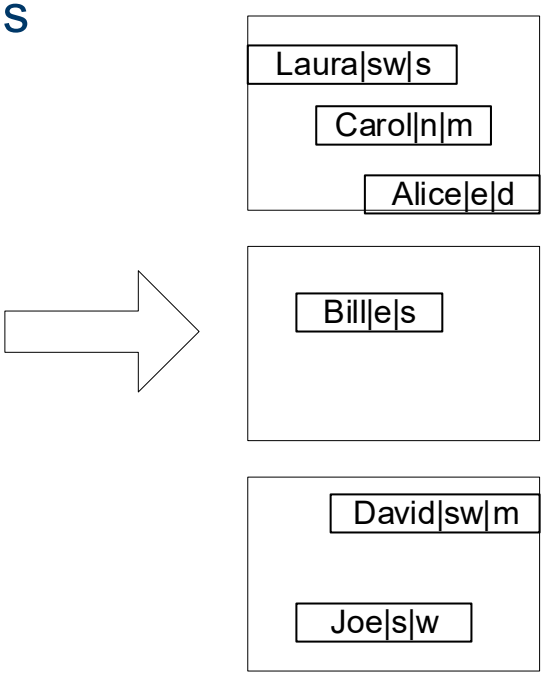


Physical Storage in Relational DBMS

- Rows in tables are stored as records in pages/blocks
- In order to read/write data, the corresponding pages must be brought into main memory (the buffer)
- Number of page read/writes (I/O) is essential

Relation (logical)

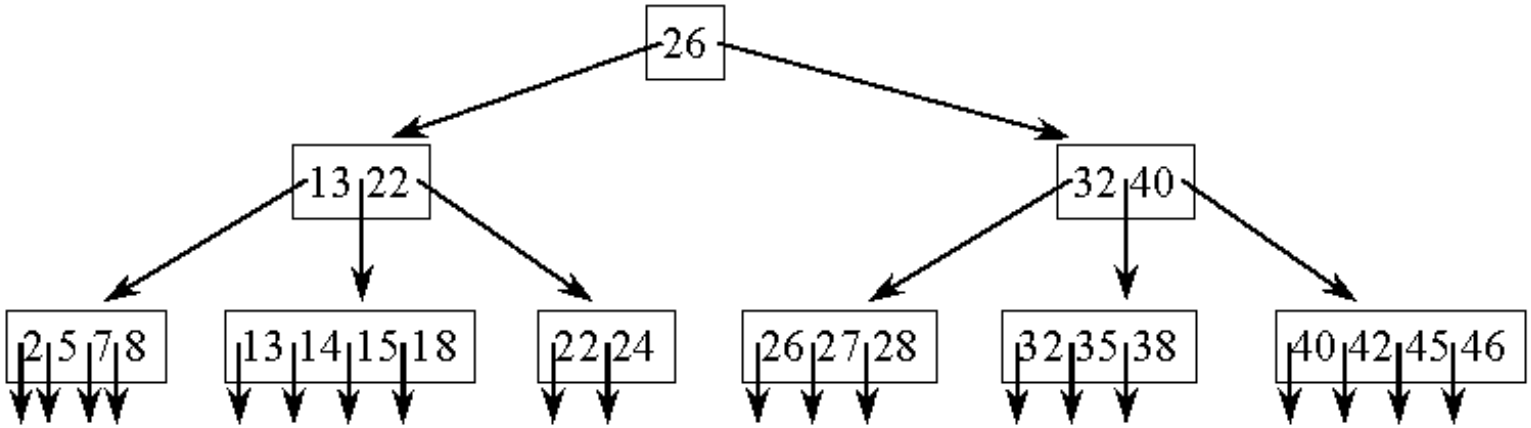
Name	Region	Marital_status
Bill	e	s
Carol	n	m
Alice	e	d
Joe	s	w
Laura	sw	s
David	sw	m



Seiten (physical)

Tuning: Indexing

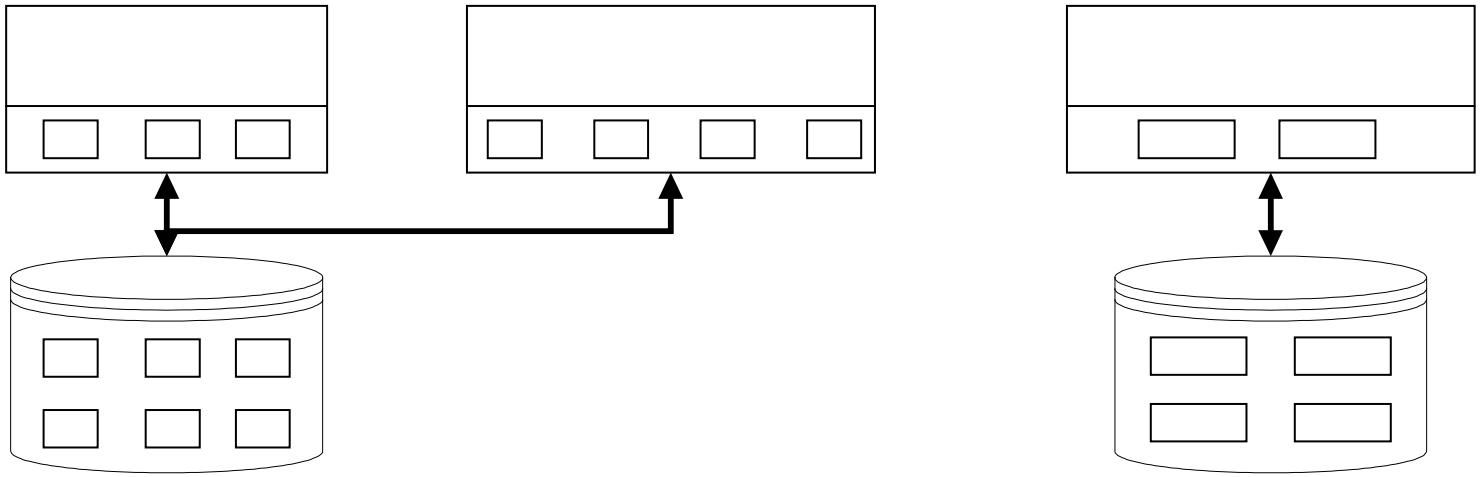
- Indexes
  - An index inverts the association between a key or a location and an attribute value
  - Direct access vs. search in entire table (table scan)
- Typically indexes are structured as trees



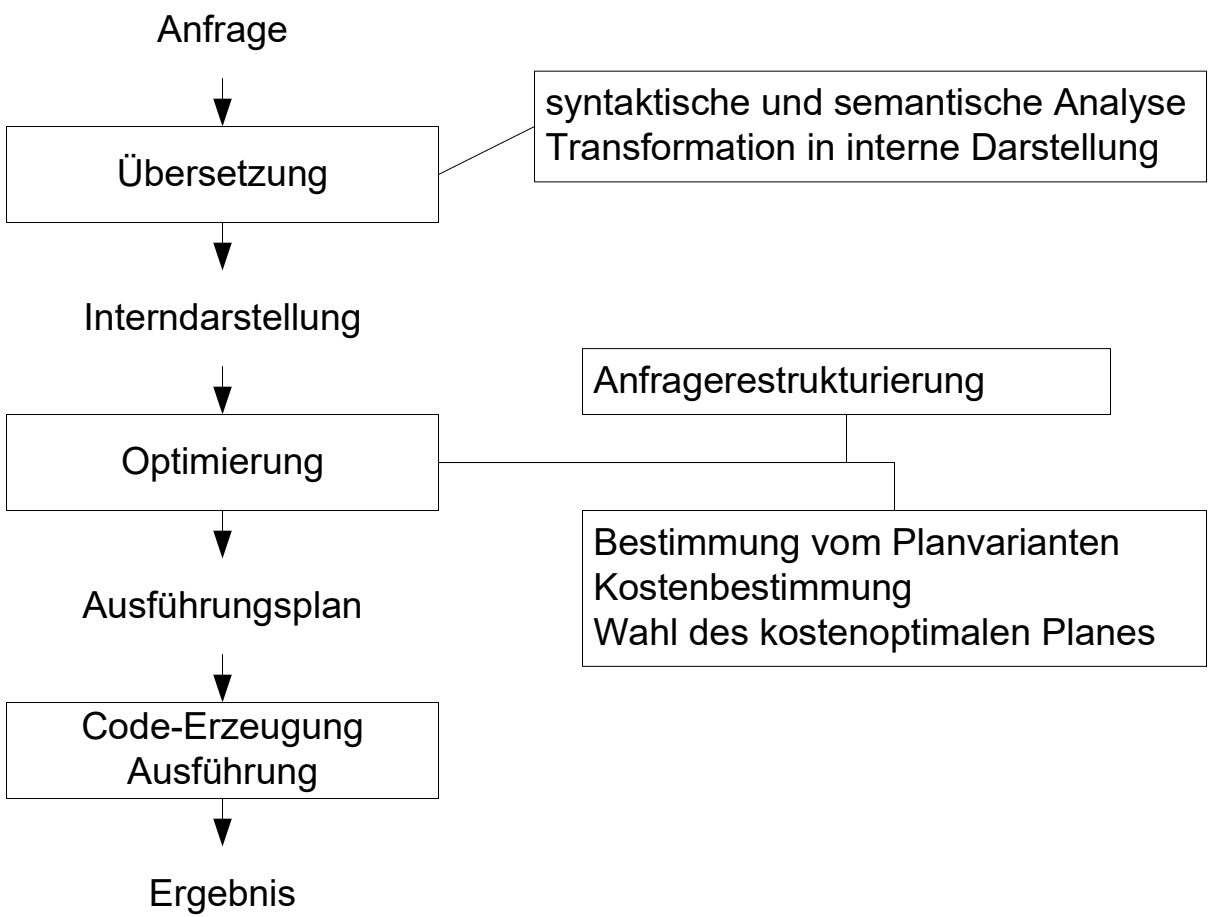
- Source: http://db.ucsd.edu/CSE232W99/Indexing/sld020.htm

