

PA220: Database systems for data analytics

Data Warehouse Indexing & Optimization

Contents

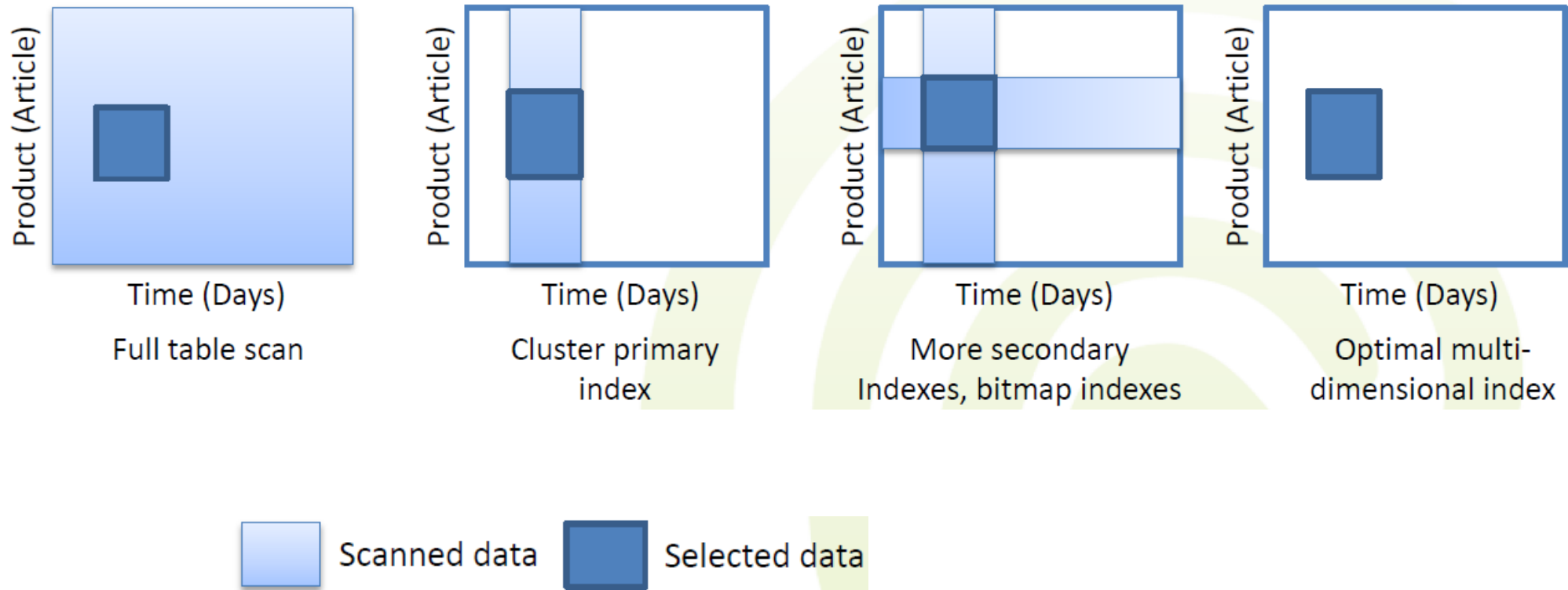
- Approaches to indexing
- Data partitioning
- Joins
- Materialized views

Why Indexes?

- Consider a 100 GB table; at 100 MB/s read speed we need 17 minutes for a full table scan
- Query for the number of “Bosch S500” washing machines sold in Germany last month
 - Applying restrictions (product, location) the **selectivity** would be strongly improved
 - If we have 30 locations, 10,000 products and 24 months in the DW, the selectivity value is $1/30 * 1/10,000 * 1/24 = 0,000\ 000\ 14$
- So... we read 100 GB for 1,4KB of data
- The problem is: *how to filter data in a fact table as much as possible*

Why Indexes?

- Reduce the size of read pages of data cube to a minimum with indexes

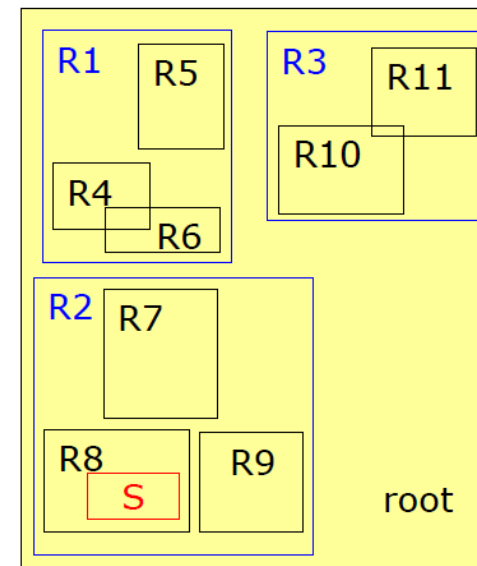
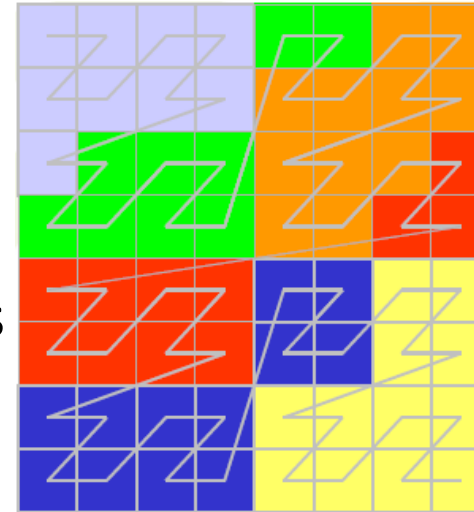


Index Types

- Tree structures
 - B⁺-tree, R-tree, ...
- Hash based
 - Dynamic hash table
- Special
 - Bitmap index
 - Block-Range INdex (in Pg)

Multidimensional Data

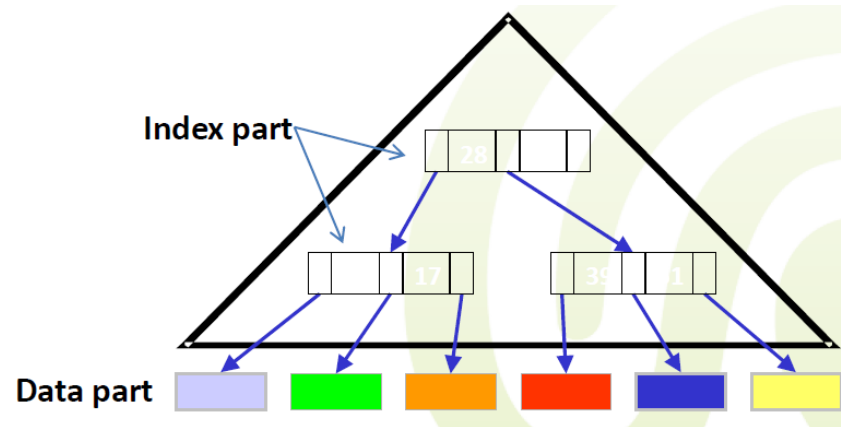
- B⁺-tree
 - classic structure – very efficient in updates
 - supports point and range queries
 - limited to 1D data
- UB-tree
 - uses B^{*}-tree and
 - Z-curve to linearize n-dim data
- R-tree
 - wrapping by n-dim rectangles
 - R⁺, R^{*}, X-tree



