6.1 Multidimensional Storage

• The basic storage structure is the multidimensional array
  – Customized based upon
    • The data e.g., sparse or dense
    • Characteristics of the secondary memory e.g., block- or page-oriented
  – Cube data cells are stored sequentially
    • Multidimensional cubes are linearized

6.1 Multidimensional Storage

• General idea of linearization
  – Considering a cube \( C = (D_1, D_2, ..., D_n) \), \((M_1:Type_1, M_2:Type_2, ..., M_n:Type_n)\), the index of a cube cell \( z \) with coordinates \((x_1, x_2, ..., x_n)\) can be linearized as follows:

\[
\text{Index}(z) = x_1 + (x_2 - 1) \cdot |D_1| + (x_3 - 1) \cdot |D_1| \cdot |D_2| + ... + (x_n - 1) \cdot |D_1| \cdot |D_2| \cdot ... \cdot |D_{n-1}| = 1 + \sum_{i=1}^{n} \left( (x_i - 1) \cdot \prod_{j=1}^{i-1} |D_j| \right)
\]

6.1 Problems in Array-Storage

• Problems in array-storage
  – Influence of the order of the dimensions in the cube definition
    • In the cube the cells of \( D_i \) are ordered one under the other, e.g., Pants
    A query of sales of all pants involves a column in the cube
  • After linearization, the information is spread among more data blocks or pages
  – If we consider a data block can hold 5 cells, a query over all products sold in January can be answered with just 1 block read, but a query of all sold pants, involves reading 4 blocks
6.1 Problems in Array-Storage

- The problem of dimensions order can be diminished by using caching solutions
  - Caching and swapping is performed also by the operating system
  - MDBMS has to manage its caches such that the OS doesn’t perform any damaging swaps

6.1 Problems in Array-Storage

- Storage of dense cubes
  - If cubes are dense, array storage is more efficient. However, operations suffer due to the large cubes
  - The solution is to store dense cubes not linear but on 2 levels
    - The first contains an indexes and the second the data cells stored in blocks
    - Different optimization procedures like indexes (trees, bitmaps), physical partitioning, and compression (run-length-encoding) can be used

6.1 Problems in Array-Storage

- Storage of sparse cubes
  - All the cells of a cube, including empty ones, have to be stored
  - Sparseness leads to data being stored in many physical blocks or pages
    - The query speed is affected by the large number of block accesses on the secondary memory
  - Solutions:
    - Do not store empty blocks or pages: if there are large empty portions of the array, they will not be physically stored, but the index structure will be adapted
    - 2 level data structure: upper layer holds all possible combinations of the sparse dimensions, lower layer holds dense dimensions

6.2 Indexes

- Indexes are used to optimize queries. OLAP queries have an aggregation role
  - How many articles from product group washing devices were sold in 2008 for each month in each region
    - Very big detailed data set (lots of sales in the sales fact table)
    - Such aggregation (2008, region) queries on big data sets are costly; e.g., consider 100 GB of sales data stored in a star schema; for this query the whole set needs to be read…at an average speed of 40 MB/s it still takes 43 minutes only to read the data
6.2 Remember OLAP Queries?

- Partial range query
  - Some dimensions are not restricted
  - Geometrically described as a sub-space
- Partial match query
  - Restricts more dimensions on a point, while other dimensions remain unspecified
  - Geometrically described as a hyper-level in the data set
- Point query
  - Restricted to a point on all dimensions

6.2 Optimization Procedures

- Base-layer of the data
- Layer of materialization
- Layer of partitioning
- Layer of Index struct.
- MV1
- MV2
- P1
- P2
- R*-Baum
- RB-Baum
- UB-Baum
- Physical access paths
- Logical access paths

6.2 Index Structures

- Why index?
  - Consider a 10 GB table; at 10 MB/s read speed we need 17 minutes for a full table scan
  - Consider an OLAP query: the number of Bosch S500 washing machines sold in Braunschweig last month?
    - Applying restrictions (product, location) the selectivity would be strongly reduced
      - If we have 30 location, 1000 products and 24 months in the DW, the selectivity is:
        \[ \frac{1}{30} \times \frac{1}{1000} \times \frac{1}{24} = 0.00000014 \]
    - So... we read 10 GB for 1,4KB of data ...not very smart