Tuning: Buffer and Page Sizes

- Database buffer
  - Contains copies of storage blocks
  - Larger buffer ⇒ more pages fit into buffer
  - Larger pages
    - One can read/write more records per page IO
    - But fewer pages fit into the buffer (thus probably more page IO)



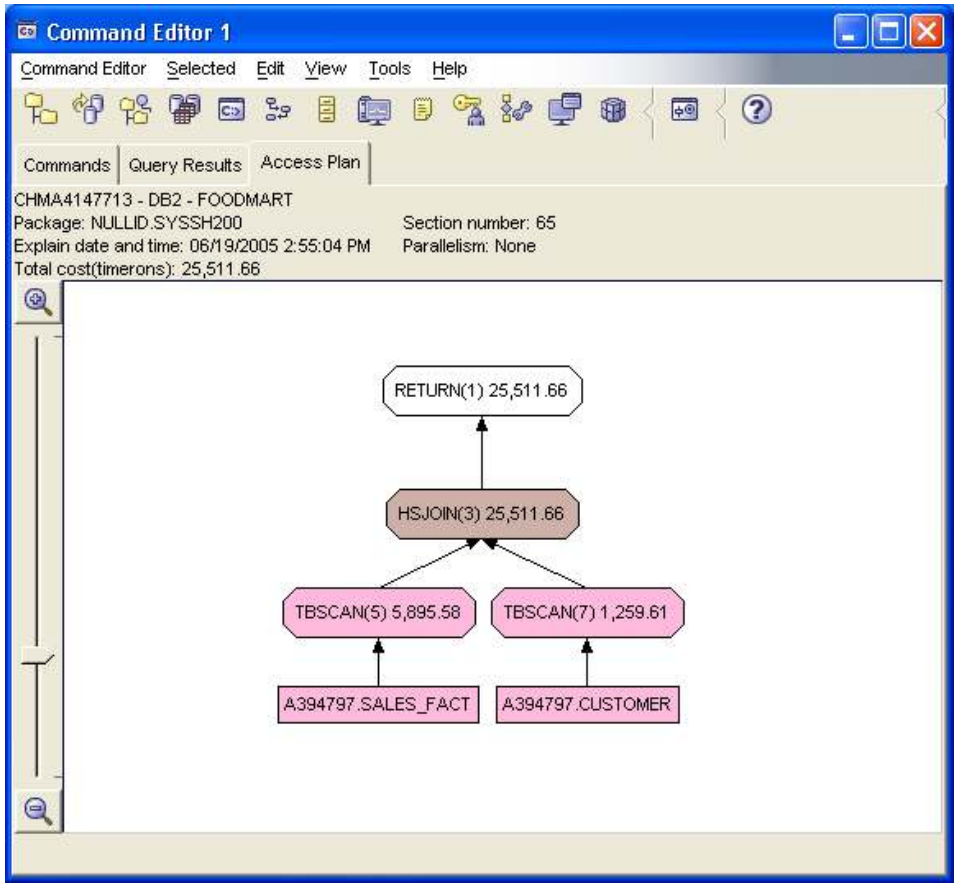
Query Execution Phases

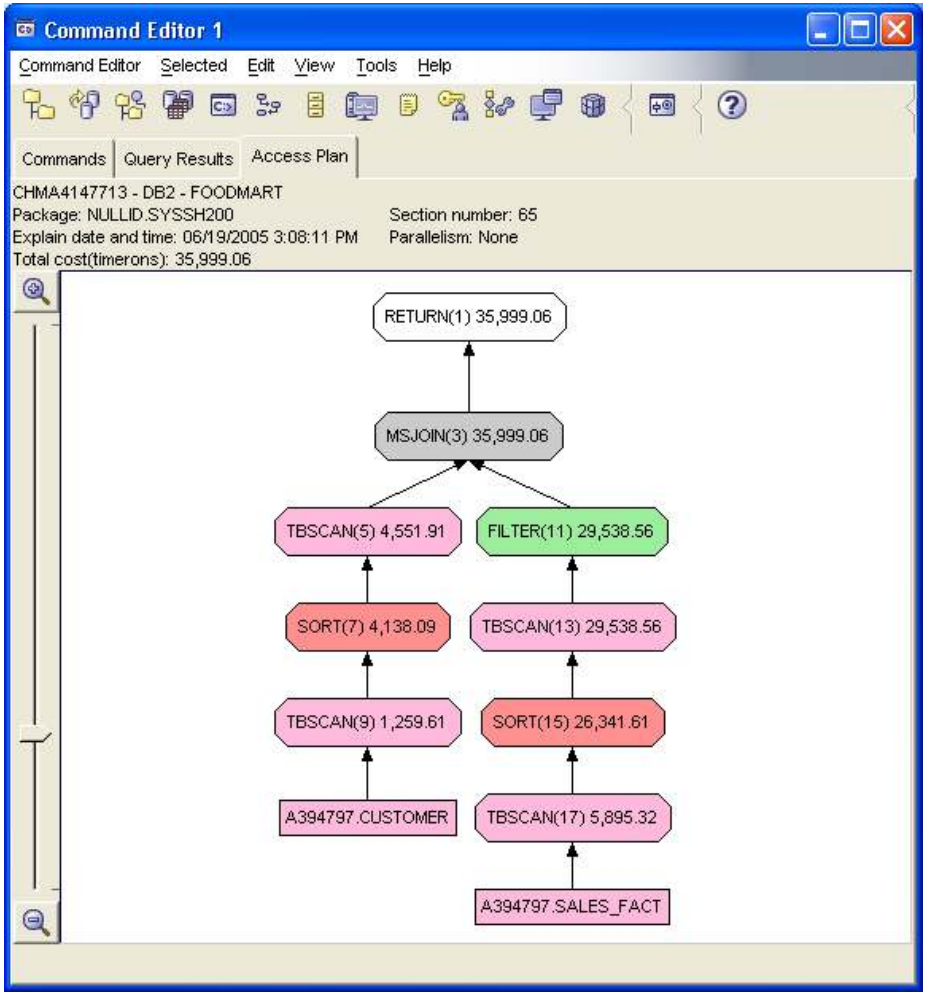


Query Optimization and Execution

- The query is transformed into an internal representation: operator graph
  - Relational operators as nodes and relations as leaves
- Optimization problem: find best operator graph (**what is "best"?**)
  - Find equivalent operator graphs for query
  - Determine optimal one based on cost model
  - Impact of statistics
- Important goal: minimization of intermediate result size
  - Perform selections and projections as early as possible (push-down)
  - Choose most efficient join order
  - Choose access paths / indexes for most efficient selection

Example Query Execution Plans in DB2





Outline

1. Introduction
2. **Indexing**
3. Star Joins and Join Indexes
4. Partitioning
5. Column Stores
6. Main-memory Databases
7. Materialized Views

B-Trees: Problems

- One-dimensional index structures
  - B-trees index a single attribute or a concatenation of attributes
  - Sequence, not set of attributes!
- B-tree useful only when selectivity is high (small number of tuples fulfill selection predicate)
- Candidate attributes for indexes:
  - ☺ Primary keys, “almost identifying” attributes
  - ☹ Attributes with small range of values, such as gender or canton
- Example: B-Tree over gender in Customer table
  - 1.000.000 rows
  - ➔ Two lists with approx. 500K entries
  - Query for female customers requires 500K accesses
  - ➔ Table scan is way faster

Bitmap Indexes

- Bitmap index over attribute A generates one bitmap for every possible value of A
- Each row in the table corresponds to a position in the bitmaps
- Prerequisite: consecutive numbering of rows
- Bit at position n in bitmap for value V equals 1: Row number n has value V for attribute A
- One bitmap per attribute value!

Bitmap Indexes

- Examples:
  - Bitmap index over attribute gender
  - 2 possible values, thus two bitmaps

Name	Region	Gender	Marital_status
Bill	e	m	s
Carol	n	f	m
Alice	e	f	d
Jill	ne	f	m
Joe	s	m	w
Jim	sw	m	m
Sarah	n	f	m
Laura	sw	f	s
David	sw	m	m
Don	n	m	m

Gender

F	M
0	1
1	0
1	0
1	0
0	1
0	1
1	0
1	0
0	1
0	1

Bitmap Indexes: Properties

- Index size is not an issue
- Bitmaps are sparse when indexed attributes has many different values
- Various compression approaches can be used
- Example:
  - 10K rows, 500 unique values
  - 0.2% * 10K = 20 1s in each bitmap (on average)
- RLE (run length compression)
  - 0 [2 Bytes length] 1 0 [2 Bytes length] ... 1 0 [2 Bytes length]
  - In principle: per bitmap 20 x 2 Bytes = 40 Bytes
  - 500 x 40 Bytes ≈ 20 kB for all bitmaps!

Bitmap Indexes: Properties (2)

- Because of their small size, bitmap indexes can be kept in main memory
  - Almost no I/O
  - Bitmap for thousands of rows in a single page
  - 4k page → 32768 rows

- In OLTP mode not useful because of locking!

☹ Via a single page (of the bitmap index) 32768 records will be locked!