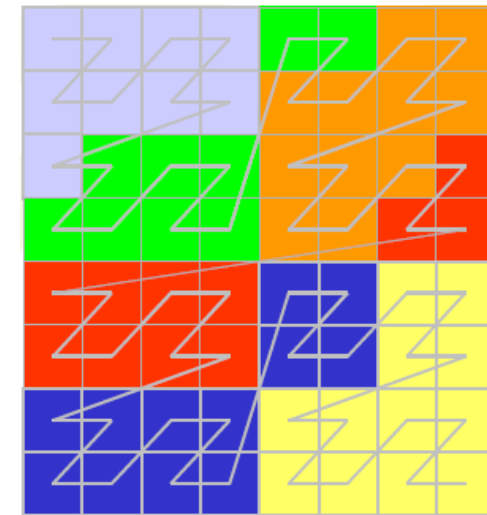
UB-Trees

- Convert n-dim data to a single dimension by the Z-curve and

- Index by B* tree



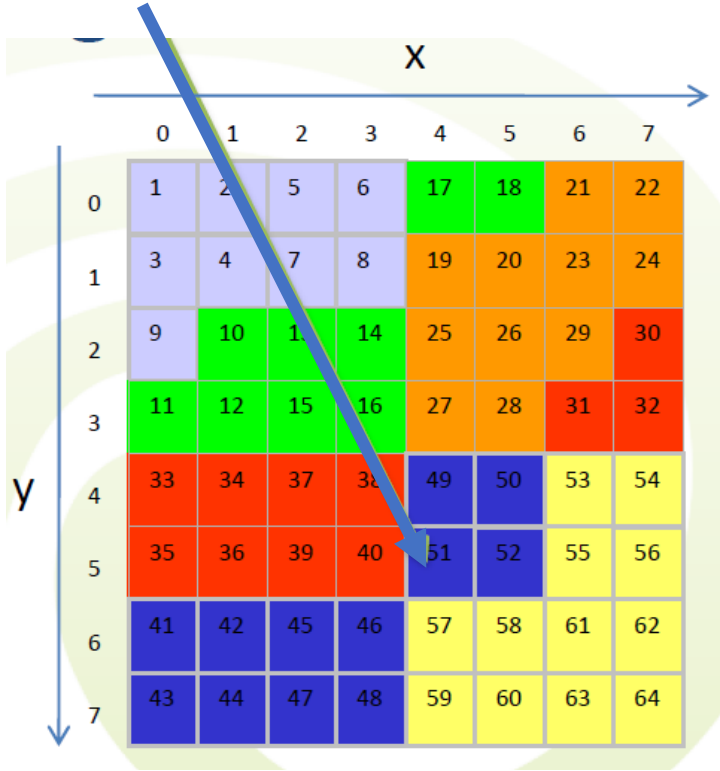
- The Z-curve provides for good performance for range queries!
 - Consecutive values on the Z-curve index similar data
 - Similarity by means of neighborhood



UB-Trees

- Z-Value address representation
 - Calculate the z-values such that neighboring data is clustered together
 - Calculated through bit interleaving of the coordinates of the tuple
 - To localize a value with coordinates one must perform de-interleaving

row at x=4, y=5



For Z-value 51, we have the offset 50.
50 in binary is 110010

$$\begin{array}{c}
 \text{Z-value} = 110010 \\
 \swarrow \quad \nwarrow \\
 Y = 101 = 5 \quad X = 100 = 4
 \end{array}$$

We have Z-regions – describes one block in storage.
E.g. [1-9], [10-18].

UB-Trees – Range Query

- Range queries (RQ) in UB-Trees

- Each query can be specified by 2 coordinates
 - q_a (the upper left corner of the query rectangle)
 - q_b (the lower right corner of the query rectangle)

- Range Query Algorithm

1. Calculate z-values for q_a and q_b
2. Get a node with Z-Region containing q_a
 - e.g., Z-Region of q_a is [10:18]
3. The corresponding page is loaded and filtered with the query predicate
 - E.g., value 10 has after de-interleaving $x=1$ and $y=2$, which is outside the query rectangle

Q: $x \in [2;5], y \in [3;6]$

1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

UB-Trees – Range Query

- Range Query Algorithm (cont.)

4. After q_a , all values on the Z-curve are de-interleaved and checked by their coordinates

- The data is only accessed from the disk.
- The next jump point on the Z-curve is 27.

5. Repeat Steps 2 and 3 until the decoded end-address of the last filtered region is bigger than q_b

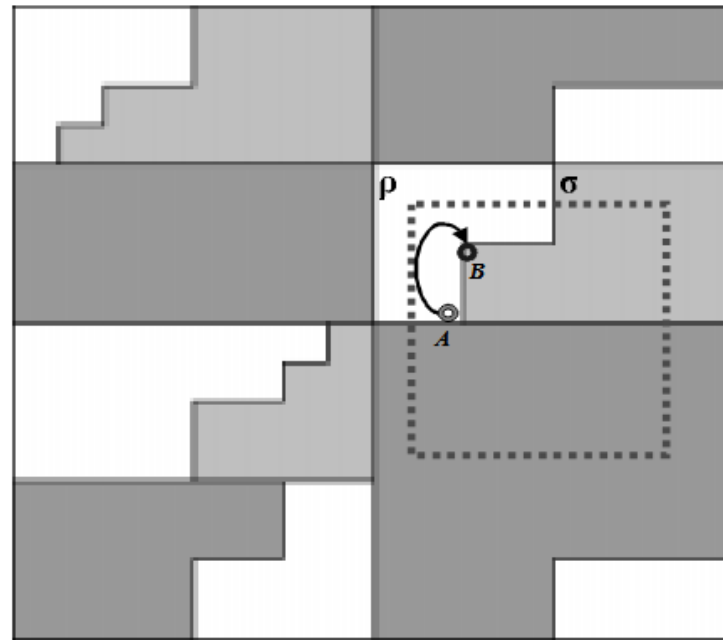
Calculating the *jump point* mostly involves:

- Performing **bit operations** and comparisons
- 3 points: q_a , q_b and the current Z-Value

1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

UB-Trees – next “jump” point

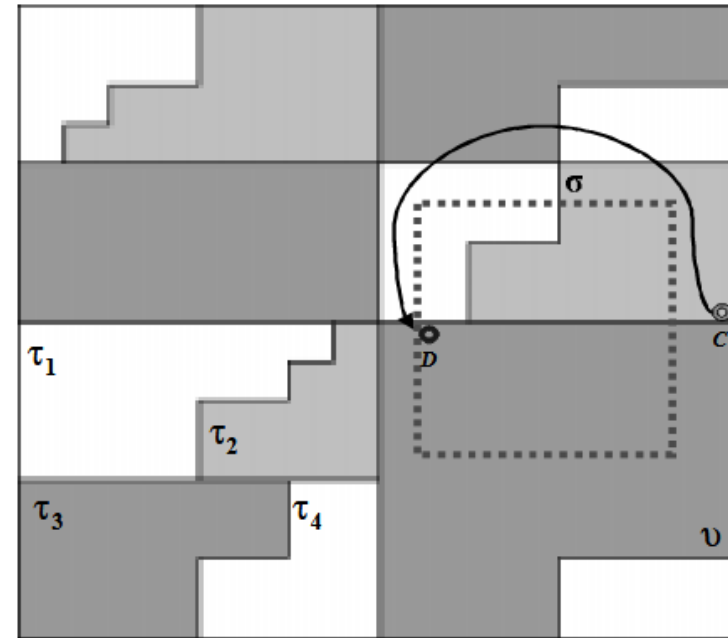
- Idea of getting next jump point



(a)