- Types of queries
  - k-nearest-neighbor-Search (k-NN-Search)
    - Find the first K objects with the smallest distance to the query object
    - Such a query is usually performed with approximations (with an error rate)
  - Reverse-nearest-neighbor-Search
    - Find all the objects whose nearest neighbor is the queried object

- Curse of Dimensionality (Richard Bellman)
  - The volume increases exponentially with the dimensions number
  - More dimensions mean more comparisons will be performed
  - At the present time there is no real scalable indexing
6.2 Index Structures

- Classification criteria for index structures
  - Clustering: Data with high probability of being used together
  - Dimensionality: Refers to the number of attributes used to calculate the index key
  - Symmetry: The order of the index attribute is not performance relevant
  - Tuple identifier (TID): TIDs are position numbers pointing to the physical storage place of the corresponding data

- Dynamical behavior: Effort needed for dynamical modifications can strongly vary from index to index

6.2 Trees

- In the beginning...there were B-Trees: Data structures for storing sorted data with amortized run times for insertion and deletion
  - Basic structure of a node

6.2 B-Trees

- What's wrong with B-Trees?
  - K-D-B trees are useful for point data only
  - Exact-point lookup!
  - Not good for storing geometrical data and multi-dimensional data
  - The R-Tree provided a way to do that (thanks to Guttman '84)

- R-Trees are recommended for lower dimensionality
  - Up to 10 dimensions
  - More scalable variants:
    - R+-Trees, R*-Trees and X-Trees
    - Each up to 20 dimensions
6.2 R-Trees

- There is no total ordering of objects in the multidimensional space that preserves spatial proximity
  - E.g., in time dimension it makes sense to keep data belonging to consecutive quarters clustered together

- R-Trees
  - Can organize any-dimensional data
    - Representing the data by a minimum bounding rectangle (MBR)
    - Each node bounds its children
    - A node can have many objects in it
      - E.g., node capacity of 3 (in practice node capacity is of 100s)
      - 2 dimensions

- The leaves point to the actual objects (stored on disk probably)
- The height is always log n (it is height balanced)

- Leaf nodes, contain entries of the form (I, RID)
  - I same as for non-leaf
  - RID represents a unique tuple identifier

- Operations
  - Let:
    - E be the rectangle part of an index entry E
    - E_p be the tuple-identifier or pointer
    - S be the search rectangle
      - E.g., ([08 Qtr1, 09 Qtr1], [a, b])
    - T be the root of the R-Tree
  - Search:
    - Start from the root node
    - If multiple sub-trees contain the point of interest then follow all
6.2 R-Trees - Search

- Search \((T,S)\)
  - If \(T\) is not a leaf
    - Check each entry \(E\) to determine whether \(E\) overlaps \(S\)
    - For all overlapping entries, invoke Search\((E,S)\)
  - If \(T\) is a leaf
    - Check all entries \(E\) to determine whether \(E\) overlaps \(S\)
      - If \(E\) is a qualifying record
  - No good performance guarantees
  - In worst case all paths must be searched (due to overlapping)
  - Search algorithms try to cut out irrelevant regions ("Pruning")

- Insert, general idea
  - New index records are added to the leaves!
    - Nodes that overflow are split
    - Splits propagate up the tree
    - Node splitting is not trivial

- ChooseLeaf(\(E\))
  - Initialize
    - Set \(N\) to be the root node
  - Leaf check
    - If \(N\) is a leaf, return \(N\)
  - Choose sub-tree
    - Let \(F\) be the entry in \(N\) whose rectangle \(F_1\) needs least enlargement to include \(E\)
    - Resolve ties by choosing the entry with the rectangle of smallest area
    - Descend until a leaf is reached
      - Set \(N\) to be the child node pointed to by \(F_p\) and repeat from Leaf check

- Insert (\(T,E\))
  - Find position for new record
    - Invoke ChooseLeaf to select a leaf node \(L\) in which to place \(E\)
  - Add record to leaf node:
    - If \(L\) has room for \(E\) then insert \(E\) and return
    - Otherwise, invoke SplitNode to obtain \(L\) and \(LL\) containing \(E\) and all the old entries of \(L\)
  - Propagate changes upwards
    - Invoke AdjustTree on \(L\), also passing \(LL\) if a split was performed
  - Grow tree taller
    - If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