Support for Complex Predicates

- Big advantage of bitmap indexes
  - Simple and efficient logical combination
  - Example: compute conjunction of bitmaps B1 and B2

```
for (i=0; i<B1.length; i++)
        B = B1[i] & B2[i];
```

  - Because of conjunction very selective predicates
- Example: “married female customers in region SouthWest”
  - Selectivity: 1/4 * 1/2 * 1/8 = 1/64
  - 10k rows of length 200 bytes (ca. 20 rows per 4K page)
  - Table scan: 500 pages
  - Access via bitmap index: 10k/64 ≈ 150 pages (worst case)

Support for Complex Predicates

F		SW		M		
0		0		0		0
1		0		1		0
1		0		0		0
1		1		1		1
0	and	0	and	0	=	0
0		1		1		0
1		1		1		1
1		0		0		0
0		1		1		0
0		0		1		0

Bitmap Indexes: DBMS Support

- Oracle
  - Bitmap index with compression
  - Supports logical operations on compressed bitmaps

```
create bitmap index gender_ix on customer(gender)
```

  - Usage:

```
select /*+ index (customer gender_ix) */ name
from customer
where mstatus = 'M'
```

Outline

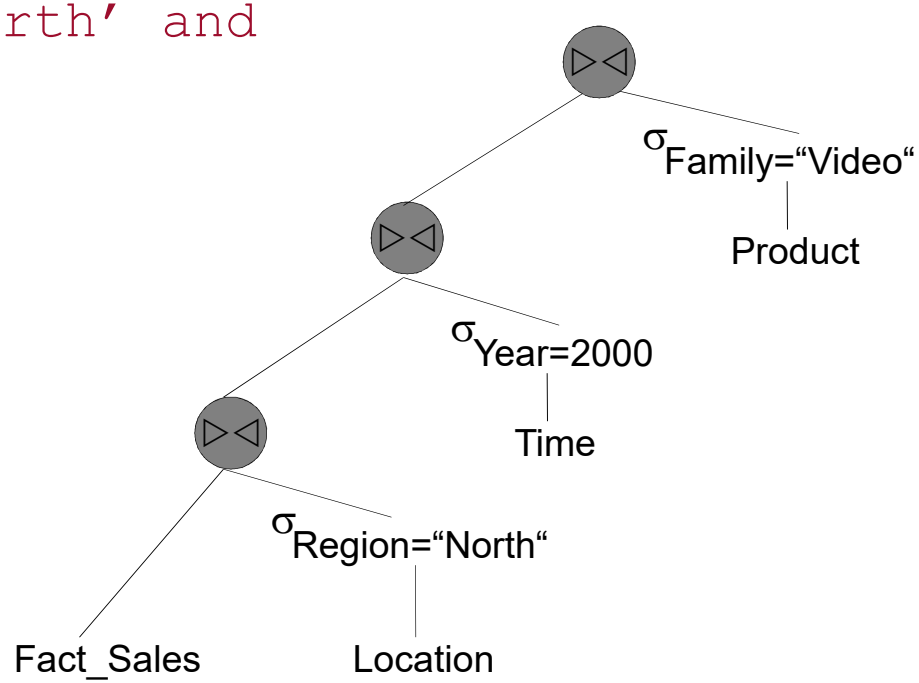
1. Introduction
2. Indexing
3. **Star Joins and Join Indexes**
4. Partitioning
5. Column Stores
6. Main-memory Databases
7. Materialized Views

Optimization of Star Queries

- Star queries typically contain multi-table joins
  - $\geq$ 3 tables
  - Selection of / grouping over attributes of the dimension tables
- Traditional optimization approach: minimize intermediate result size
- Problems:
  - Conventional cost-based optimizer tailored for two-table joins (left-deep trees)
  - Considers only tables that are related
  - The only table that is related to all others, is the fact table
  - Fact table typically has very large cardinality, dimension tables are much smaller
  - Intermediate join result become very large
  - Writing intermdiate results is very expensive

Conventional Star Join Processing: Example

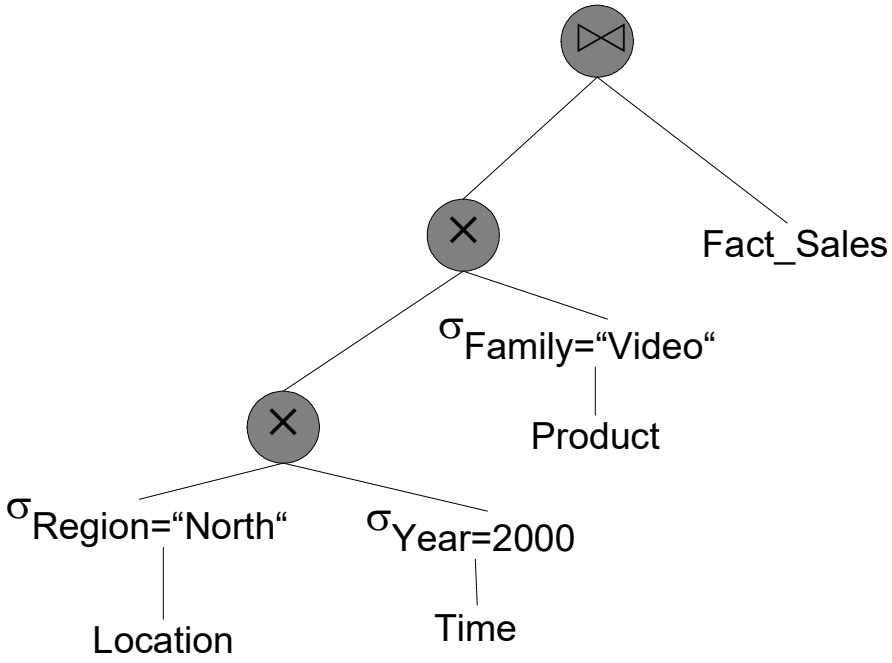
```
select sum(sales)
from   fact_sales     natural join
       product p      natural join
       location lnatural join  time t
where  p.family = 'video' and
       l.region = 'north' and
       t.year = 2000
```



Alternative Implementation of Star Joins

- First join dimension tables
- Compute complete cartesian product
- Reduction if only some of the dimension rows are joined
- Example: only months in the first quarter and all Samsung TVs
  - 4 TVs -> cartesian product has only 12 tuples
- Estimation is hard to make for the optimizer
  - Explizit hint in SQL query is possible

Star Joins: Example



- Star Join in Oracle and IBM DB2

Join Indexes

- Basic idea: index in a relation over an attribute of another relation
- Join indexes are pre-computed joins
- Rows which fulfill the join predicate are qualified up front
- Identifier pairs (or tuples) of the rleated rows are stored
- Additionally join indexes are clustered to accelerate access

Join Indexes: Examples

- Join index over fact table Facts and dimension table Article

Articles

ATID	ArticleID	Family
010	1001	Video
011	1002	Video
012	1003	Audio
013	1004	Audio

Facts

FTID	ArticleID	ShopID	Sales
001	1001	301	5
002	1002	302	4
003	1003	640	6
004	1004	302	3
005	1001	503	6
006	1002	640	7
007	1003	302	3
008	1003	301	1

$\bowtie$

ATID	FTID
010	001
010	005
011	002
011	006
012	003
012	007
012	008
013	004

Bitmap Join Indexes: Example

- Bitmap index on the fact table for each attribute of the dimension table that is restricted
- Bitmap index over product family in the fact table
- Bitmap index over region in fact table

Product

ArticleID	Family
1001	Video
1002	Video
1003	Audio
1004	Audio

Location

ShopID	Region
301	north
302	south
503	east
640	west

Fact_Sales

ArticleID	ShopID	Sales
1001	301	5
1002	302	4
1003	640	6
1004	302	3
1001	503	6
1002	640	7
1003	302	3
1003	301	1

video	audio
1	0
1	0
0	1
0	1
1	0
1	0
0	1
0	1

north	south	east	west
1	0	0	0
0	1	0	0
0	0	0	1
0	1	0	0
0	0	1	0
0	0	0	1
0	1	0	0
1	0	0	0

Bitmap Join Indexes: Example (2)

```
select   SUM(Sales)
from     Fact_Sales F NATURAL JOIN
         Product P NATURAL JOIN Location L
where    P.Family = 'Video' AND L.Region = 'North'
```

Product

ArticleID	Family
1001	Video
1002	Video
1003	Audio
1004	Audio

Location

ShopID	Region
301	north
302	south
503	east
640	west

video		north		
1		1		1
1		0		0
0		0		0
0	and	0	=	0
1		0		0
1		0		0
0		0		0
0		1		0

Fact_Sales

ArticleID	ShopID	Sales
1001	301	5
1002	302	4
1003	640	6
1004	302	3
1001	503	6
1002	640	7
1003	302	3
1003	301	1

Bitmap Join Indexes: DBMS Support

- Oracle: Bitmap Join Index

```
create  bitmap index sales_p_family
  on     fact_sales(product.family)
  from   fact_sales, product
  where  fact_sales.articleID = product.articleID
  …
```

Outline

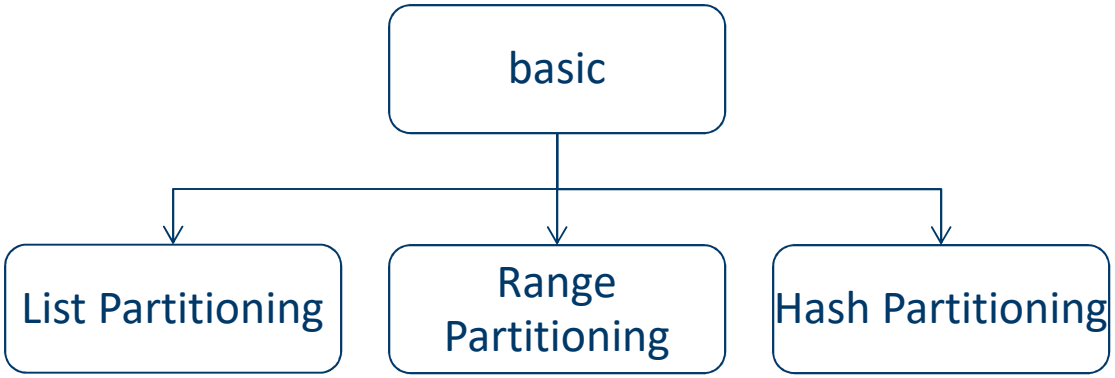
1. Introduction
2. Indexing
3. Star Joins and Join Indexes
4. **Partitioning**
5. Column Stores
6. Main-memory Databases
7. Materialized Views

Partitioning

- Many operations in a database system are unnnecessarily expensive when they have to process entire tables, although the operations affects only parts of the table
- Queries, backup, indexing, reorganization
- Partitioning means structuring of the data (of a table) according to some criterion, and the management and storage of these table parts
- In a data warehouse, time-based partitioning is obvious

Partitioning Concepts

- Every tuple in a partitioned tables will be assigned to exactly one partition
- The partitioning key is an attribute or an attribute combination which determines, into which parition the tuple belongs
- Partitioning approaches