Region Address \odot \longrightarrow \bullet Next Intersection Point

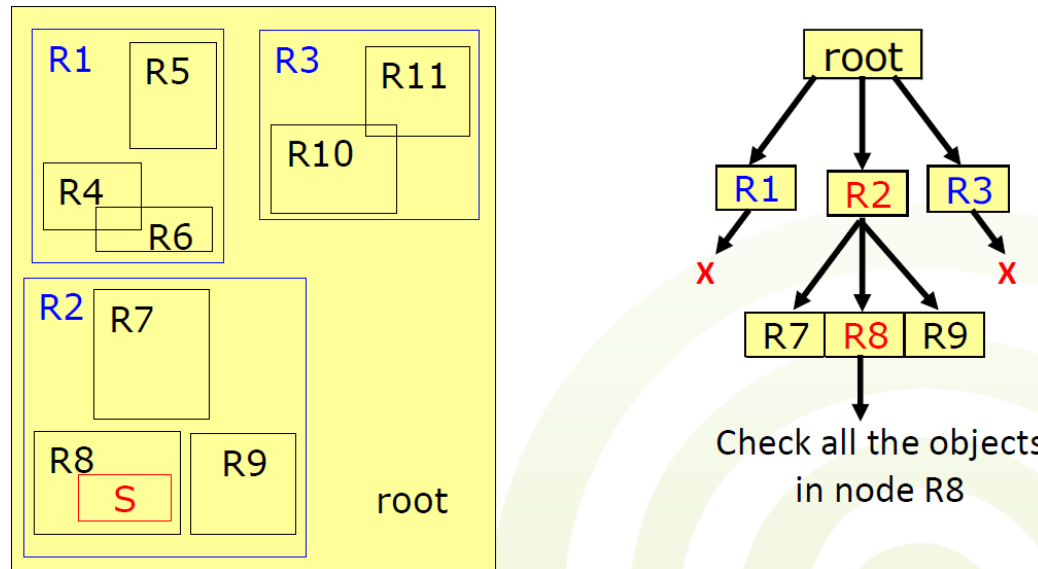
Region Address \odot \longrightarrow \bullet Next Intersection Point



(b)

R-Trees

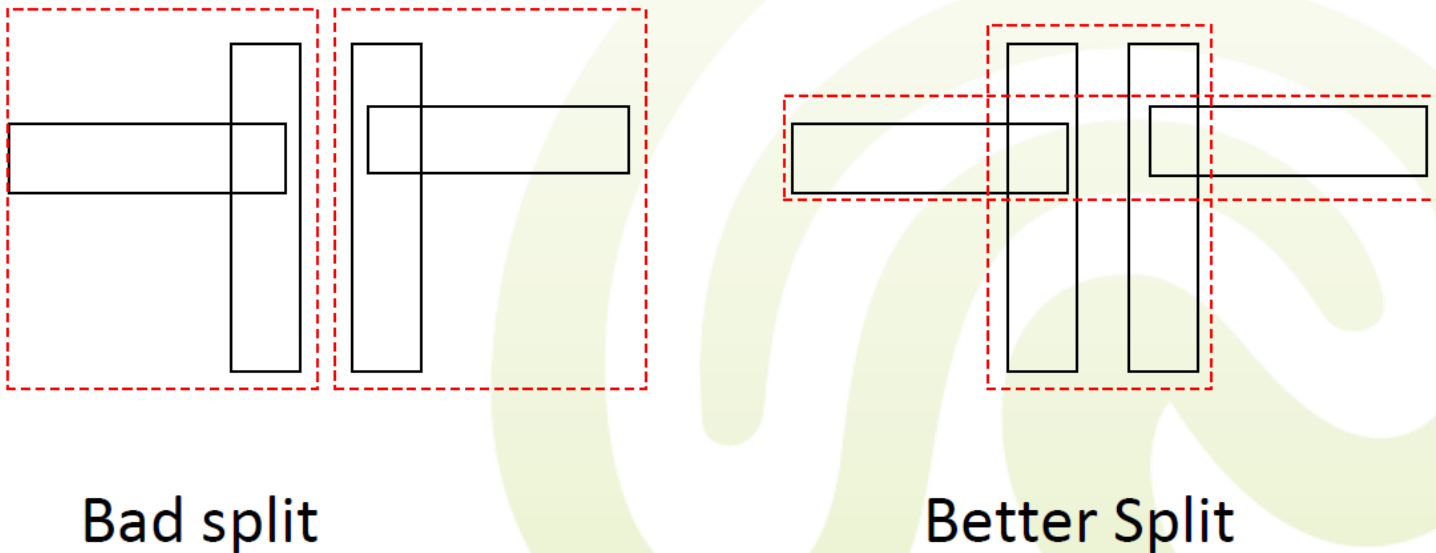
- Like B-trees
 - Data objects stored in leaf nodes
 - Nodes represented by minimum bounding rectangles
 - High-balanced structure



Query S:
7 out of 12 nodes are checked.

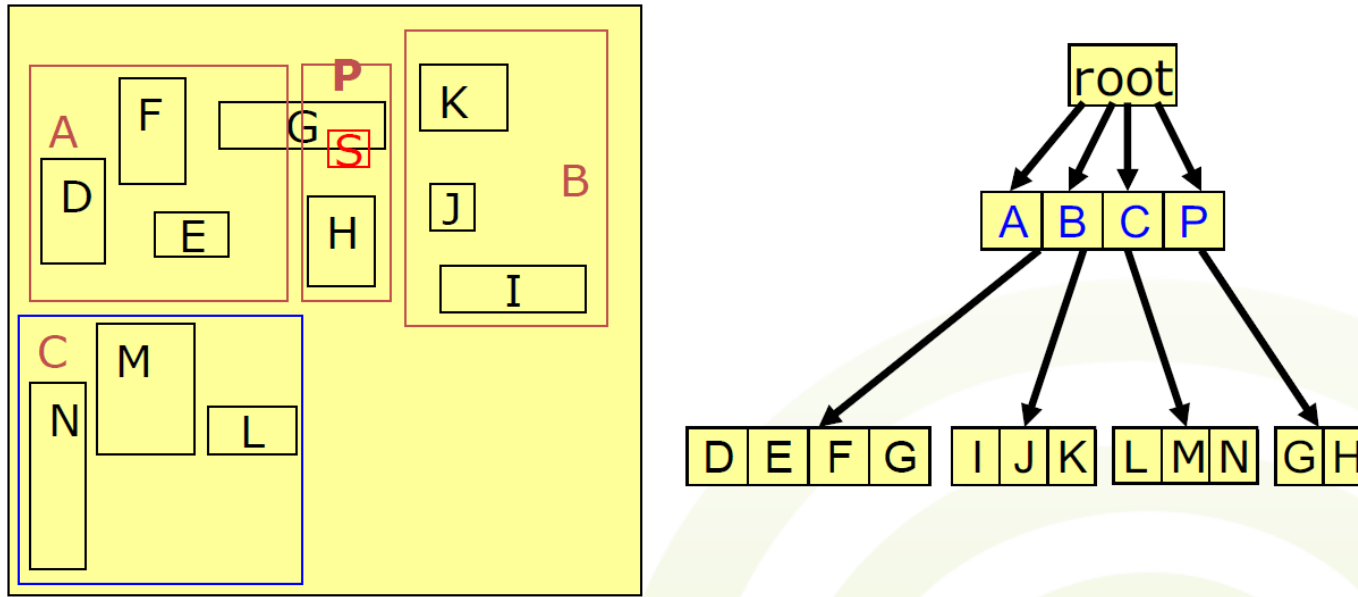
R-Trees Querying

- Many MBR overlaps deteriorate query performance
 - All nodes get visited in the worst case.
- Key is insertion/split optimization
 - Minimize volume by MBR \rightarrow overlaps.