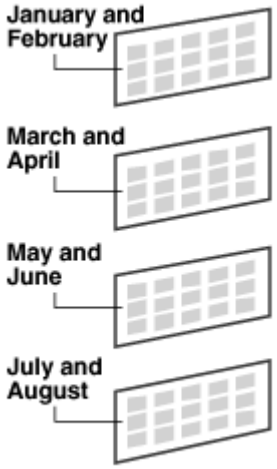
List Partitioning

- Each partition is defined by a list of discrete values (out of the range of the partitioning key)
- Default partition for all values non mentioned in any list

http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm#i460951

Range Partitioning

- Partitions are defined as subranges of the partitioning key's value range



http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm#i460951

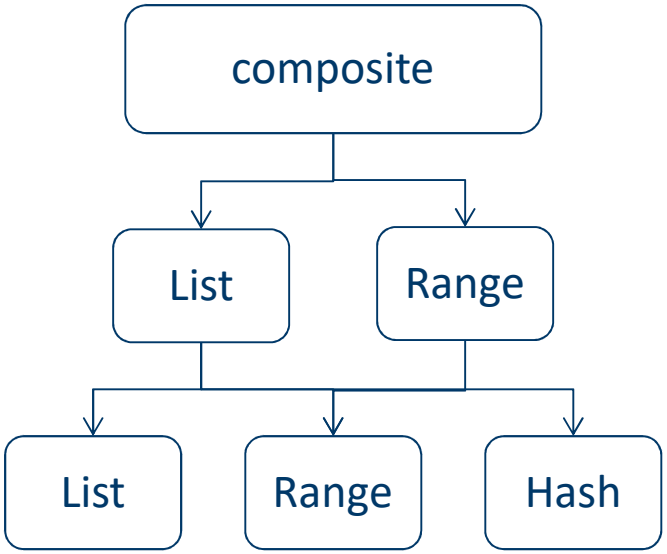
Hash Partitioning

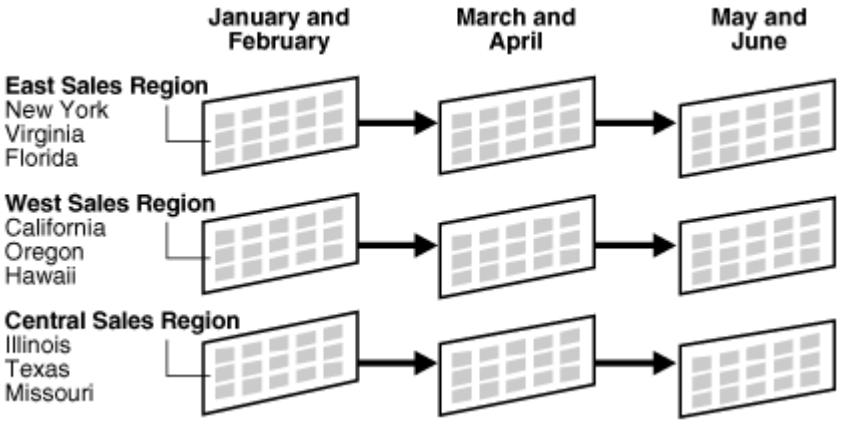
- The partitions are allocated through the result of a hash function
- Partitioning key as formal parameter

http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm#i460951

Composite Partitioning

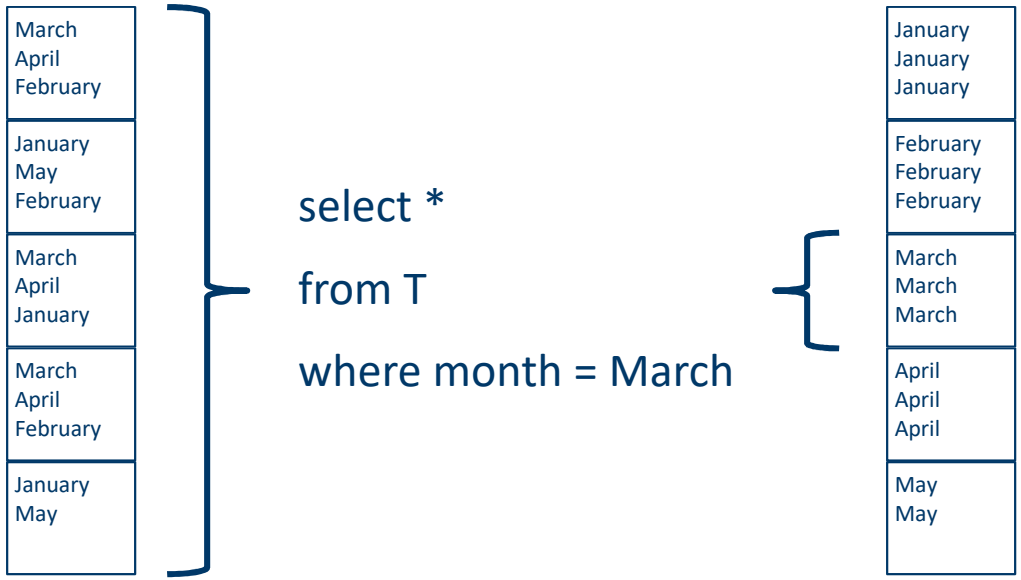
- Each partition can in turn be partitioned





Partition Pruning

- Partitioning can be exploited for predicate evaluation during query execution
- Only partitions that contain relevant data will be considered
- Partition Pruning



Partition Pruning: Example Dell dvdstore

- Table definition with partitioning

```
CREATE TABLE DS2.ORDERS
  (ORDERID NUMBER NOT NULL,
   ORDERDATE DATE NOT NULL,
   CUSTOMERID NUMBER,
   NETAMOUNT NUMBER(12, 2) NOT NULL,
   TAX NUMBER(12, 2) NOT NULL,
   TOTALAMOUNT NUMBER(12, 2) NOT NULL  )
  TABLESPACE ORDERTBS
  PARTITION BY RANGE (ORDERDATE)
    (PARTITION JAN2004 VALUES LESS THAN (TO_DATE('2004/02/01', 'YYYY/MM/DD')),
     PARTITION FEB2004 VALUES LESS THAN (TO_DATE('2004/03/01', 'YYYY/MM/DD')),
     PARTITION MAR2004 VALUES LESS THAN (TO_DATE('2004/04/01', 'YYYY/MM/DD')),
…
     PARTITION JAN2006 VALUES LESS THAN (TO_DATE('2006/02/01', 'YYYY/MM/DD')),
     PARTITION MAXVAL VALUES LESS THAN (MAXVALUE)    )  ;
```

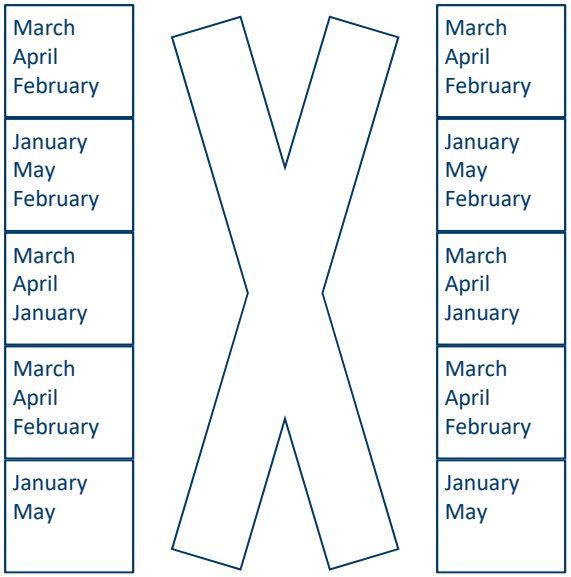
Partition Pruning: Example Dell dvdstore (2)

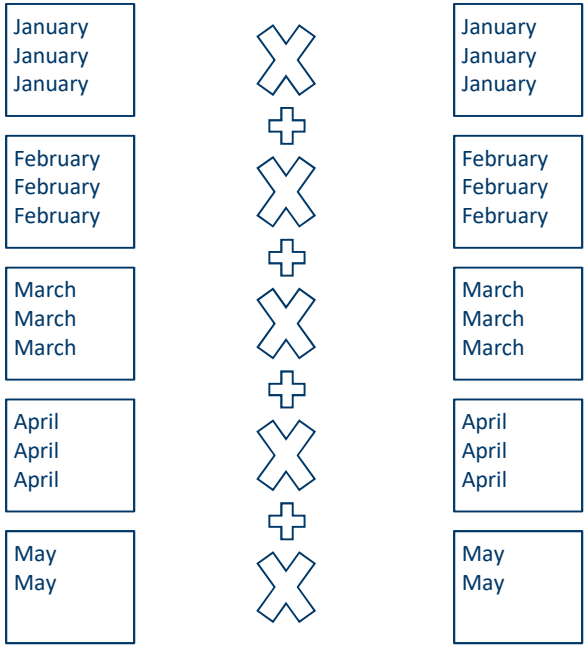
```
 explain plan for
   select orderid
     from orders
    where totalamount > 100 and
          orderdate between TO_DATE('2005/02/01',..) and TO_DATE('2005/03/31', …)  ;

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------
| Id  | Operation                | Name    | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop|
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |         |   252K|  4938K|     2   (0)| 00:00:01 |       |      |
|   1 |  PARTITION RANGE ITERATOR|         |   252K|  4938K|     2   (0)| 00:00:01 |    14 |   15 |
|*  2 |   TABLE ACCESS FULL      | ORDERS  |   252K|  4938K|     2   (0)| 00:00:01 |    14 |   15 |
-----------------------------------------------------------------------------------------------
```

Partition-wise Join

- Partitioning can also be exploited for join processing
- Join is computed partition-wise





Outline

1. Introduction
2. Indexing
3. Star Joins and Join Indexes
4. Partitioning
5. **Column Stores**
6. Main-memory Databases
7. Materialized Views