R+ Tree

- Eliminates overlaps by replication of objects in leaves



- Improves performance of point queries

Bitmap Index

- Good for data which has a “small” number of distinct values
 - E.g., gender data, clothing sizes
- Similar performance as B+ tree for read-only data
 - also, when all values are distinct
- A bitmap index for an attribute is a data structure composed of:
 - A collection of bitmaps (bit-vectors)
 - The number of bit-vectors represents the count of distinct values of an attr. in the relation
 - Bitmap (bit vector/array) is an array data structure that stores individual bits
 - Bit signals the presence of value in the row with the relative index of the bit’s position.
 - The length of each bit-vector is the cardinality of the relation.
 - It is compressed by Run-length encoding.

Bitmap Index

- Example

Shop dim

Nr	Shop
1	Saturn
2	Real
3	P&C

Sales fact

Nr	Shop_ID	Sum
1	1	150
2	2	65
3	3	160
4	2	45
5	1	350
6	2	80

Bitmap on Shop of Sales

Value	Bitmap
3	001000
2	010101
1	100010

- Records are allocated permanent numbers.
 - There is a mapping between record numbers and record addresses.
- Deletion
 - in the fact table → tombstones
 - in the index → bit is cleared
- Insertion → bit-vectors are extended, and the new record appended to the table
- Update → toggle the bits in new old bit-vector array and in the new one.

Bitmap Index – Queries

- Combine OR/AND values
 - OR/AND bit ops on vectors
 - E.g., Saturn | P&C

Nr	Shop
1	Saturn
2	Real
3	P&C

Value	Bitmap
3	001000
2	010101
1	100010

```
100010
001000
-----
101010
```

- Combine different indexes on the same table
- Bitmap indexes should be used when the selectivity value is *low*.
 - Combinations of multiple indexes increases selectivity.
- Not very good for range queries on values.
 - → Range-encoded Bitmap Index

Multi-component Bitmap Index

- Encoding using a different numeration system to reduce storage space
 - E.g., <div,mod> classes
- Idea:
 - transform values into more dimensions and project
 - intersection of projections gives the original value
- E.g., the month attribute has values between 0 and 11.
 - Encode by $X = 3*Z+Y$

X	Y			Z			
M	A _{2,-}	A _{1,-}	A _{0,-}	A _{-,3}	A _{-,2}	A _{-,1}	A _{-,0}
5	1	0	0	0	0	1	0

Multi-component Bitmap Index

- If we have 100 (0..99) different days to index we can use a multi-component bitmap index with basis of $\langle 10, 10 \rangle$
- The storage is reduced from 100 to 20 bitmap-vectors
 - 10 for y and 10 for z
- The read-access for a point (1 day out of 100) query needs however 2 read operations instead of just 1

Range-encoded Bitmap Index

- Requires a logical ordering of values
- Idea:
 - set the bit in all bit-vectors of the values following this current one
 - range queries will check just 2 bit-vectors
 - matches are: NOT previous AND current
- Disadvantage:
 - a point query requires reading 2 vectors

Range-encoded Bitmap Index

- Query: Persons born between March and August
 - So, persons which didn't exist in February but existed in August.
 - Just 2 vectors read: ((NOT A1) AND A7)

	Dec	Nov	Oct	Sep	Aug	Jul	Jun	Mai	Apr	Mar	Feb	Jan
Person	A ₁₁	A ₁₀	A ₉	A ₈	A ₇	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀
1	1	1	1	1	1	1	1	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	0	0	0
3	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0

- Normal bitmap would require 6 vectors to read.

Summary of Indexes

- B-Trees are not fit for multidimensional data
 - UB-trees can be applicable
- R-Trees may not scale to many dimensions
- Bitmap indexes are typically only a fraction of the size of the indexed data in the table
- Bitmap indexes reduce response time for large classes of ad hoc queries

Data Partitioning

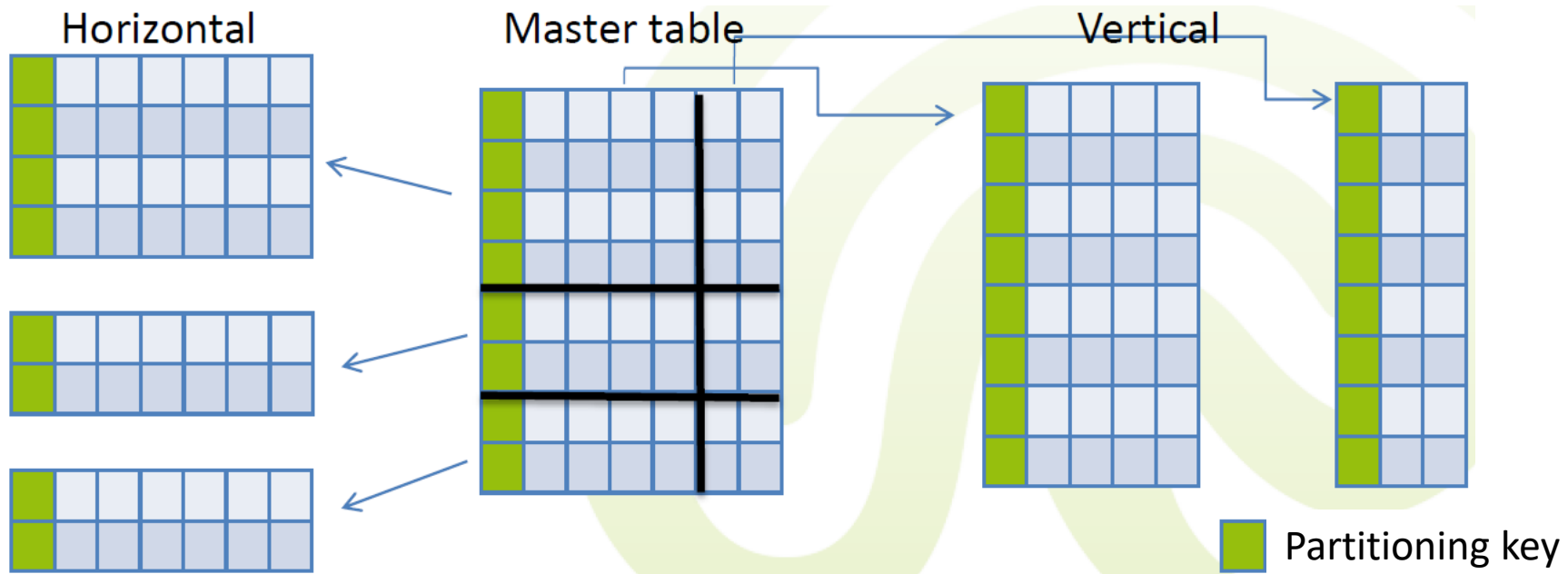
- Breaking data into “non-overlapping” parts
- Horizontal vs. vertical
- May correspond to granularity of a dimension and use ranges to define partitions
- Improves:
 - Business query performance,
 - i.e., minimize the amount of data to scan
 - Data availability,
 - e.g., back-up/restore can run at the partition level
 - Database administration,
 - e.g., archiving data, recreating indexes, loading tables

Data Partitioning

- Approaches:
 - Logical partitioning by
 - Date, Line of business, Geography, Organizational unit, Combinations of these factors, ...
 - Physical partitioning
 - Makes data available to different processing nodes
 - Possible parallelization on multiple disks/machines
- Implementation:
 - Application level
 - Database system

Data Partitioning

- Horizontal – splitting out the rows of a table into multiple tables
- Vertical – splitting out the columns of a table into multiple tables