Column Stores: Introduction

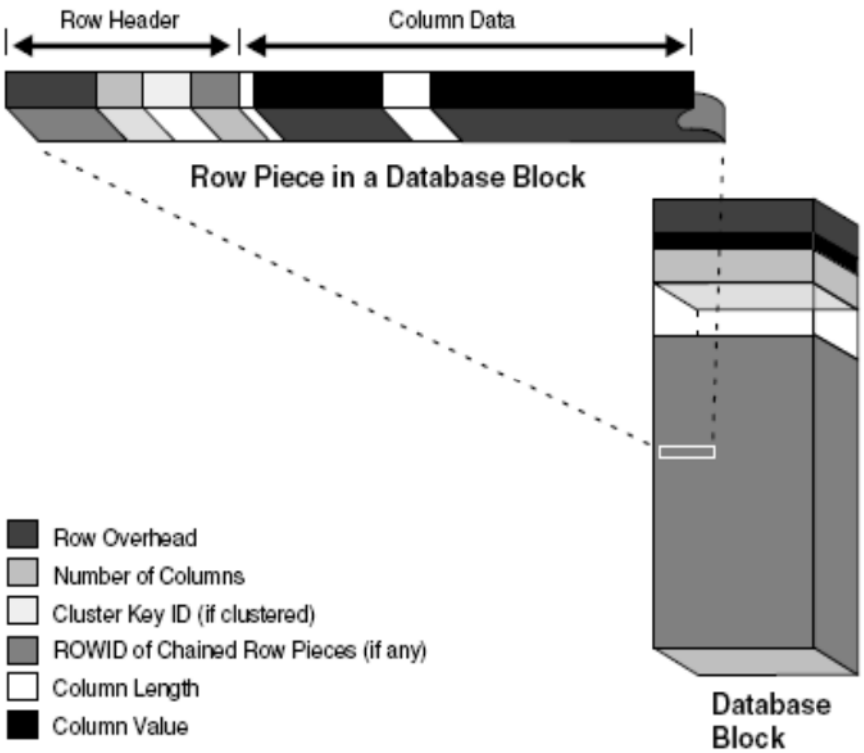
- Traditional DBMS store data row-wise
  - all attribute of a tuple are store together
  - Attributes are stored subsequently (except special cases)
- This kind of storage is very efficient for OLTP workloads and mixed profiles
  - Many, concurrent read/write transactions
  - Write-performance is critical
  - Tables are rather small (compared to data warehousing workloads)
  - Rather „thin“ tables (few columns)
  - Transactions access all columns of a table
  - Access to single or a few tuples

Column Stores: Introduction

- In DWH- and OLAP-workloads, row-based storage is suboptimal for several reasons:
  - Except loading, transactions are purely read-only
  - Query-performance is critical
  - Tables are very large (many rows)
  - Tables are often very wide (many columns), e.g., dimension tables
  - Many queries access only a small subset of the columns
  - Many queries compute aggregates
- Over the recent years, a new DBMS architecture has been developed and used productively that stores tables column-wise
  – **Column Stores**

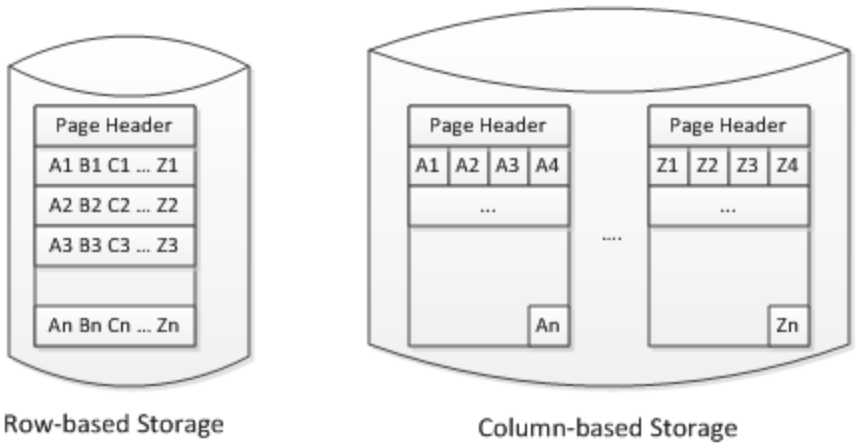
Row Stores: Storage Management

- Example Oracle:
  - Tuples are stored as records of variable length
  - A tuple can be stored as multiple records
  - Each record (piece) has a header and a data part (payload)



Column Stores: Storage Management

- In a column store, data are stored column-wise
- The mapping attribute value → tuple must be maintained
- In the general case, a column can be represented as a binary relation (OID, Attribute)
- In case the order of OIDs is intrinsic (e.g., order of insertion), then the OID-column is not necessary

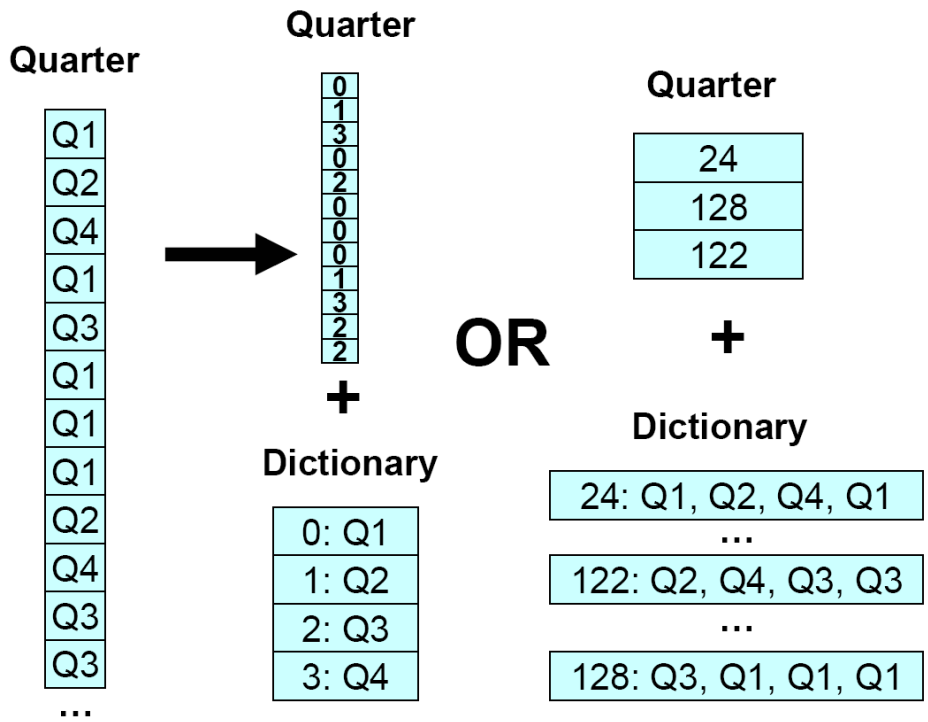


Column Stores: Compression

- A disadvantage of row-based storage is the lower compression potential
- Data are then compressed page-wise
  - Because of the heterogeneity of the attributes, compression factor is rather low
  - E.g. (Name, phone number, profession, date of birth)
- With column-wise storage, subsequent values are from the same domain, which means a much higher compression factor (up to 1:10)
- Examples: large fact tables, attributes with low cardinality in dimension tables
- Because of compression, fewer pages must be read from disk, which means significant I/O performance improvements
- In case operations can be executed on compressed data, pages do not have to be de-compressed in the buffer
  - More data can be kept in the buffer
  - No compute effort for de-compressing data
- Compression factors are better when compressed column is sorted
  - Are multiple columns of a table sorted, then explicit joins are necessary upon materialization

Compression in Column Stores: Dictionary Encoding

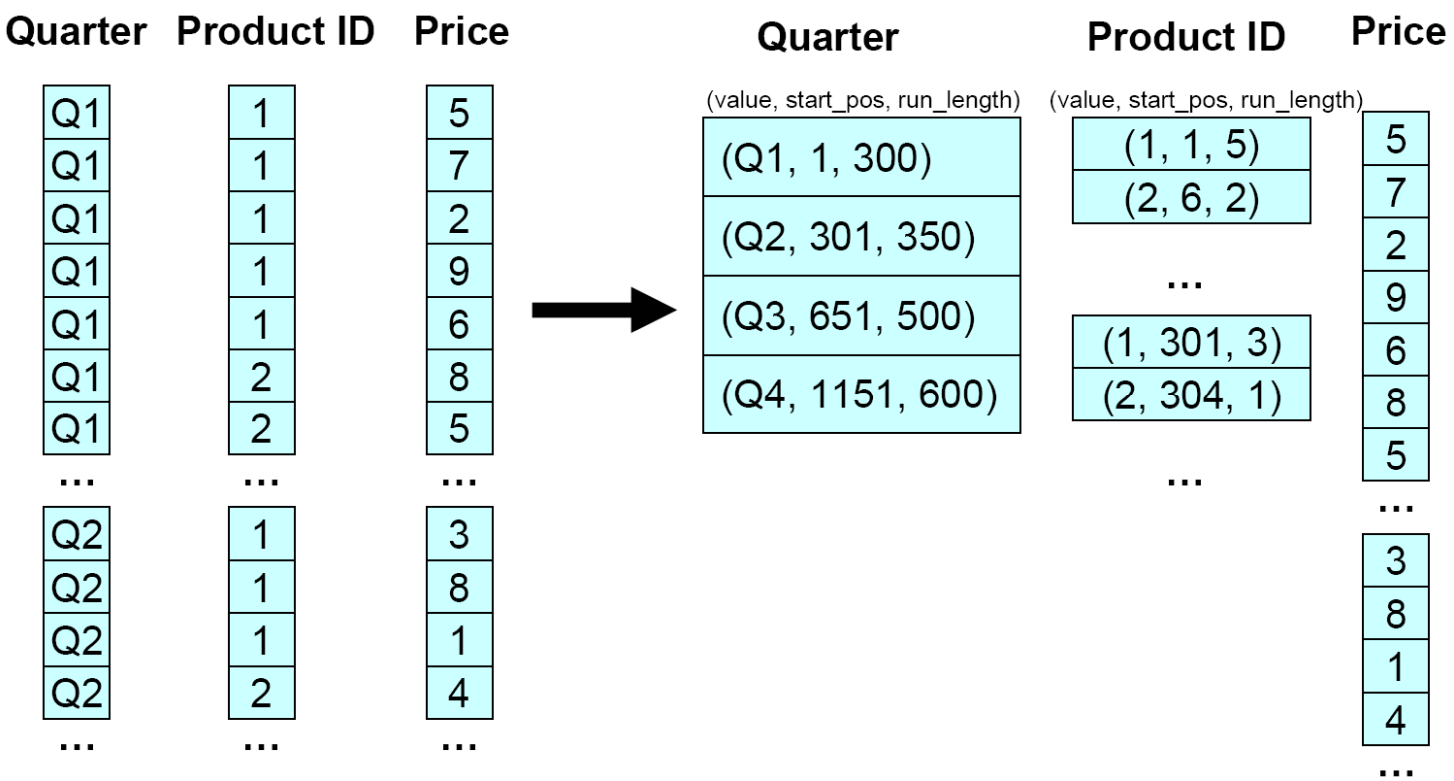
- Every unique value is replaced by a unique number
- Dictionary entries can also encode multiple values



Quelle: Prof. Dr. W. Lehner

Compression in Column Stores: Run-Length Encoding

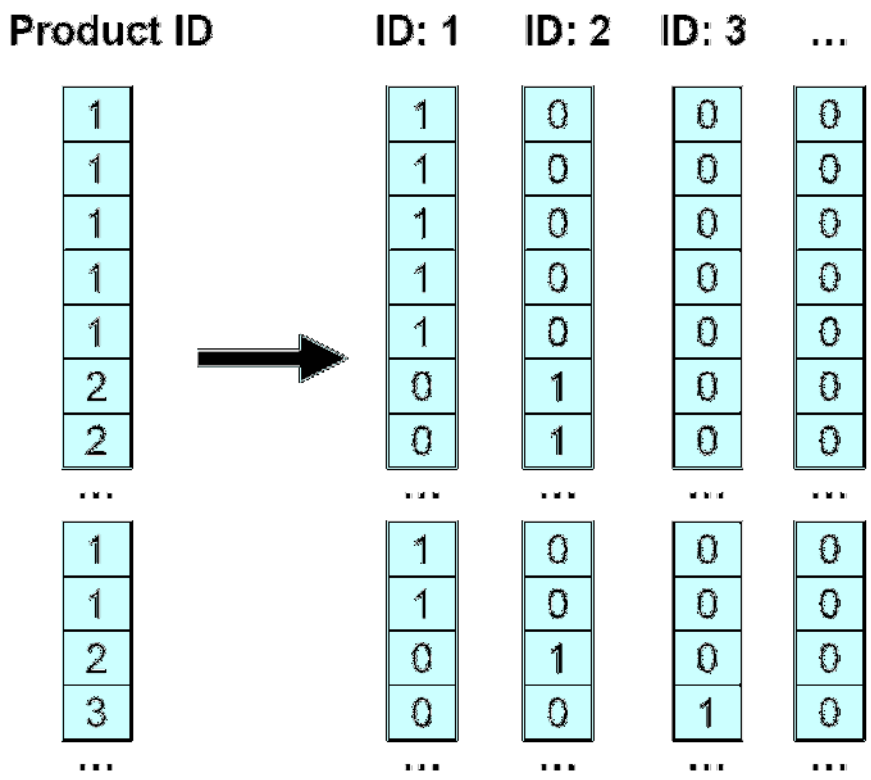
- n subsequent occurrences of the same value w starting at position p are replaced with the number of occurrences (i.e.,



Quelle:
Prof. Dr. W. Lehner

Compression in Column Stores: Bit Vector Encoding

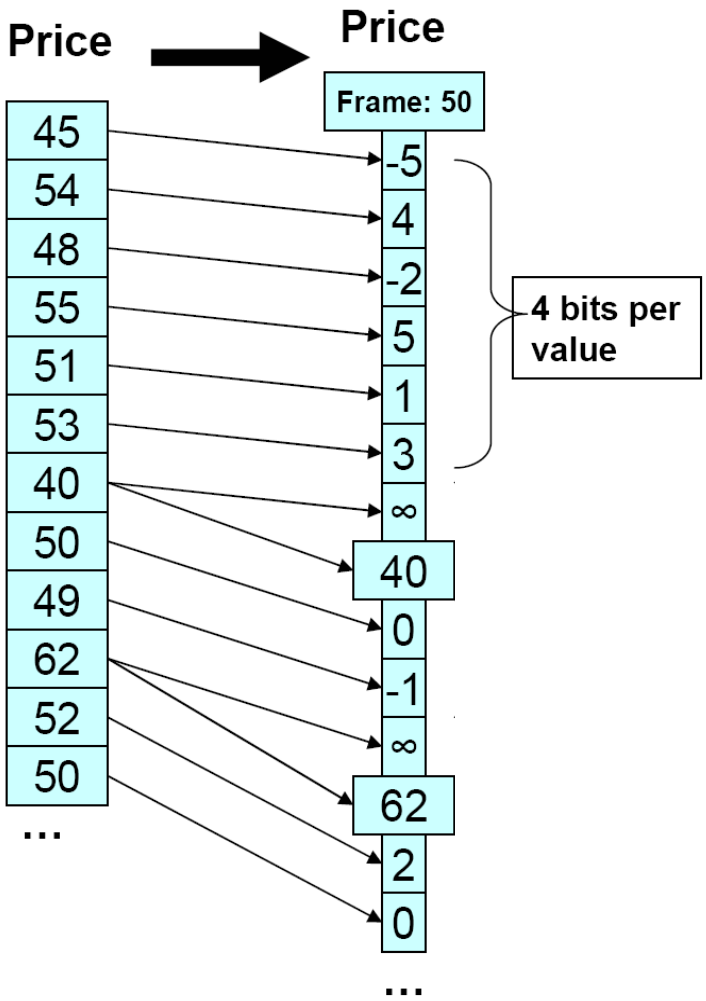
- Create a bitmap for every unique value
- See bitmap indexes
- Sparse bitmaps can be compressed further



Quelle:
Prof. Dr. W. Lehner

Compression in Column Stores: Frame of Reference Encoding

- Every value in a column is replaced with ist difference to a base value
- In case the difference becomes too large, a new reference value is used



Quelle:
Prof. Dr. W. Lehner

Column Stores: Query Evaluation

- Column storage enables siginificant query evaluation improvements
- With row storage, tables are processed on a tupel-by-tuple basis
- With column storage, entire blocks can be processed as a whole
  - High locality of code, which means good exploitation of modern hardware architectures

Column Stores: Types and Products

- Pure column stores
  - Sybase IQ, Exasol, Infobright, C-Store, Vertica
- hybrid systems (row- and column storage)
  - Teradata, Oracle Exadata, Greenplum
- Systems with column store indexing
  - SQL Server 2012

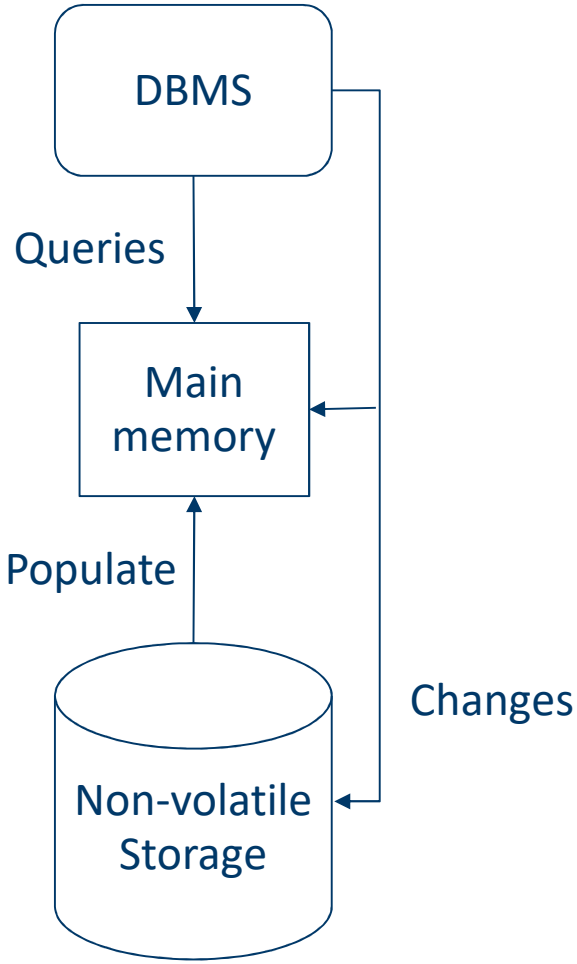
Outline

1. Introduction
2. Indexing
3. Star Joins and Join Indexes
4. Partitioning
5. Column Stores
6. **Main-memory Databases**
7. Materialized Views

In-memory Database Systems

- Two essential database features have traditionally been implemented by storing data on non-volatile storage:
  - ACID (transaction) guarantees, in particular durability and atomicity
  - ability to store and query data volumes that are much larger than the available main memory (RAM)
- Reading data from (and writing to) disk is however a critical performance bottleneck especially in DWH and analytic applications
- In the recent past, memory got bot cheaper and larger, so that large main memory configurations are now affordable
- **In-memory databases**

In-memory Database Systems



In-memory Database Systems

- **Pure in-memory database systems** keep data only in memory
  - at startup, data is read from a persistent representation (e.g. disk)
  - updates have to be persisted to guarantee durability, e.g. by persisting the transaction log
  - Query execution (as well as other features) is geared towards main-memory representation of data
- **Hybrid in-memory database systems** keep data in memory and on disk
  - only often used, performance critical data is pinned to main memory
  - more flexibility for applications
  - but also (much) higher complexity for optimizer, etc.