Horizontal Partitioning

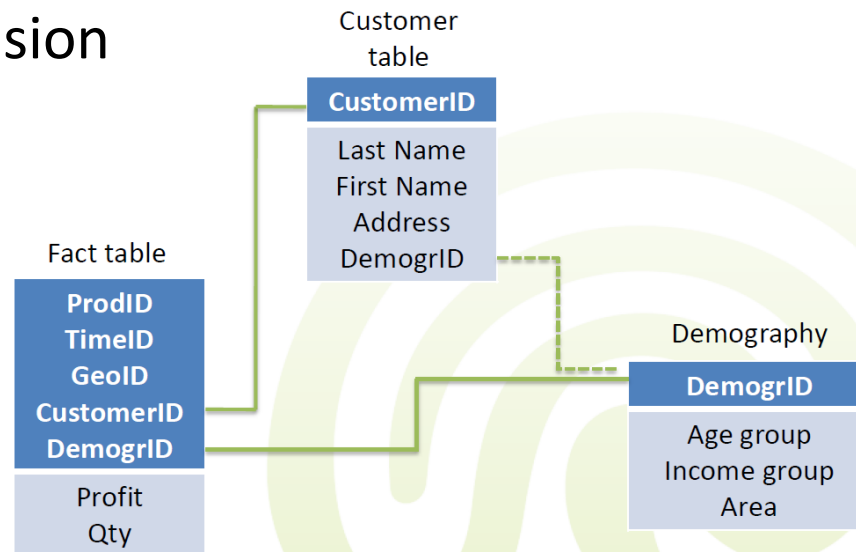
- Distributes records into disjoint tables
- Typically, “view” over union of the table is available
- Types:
 - range – a range of values per table
 - list – enumeration of values per table
 - hash – result of a hash function determines the table
- In DWs typically:
 - Generated reports can identify the partitioning key.
 - Time dimension – weeks, months or age of data
 - Another dim if it does not change often – branch, region
 - Table size – requires some meta-data to constraint the contents

Vertical Partitioning

- Involves creating tables with fewer columns and using additional tables to store the remaining columns
 - Usually called row splitting
 - Row splitting creates one-to-one relationships between the partitions
- Different physical storage might be used
 - E.g., storing infrequently used or very wide columns on a different device
- In DWs typically:
 - move seldom used columns from a highly-used table to another
 - create a view that merges them

Vertical Partitioning

- Mini-dimension with outrigger is a solution
 - Many dimension attributes are used very frequently as browsing constraints
 - In big dimensions these constraints can be hard to find among the lesser used ones
 - Logical groups of often used constraints can be separated into small dimensions
 - which are very well indexed and easily accessible for browsing
- E.g., demography dimension
 - Notice the foreign key in customer



Summary of Partitioning

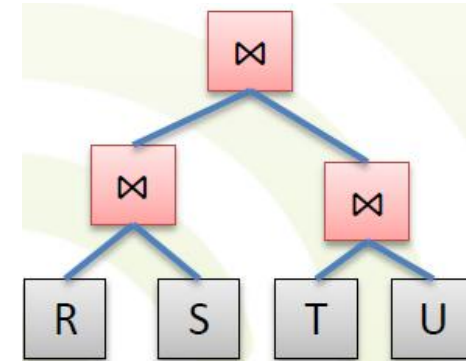
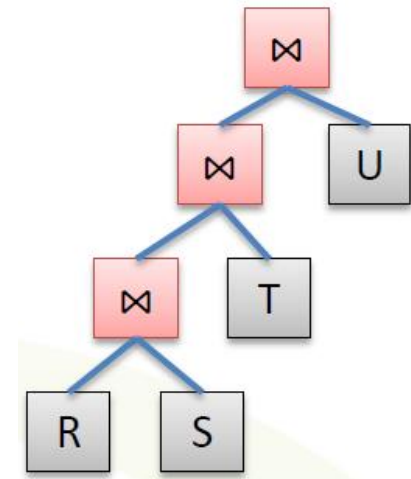
- Advantages
 - Records used together are grouped together
 - Each partition can be optimized for performance
 - Security, recovery
 - Partitions stored on different disks reduces contention
 - Take advantage of parallel processing capability
- Disadvantages
 - Slow retrieval across partitions (expensive joins when vertical partitioning)
 - Complexity
- Recommendations
 - A table is larger than 2GB (from Oracle)
 - A table has more than 100 million rows (practice)

Join Optimization

- Queries over several partitions are often needed
 - This results in joins over the data
 - Though joins are generally expensive operations, the overall cost of the query may strongly differ with the chosen evaluation plan for the joins
- Joins are commutative and associative
 - $R \bowtie S \equiv S \bowtie R$
 - $R \bowtie (S \bowtie T) \equiv (S \bowtie R) \bowtie T$

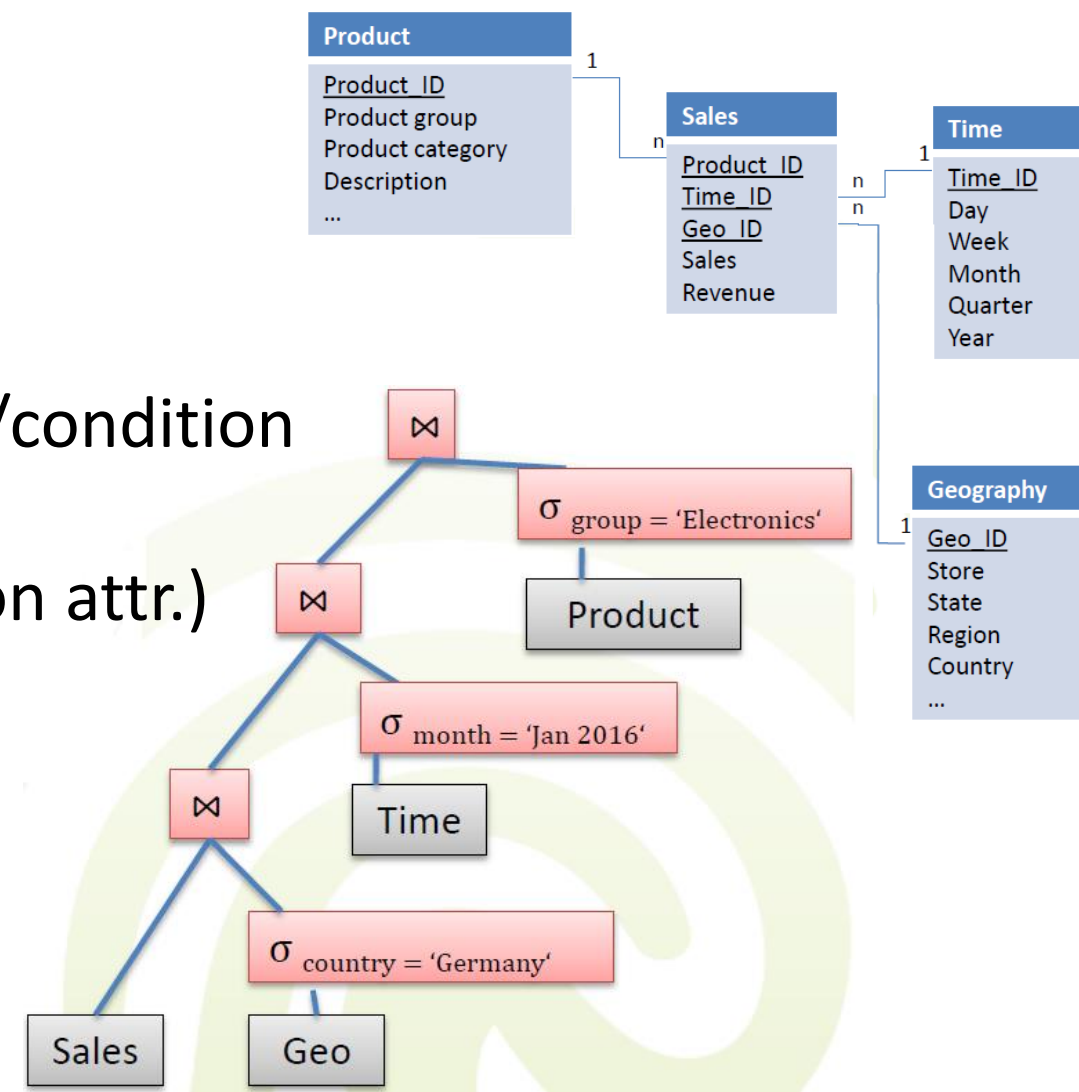
Join Optimization

- This allows evaluating individual joins in any order
 - Results in join trees
 - Different join trees may show very different evaluation performance
 - Join trees have different shapes
 - Within a shape, there are different relation assignments possible
 - Number of possible join trees grows rapidly ($n!$)
- DBMS' optimizer considers
 - statistics to minimize result size
 - all possibilities \rightarrow impossible for large n
 - heuristics to pick promising ones
 - when the number of relations is high (e.g., >6)
 - e.g., genetic algorithms



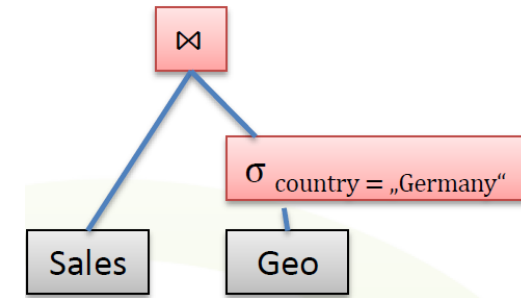
Join Selection Heuristics

- Join relations that relate by an attribute/condition
 - avoiding cross joins
- Minimize the result size (A is the common attr.)
 - $$\frac{T(R)*T(S)}{\max(V(R,A),V(S,A))}$$
- Availability of indexes and selectivity of other conditions
- User tuning
 - Hints in Oracle
 - Set the parameter **join_collapse_limit** in PostgreSQL



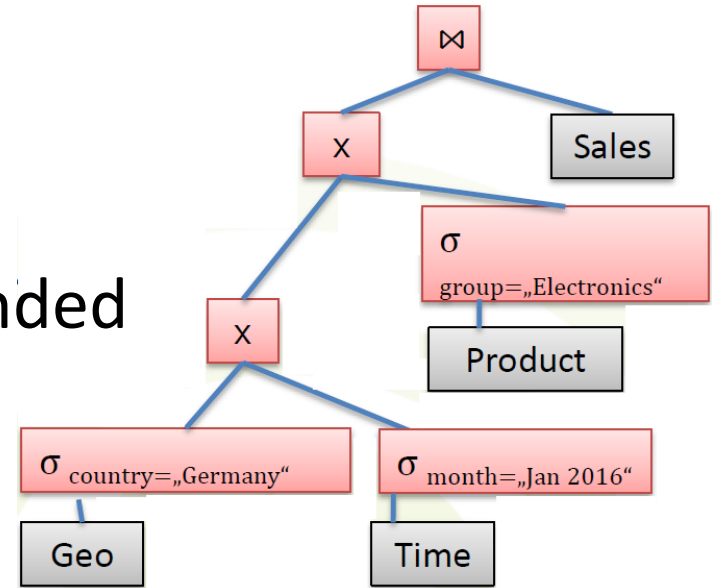
Join Selection Heuristics in DWs

- OLTP's heuristics are not suitable in DWs
 - E.g., join Sales with Geo in the following case:
 - Sales has 10 mil records, in Germany there are 10 stores, in January 2016 there were products sold in 20 days, and the Electronics group has 50 products
 - If 20 % of our sales were performed in Germany,
 - the selectivity value is high.
 - so, an index would not help that much
 - The intermediate result would still comprise 2 mil records
- Cross join is recommended

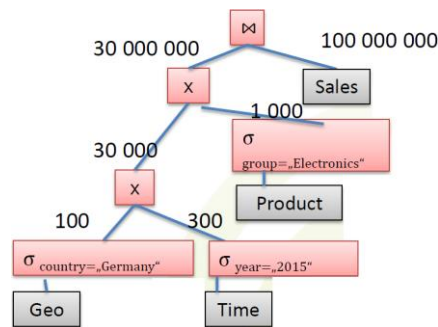


Join Selection Heuristics in DWs

- The **cross join** of the dimension tables is recommended
 - Geo dimension – 10 stores in Germany
 - Time dimension – 20 days in Jan 2016
 - Product dimension – 50 products in Electronics
 - 10m facts in Sales
 - $10 * 20 * 50 = 10,000$ records after performing the cross product



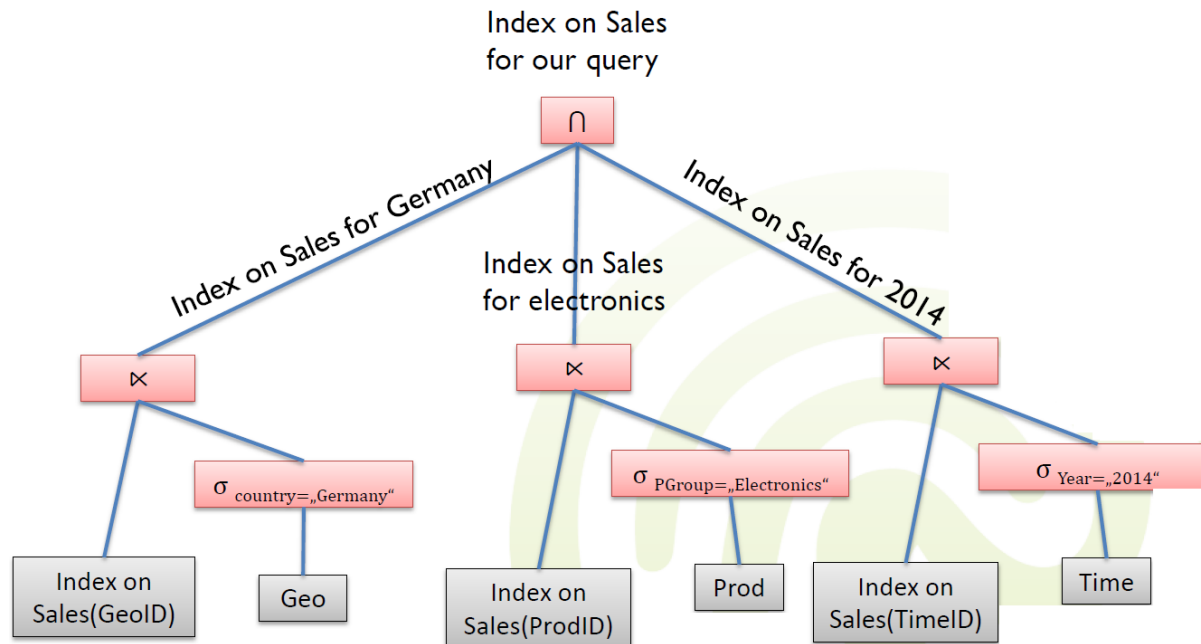
- But can also be expensive!