In-memory Database Systems
Row and Column Formats

- The data representation format can be decided based on intended applications, or offered as a choice for applications
- in-memory row format represents data conventionally row-wise
  - well-suited for frequent changes (OLTP)
  - suboptimal for DWH and analytics
- in-memory column format represents data column-wise (see above)
  - yields all advantages of column stores
  - in particular, because of compression, more data can be kept with a given main memory size

In-memory Database Systems
Example: Oracle

- Oracle In-memory is a hybrid in-memory database system
- row and column format are supported
- In-memory column store can be chosen for tables, partitions, materialized views, and tablespaces
- Example: `alter table sales_fact inmemory;`
- Exclusion of individual columns possible
- Column compression (6 different levels)
  - none
  - minimal (for DML performance)
  - for query performance
  - for space efficiency

Outline

1. Introduction
2. Indexing
3. Star Joins and Join Indexes
4. Partitioning
5. Column Stores
6. Main-memory Databases
7. **Materialized Views**

Materialized Views

- Pre-calculation is a common solution for performance improvement in database systems
  - Pre-computation of intermediate query results
  - Intermediate result will be stored in the database („materialized“) and reused for future queries
  - Aggregation is typically used in order to reduce data volume
  - Materialized views, summary tables, pre-aggregates
- Materialized views can improve query performance
  - Typical improvements by one or more factors of magnitude
- Trade-off redundancy: Redundancy → Storage, consistency

Materialized Views: Example

- Query

```
SELECT      P.Group, L.State, SUM(F.Sales)
FROM        Fact_Sales F       NATURAL JOIN
                 Product P     NATURAL JOIN Location L
WHERE       L.Country = „USA"
GROUP BY    P.Group, L.State
```

- Materialized View

```
CREATE TABLE Fact_Sales_View AS
  SELECT      P.Family, L.City, T.Year, SUM(F.Sales), COUNT(F.Sales)
  FROM        Fact_Sales F       NATURAL JOIN
                 Product P       NATURAL JOIN
                 Location L      NATURAL JOIN Time T
  GROUP BY    P.Family, L.City, T.Year
```

➔Usage of Fact_Sales_View instead of Fact_Sales possible

➔Fact_Sales_View << Fact_Sales → query execution accleration

Materialized Views: Example (2)

Cust_id	city	...	Store_id	state	...	Cust_id	Store_id	Sales	Cost	units
7204	Milwaukie	...	11	OR	...	7204	11	5.55	2535.0000	3
6758	Beaverton	...	11	OR	...	6758	11	5.43	1.9548	3
6616	Beaverton	...	11	OR	...	6616	11	9.32	2.9824	4
4243	Portland	...	11	OR	...	4243	11	2.44	1248.0000	2
2039	W. Linn	...	11	OR	...	2039	11	1.65	0.5115	3
1594	Milwaukie	...	11	OR	...	1594	11	5.19	2.4912	3
7464	Beaverton	...	11	OR	...	7464	11	6.72	2.6208	4
4808	W. Linn	...	11	OR	...	4808	11	5.49	2.6352	3
5858	Oregon City	...	11	OR	...	5858	11	81.00	2.6433	3
3718	Oregon City	...	11	OR	...	3718	11	12.84	5.7780	4
3401	Lake Oswego	...	11	OR	...	3401	11	2.30	0.8510	2
1202	W. Linn	...	11	OR	...	1202	11	5.88	258.0000	3

count(*): 244,720

Materialized Views: Example (3)

State	City	Date	Sales_sum	Sales_cnt
OR	Portland	2004-12-04	735	227
BC	Victoria	2004-02-06	236	73
WA	Seattle	2004-04-17	363	117
WA	Tacoma	2003-03-22	702	219
WA	Walla Walla	2003-12-26	61	32
Zacatecas	Camacho	2004-09-08	390	119
CA	San Diego	2004-06-14	437	135
WA	Tacoma	2004-06-25	811	255
CA	Los Angeles	2004-03-21	463	147
WA	Spokane	2003-03-01	372	118
Guerrero	Acapulco	2004-10-08	313	95
BC	Victoria	2004-04-09	163	50

count(*): 2,145

Issues When Using Materialized Views: Selection

- Selection problem (Administrator)
  - Which views should we materialize?
  - Use as litle storage as possible, gain as much as possible execution time

A	B	C
a1	b1	c7
a2	b2	c6
a3	b1	c5
a4	b2	c4
a5	b1	c3
a6	b2	c2
a7	b1	c1

D	E	F
d1	e1	f7
d2	e2	f6
d3	e1	f5
d4	e2	f4
d5	e1	f3
d6	e2	f2
d7	e1	f1

B	agg(C)
b1	c8
b2	c9

B	E	agg(C)	agg(F)
b1	e1	c10	f10
b2	e1	c11	f11
b1	e2	c12	f12
b2	e2	c13	f13

E	agg(F)
e1	e8
e2	e9

Issues When Using Materialized Views: Usage

- Usage problem (Optimizer)
  - Which views should be used to answer a query?
  - Cost-based construction of replacements

select B, agg(C)
from …
group by ...

A	B	C
a1	b1	c7
a2	b2	c6
a3	b1	c5
a4	b2	c4
a5	b1	c3
a6	b2	c2
a7	b1	c1

D	E	F
d1	e1	f7
d2	e2	f6
d3	e1	f5
d4	e2	f4
d5	e1	f3
d6	e2	f2
d7	e1	f1

B	agg(C)
b1	c8
b2	c9

B	E	agg(C)	agg(F)
b1	e1	c10	f10
b2	e1	c11	f11
b1	e2	c12	f12
b2	e2	c13	f13

E	agg(F)
e1	e8
e2	e9

Issues When Using Materialized Views: Refresh

- Refresh problem (Administrator)
  - How and how often should materialized views be refreshed?
  - Incremental update possible?

insert into …
update …
delete from ...

A	B	C
a1	b1	c7
a2	b2	c6
a3	b1	c5
a4	b2	c4
a5	b1	c3
a6	b2	c2
a7	b1	c1

D	E	F
d1	e1	f7
d2	e2	f6
d3	e1	f5
d4	e2	f4
d5	e1	f3
d6	e2	f2
d7	e1	f1

B	agg(C)
b1	c8
b2	c9

B	E	agg(C)	agg(F)
b1	e1	c10	f10
b2	e1	c11	f11
b1	e2	c12	f12
b2	e2	c13	f13

E	agg(F)
e1	e8
e2	e9

Selection of Materialized Views

- Trade-off:
  - ☺ Improved query execution time
  - ☹ Higher storage costs
  - ☹ Refresh costs
- Impact of granularities, i.e. grouping criteria:
  - Fine granularity
    - Higher storage consumption, less performance improvement
    - More often usable
  - Coarse granularity
    - Less storage consumption, high performance improvement
    - Less often usable

Usage of Materialized Views

- Goal:
  - Construct a replacement (of the base queries in the original query) by using one or more materialized views
  - Modified query should obviously be equivalent, i.e., return the same result as the original one
- The replacement has to add a “delta” (additional work) in order to bridge the gap between the view(s) and the initial query
- Problem:
  - Given query Q and n materialized views, find the optimal replacement

Equivalence of Queries

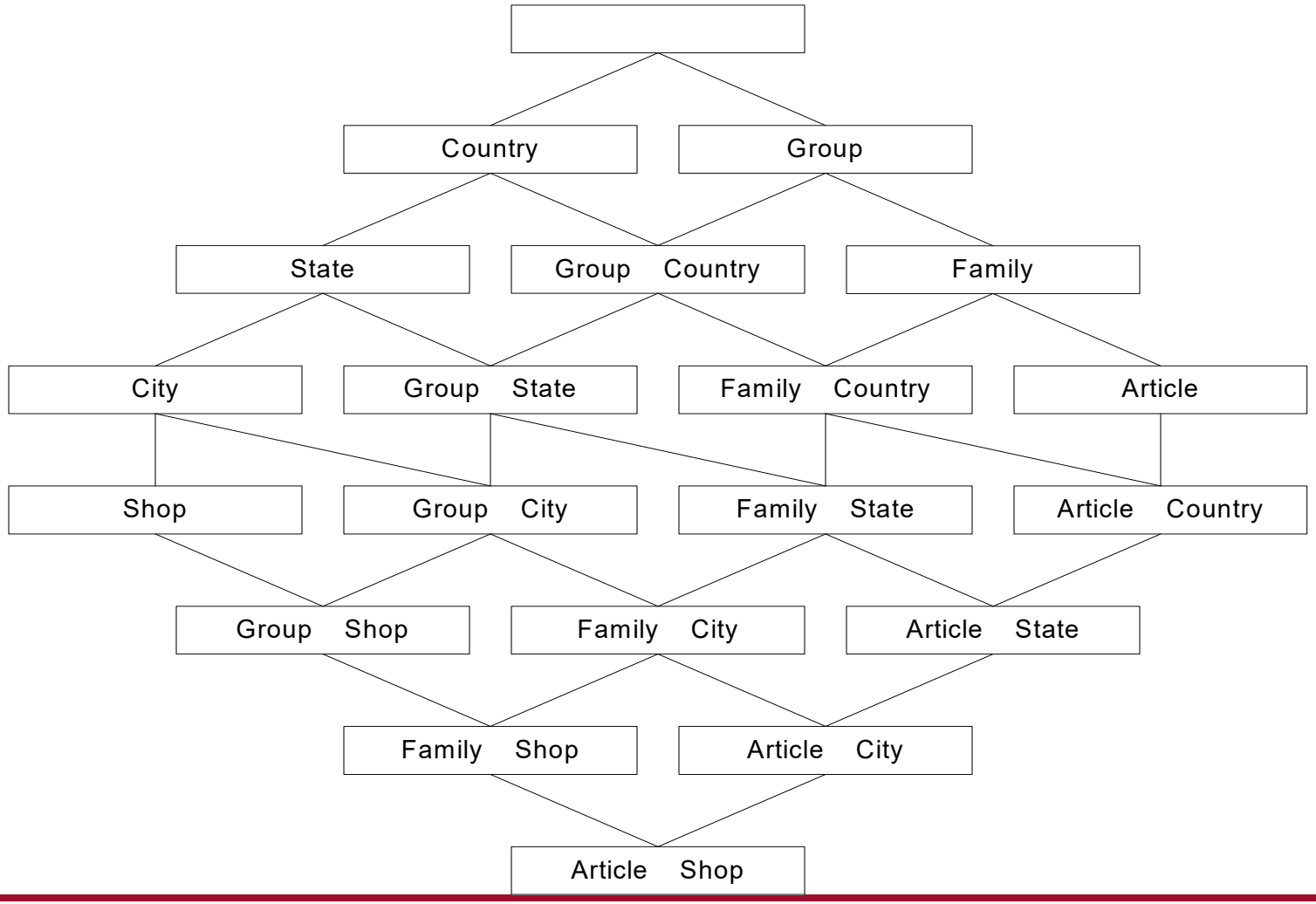
- Eqiuvalence requires original query and MV to meet some prerequisites
- Granularity of the view needs to be finer than that of the query (why ?)
- Selection predicates are less selective
- The query‘s selection predicates can be applied to the view
- Desired aggregates can be computed over the view:
  - additive aggregates:
    SUM in query→ SUM in view
  - Additive computable aggfregates:
    AVG in query → SUM and COUNT in view
- Aggregate restrictions in the view
  - Can be a problem, because they depend on the granularity
  - SUM(Sales) < 10 on level L.City does not help much for SUM(Sales) on level L.State