- Allows
 - a single pass over the Sales
 - using an index on the most selective attribute yet

Join Selection Heuristics in DWs

- If cross join is too large, **intersect partial joins**
 - applicable when all dimension FKs are indexed
 - in fact, it is a **semi-join** (no record duplication can take place)



Summary of Joins

- Prefer a cross-join on dimensions first
 - If not all dimension FKs are indexed
- Intersect semi-joins otherwise

- Avoid standard DBMS's plans
 - But check the plan first 😊

Materialized Views

- Views whose tuples are stored in the database are said to be materialized
- They provide fast access, like a (very high-level) cache
- Need to maintain the view's contents as the underlying tables change
 - Ideally, we want incremental view maintenance algorithms

Materialized Views

- How can we use MV in DW?
 - E.g., we have queries requiring us to join the Sales table with another table and aggregate the result
 - `SELECT P.Categ, SUM(S.Qty) FROM Product P, Sales S WHERE P.ProdID=S.ProdID GROUP BY P.Categ`
 - `SELECT G.Store, SUM(S.Qty) FROM Geo G, Sales S WHERE G.GeoID=S.GeoID GROUP BY G.Store`
 - ...
 - There are more solutions to speed up such queries
 - Pre-compute the two joins involved (product with sales and geo with sales)
 - Pre-compute each query in its entirety
 - Or use a common and already materialized view

Materialized Views

- Having the following view materialized
 - CREATE MATERIALIZED VIEW Totalsales(ProdID, GeoID, total) AS
SELECT S.ProdID, S.GeoID, SUM(S.Qty) FROM Sales S
GROUP BY S.ProdID, S.GeoID
- We can use it in our queries
 - SELECT P.Categ, SUM(T.Total) FROM Product P, Totalsales T
WHERE P.ProdID=T.ProdID GROUP BY P.Categ
 - SELECT G.Store, SUM(T.Total) FROM Geo G, Totalsales T
WHERE G.GeoID=T.GeoID GROUP BY G.Store

Materialized Views

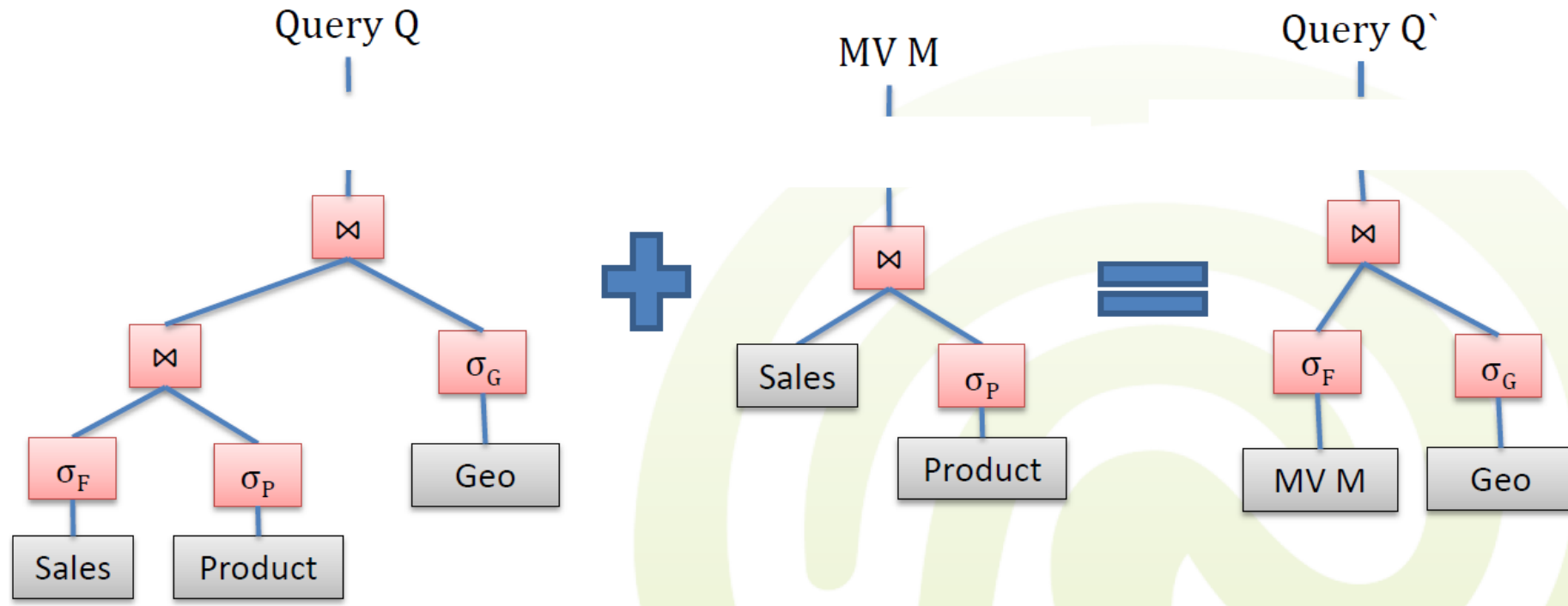
- MV issues
 - Choice of materialized views
 - What views should we materialize, and what indexes should we build on the pre-computed results?
 - Utilization
 - Given a query and a set of materialized views, can we use the materialized views to answer the query?
 - Maintenance
 - How frequently should we refresh materialized views to make them consistent with the underlying tables?
 - And how can we do this incrementally?

Materialized Views: Utilization

- Utilization must be transparent
 - Queries are internally rewritten to use the available MVs by the query rewriter
 - The query rewriter performs integration of the MV based on the query execution graph

Materialized Views: Utilization

- E.g., mono-block query (perfect match)

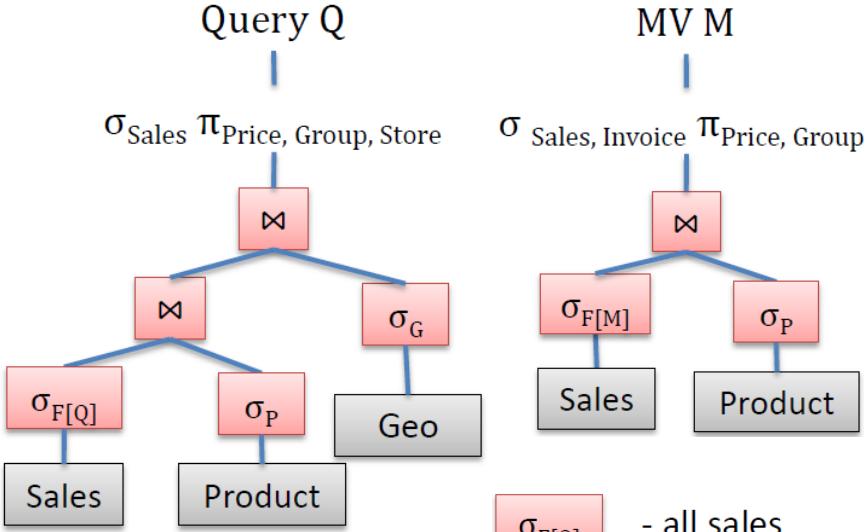


Materialized Views: Integration

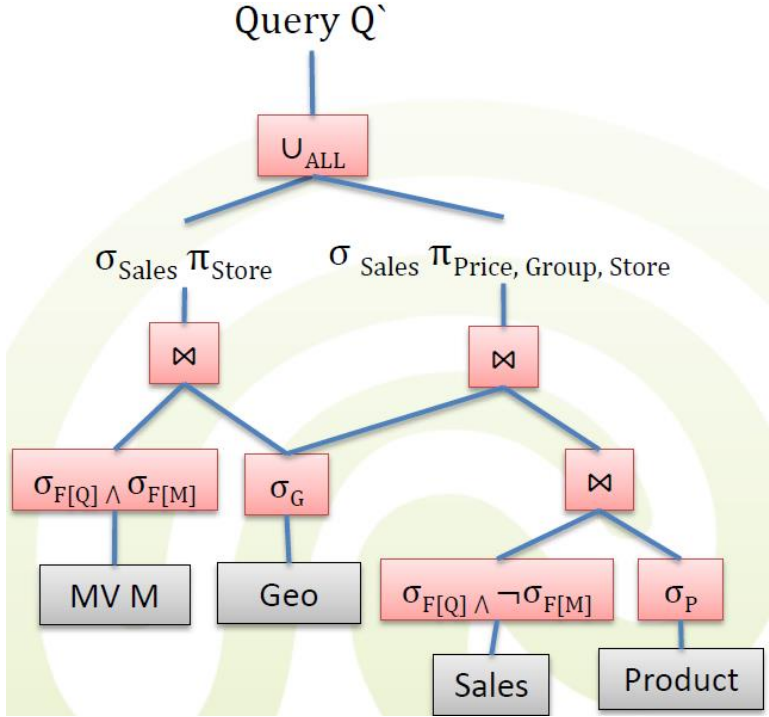
- Correctness:
 - A query Q' represents a valid replacement of query Q by utilizing the materialized view M , if Q and Q' always deliver the same result.
- Implementation requires the following conditions:
 - The selection condition in M cannot be more restrictive than the one in Q .
 - The projection from Q must be a subset of the projection from M .
 - It must be possible to derive the aggregation functions in Q from ones in M .
 - Additional selection conditions in Q must be possible also on M .