Derivability of Grouping Criteria

- Relation „finer“
  - Query with grouping attributes G‘ can be computed based on G, if
    - G is a superset of G’, respectively
    - G does not contain coarser granularities than G‘
  - formally: $G = (G_1, ..., G_n) \leq G' = (G'_1, ..., G'_m)$

    $\Leftrightarrow \forall G'_j \in G' \; \exists G_i \in G: G_i \rightarrow G'_j$
  - Partial ordering
- Aggregation lattice

Aggregation Lattice: Example



Derivability of Grouping Attributes

- Functional dependencies (implementing dimension hierarchies) can be deduced from foreign keys or referential constraints
- However, Star schemas are denormalized
- The optimizer thus has no possibility to deduce the structure of dimension hierarchies
- Classification hierarchies (dimension hierarchies) must be made known to the optimizer
- create dimension clause in Oracle

Derivability of Grouping Attributes

- Oracle: Create Dimension

```
CREATE DIMENSION customers_dim
  LEVEL customer        IS (customers.cust_id)
  LEVEL city            IS (customers.cust_city)
  LEVEL state           IS (customers.cust_state_province)
  LEVEL country         IS (countries.country_id)
  LEVEL region          IS (countries.country_region)
   HIERARCHY geo_rollup (customer    CHILD OF
                        city         CHILD OF
                        state        CHILD OF
                        country      CHILD OF
                        region
   JOIN KEY (customers.country_id) REFERENCES country)
   ATTRIBUTE customer DETERMINES
                  (cust_first_name, cust_last_name, cust_gender,
                  cust_marital_status,  cust_year_of_birth,
                  cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
```

Replacement in Oracle

- Materialized view

```
create materialized view sales_cat_quarter_mv
  …
  enable query rewrite /* automatic construction of a replacement*/
  as
       select     p.category, t.quarter,
                  sum(store_sales) as sum_sales
       from       product_all p, time_all t, sales_fact f
       where      f.product_id = p.product_id and f.time_id=t.time_id
       group by   p.category, t.quarter;
```

Replacement in Oracle

- Query

```
select   p.department, t.year, sum(store_sales)
from     product_all p, time_all t, sales_fact f
where    f.product_id = p.product_id and f.time_id = t.time_id
group  by p.department, t.year;
```

- Generated replacement

```
select p.department, t.year, sum(sum_sales)
from   (select distinct category, department from product_all) p
       (select distinct quarter, year from time_all) t,
       sales_cat_quarter_mv v
where  v.category = p.category
       and v.quarter = t.quarter
group  by p.department, t.year;
```

Refreshment of Materialized Views

- Base relations are updated (as part of ETL process)
  - Materialized views are out of sync and need to be updated as well
  - Different update strategies meet different currency requirements
- Query performance should not be the only criterion
  - Many people tend to think that storage costs can be neglected
  - Potential performance gains tempts to define many materialized views
  - Especially if selection algorithms are not very smart andhave many exceptions
- Update performance
  - Update cost are not small
  - Conventional refresh algorithms take database offline during refresh

⇨ Performance gains and update costs must be balanced!

Materialized Views Update Options

- Re-materialization
- Deletes old state of the view
- Re-computes new state from scratch

⇨ Often too expensive

- Incremental refresh
- Obtain new state by applying delta to old state:
  $V' = V + \Delta^{V} = (V - \Delta^{-V}) \cup \Delta^{+V}$
- Incremental refresh is the more efficient option in typical cases
- When number of modifications is large, incremental refresh may be more expensive than re-materialization

Refresh: Example

- Materialized view, grouped by month and product
- sum and count in columns

insert into …
update …
delete from …

Product	Date	Sales
a	2002-06-03	11
b	2002-06-03	12
c	2002-06-03	13
a	2002-06-04	10
b	2002-06-04	10
a	2002-06-05	9
a	2002-06-06	12

Product	Month	sum_sales	cnt_sales
a	2002-06	30	3
b	2002-06	22	2
c	2002-06	13	1

+

Product	Month	sum_sales	cnt_sales
a	2002-06	12	1

⇩

Product	Month	sum_sales	cnt_sales
a	2002-06	42	4
b	2002-06	22	2
c	2002-06	13	1

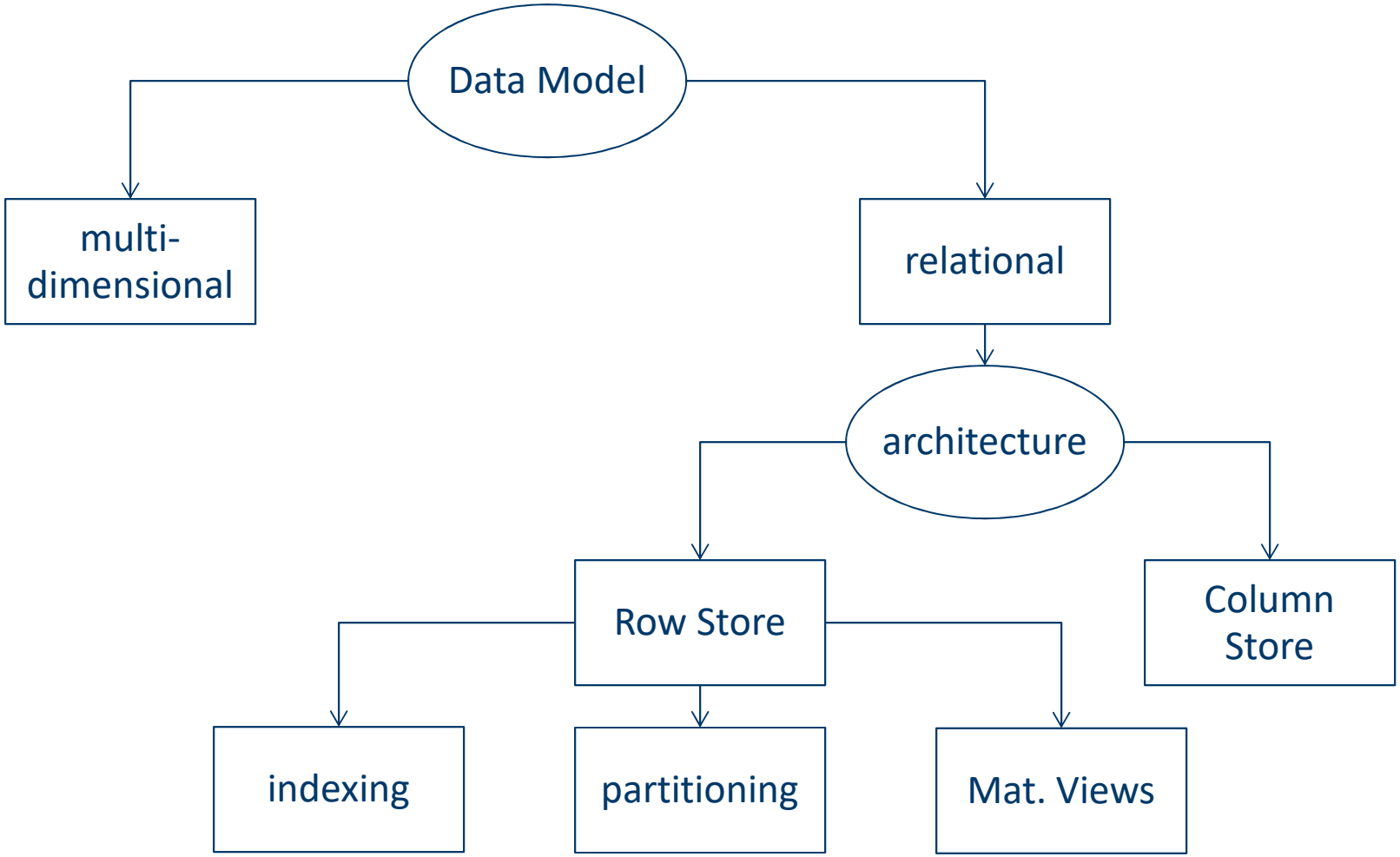
Oracle Materialized Views

- View materialization time
  - At definition time: build immediate
  - At a later time: build deferred
- View refresh time
  - on commit: automatically after commit of the base table(s)
  - on demand: manually (refresh viewname, refresh_all_mviews, refresh_dependent)
- Refresh options
  - complete: complete re-materialization
  - fast: incremental refresh using change of the redo log
  - force: fast if possible, otherwise complete
  - never

Oracle Materialized Views: Example

```
create materialized view article_country_year_sales_mv
  build immediate
  refresh complete on commit
  enable query rewrite
  as
      select p.article, s.country, t.year,
        sum(sales) as sumsales
      from products p, shops s, time t, fact f
      where  p.article = f.article
        and s.outlet = f.outlet
        and t.day = f.day
      group by p.article, s.country, t.year
```

The End


Data Model
multi-dimensional
relational
architecture
Row Store
Column Store
indexing
partitioning
Mat. Views