Materialized Views: Integration

- A way to integrate a more restrictive view:
 - Split the query Q in two parts, Q_a and Q_b , such that
 - $\sigma(Q_a) = (\sigma(Q) \wedge \sigma(M))$ and
 - $\sigma(Q_b) = (\sigma(Q) \wedge \neg\sigma(M))$

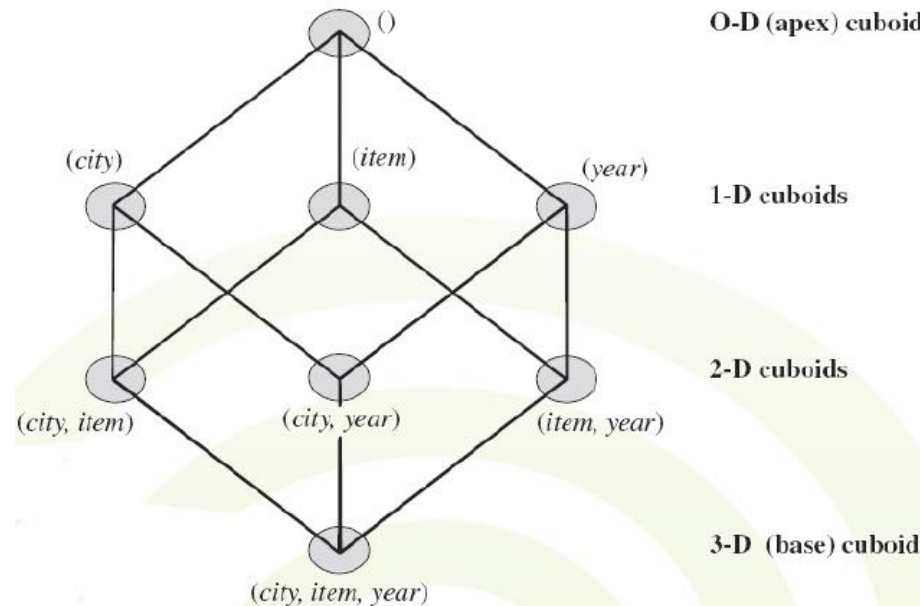


$\sigma_{F[Q]}$ - all sales
 $\sigma_{F[M]}$ - More restrictive:
 all sales above a threshold



Materialized Views & DWs

- Often store aggregated results
- For a set of “n” group-by attributes, there are 2^n possible combinations
 - Too many to materialize all
 - What to materialize?



Materialized Views & DWs

- Choosing the views to materialize
 - Static choice:
 - The choice is performed at a certain time point
 - by the DB administrator (not very often) or by an algorithm
 - The set of MVs remains unmodified until the next refresh
 - The chosen MVs correspond to older queries
 - Dynamical choice:
 - The MV set adapts itself according to new queries

Views to Materialize

- Static choice
 - Choose which views to materialize, in concordance with the “benefit” they bring
 - The benefit is computed based on a cost function
 - The *cost function* involves
 - Query costs
 - Statistical approximations of the frequency of the query
 - Actualization/maintenance costs
 - Classical knapsack problem – a limit on MV storage and the cost of each MV
 - Greedy algorithm
 - Input: the lattice of cuboids, the expected cardinality of each node, and the maximum storage size available to save MVs
 - It calculates the nodes from the lattice which bring the highest benefit according to the cost function, until there is no more space to store MVs
 - Output: the list of lattice nodes to be materialized

Views to Materialize

- Disadvantages of static choice
 - OLAP applications are interactive
 - Usually, the user runs a series of queries to explain a behavior he has observed, which happened for the first time
 - So now the query set comprises hard to predict, ad-hoc queries
 - Even if the query pattern is observed after a while, it is unknown for how much time the pattern will remain valid
 - Queries are always changing
 - Often modification to the data leads to high update effort
- There are, however, also for OLAP applications, some often repeating queries that should in any case be statically materialized

Views to Materialize

- Dynamic choice
 - Monitor the queries being executed over time
 - Maintain a materialized view processing plan (MVPP) by incorporating most frequently executed queries
 - Modify MVPP incrementally by executing MVPP generation algorithm
 - as a background process
 - Decide on the views to be materialized
 - Reorganize the existing views
- It works on the same principle as caching, but with semantic knowledge

Views to Materialize

- Dynamic choice
 - Updates of cached MV:
 - In each step, the cost of MV in the cache as well as of the query is calculated
 - All MVs as well as the query result are sorted according to their costs
 - The cache is then filled with MV in the order of their costs, from high to low
 - This way it can happen that one or more old MVs are replaced with the current query
 - Factors consider in the *cost function*:
 - Time of the last access
 - Frequency of query
 - Size of the materialized view
 - The costs a new calculation or actualization would produce for a MV
 - Number of queries which were answered with the MV
 - Number of queries which could be answered with this MV

Maintenance of Materialized Views

- Keeping a materialized view up-to-date with the underlying data
 - How do we refresh a view when an underlying table is refreshed?
 - When should we refresh a view in response to a change in the underlying table?
- Approaches:
 - Re-computation – re-calculated from the scratch
 - Incremental – updated by new data, not easy to implement
- Immediate – as part of the transaction that modifies the underlying data tables
 - Advantage: materialized view is always consistent
 - Disadvantage: updates are slowed down
- Deferred – some time later, in a separate transaction
 - Advantage: can scale to maintain many views without slowing updates
 - Disadvantage: view briefly becomes inconsistent

Maintenance of Materialized Views

- Incremental maintenance
 - Changes to database relations are used to compute changes to the materialized view, which is then updated
 - Considering that we have a materialized view V , and that the basis relations suffer modifications through inserts, updates or deletes, we can calculate V' as follows
 - $V' = (V - \Delta^-) \cup \Delta^+$, where Δ^- and Δ^+ represent deleted and inserted tuples, respectively

Maintenance of Materialized Views

- Deferred update options:
 - Lazy
 - delay refresh until next query on view, then refresh before answering the query
 - Periodic (Snapshot)
 - refresh periodically – queries are possibly answered using outdated version of view tuples
 - widely used in DWs
 - Event-based
 - e.g., refresh after a fixed number of updates to underlying data tables

Summary

- The term **selectivity** and its interpretation.
- Bitmap indexes are universal, space efficient
- R*-trees, X-trees for multidimensional data
- Partitioning
 - Records used together should be stored together
 - Mini-dimension
- Joins
 - Computing cross join on dimension table is an option
- Materialized views can replace parts of a query
 - Select what to materialize (not everything) statically or dynamically