- Node splitting
  - A full node contains \(M\) entries
  - Divide the collection of \(M+1\) entries between 2 nodes.
  - Objective: Make it as unlikely as possible for the resulting two new nodes to be examined on subsequent searches.
  - Heuristic: The total area of two covering rectangles after a split should be minimized
6.2 R-Trees - Insert

- Node splitting
  - Bad split
  - Good split

6.2 R-Trees - Insert

- Node splitting methods
  - Exhaustive algorithm
    - Generate all possible groups and choose the best with minimum area
    - Number of possibilities ~ \(2^{M-1}\)
    - For \(M \approx 50\) Number of possibilities ~ 600 Trillion
      - Which is even more than the Obama administration spent with the crisis!!

- Quadratic-cost algorithm
  - A heuristic to find a small-area split
  - Cost is quadratic in \(M\) and linear in the number of dimensions
  - Pick two of the \(M+1\) entries to be the first elements of the two new groups
    - Calculate the MBR for each pair, and choose the one with the largest MBR
    - These 2 objects are the new starting points for the resulting 2 nodes

- Linear cost algorithm
  - Identical to Quadratic with the following differences:
    - Uses a linear procedure to identify the starting entries
      - Find in each dimension 2 rectangles
        - the rectangle with the highest minimum coordinates
        - and the rectangle with the lowest maximum coordinates
    - Order the next entries so that the volume growth is the smallest from one step to another
6.2 R-Trees - Insert

- Quadratic vs. Linear cost algorithm
  - Quadratic:
    - Choose two objects that create as much empty space as possible
  - Linear:
    - Choose two objects that are furthest apart
    - Linear node-split is simple, fast, and as good as quadratic!
    - Quality of the splits is slightly worse!

6.2 R-Trees - Delete

- Delete entry
  - Start a normal search of the entry to delete (FindLeaf)
  - Delete the record from the leaf (DeleteRecord)
  - Condense Tree if needed (if there are now nodes which only have few entries)
    - At condensation the node to be condensed is deleted as a whole and the entries which should remain are then inserted
  - If the root has just one child, it will be the new root

6.2 R-Trees - Update

- Update
  - If the datasets are updated the existent rectangles can be changed
  - In this case the index entry must be deleted updated and inserted

6.2 R-Trees

- Every non-leaf node has between m and M children
  - Root node is the exception
  - For each entry (I, P) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node
  - The root node has at least two children unless it is a leaf
  - All leaves appear on the same level
  - Height of a tree = \( \text{ceiling}(\log_m N) - 1 \)
  - Worst case utilization for all nodes except the root is \( m/M \)

6.2 R-Trees

- Advantages
  - Efficient for non-point queries
  - No downward cascading splits
  - Guaranteed utilization

- Disadvantages
  - Dimension dependent fan-out
  - Overlapping regions - search performance problem
6.2 R-Tree Variations

- **R+-Trees** enhances retrieval performance by avoiding visiting multiple paths when searching for point queries.
  - No overlap for MBRs at the same level (internal nodes)
  - Specific object's entry might be duplicated
  - Insertions might lead to a series of update operations in a chain-reaction.

- Compared to R-trees,
  - Nodes are not guaranteed to be at least half filled
  - The entries of any internal node do not overlap
  - An object ID may be stored in more than one leaf node

- Advantages
  - Because nodes are not overlapped with each other, point query performance benefits, e.g., a single path is followed and fewer nodes are visited than with the R-tree
  - Disadvantages
    - Since rectangles are duplicated, an R+-Tree can be larger than an R-Tree built on same data set
    - Construction and maintenance of R+ trees is more complex than the construction and maintenance of R-Trees

- R'-Trees
  - Advantages
    - Node split is more sophisticated
      - When a node overflows, p entries are extracted and reinserted in the tree (p might be 25%)
  - Considers minimization of:
    - Overlapping between minimum bounding rectangles at the same level
    - Perimeter of the produced minimum bounding rectangles
    - Insertion is more expensive while retrievals are faster

6.2 UB-Trees

- Combination of B'-Tree and Z-curve = Universal B-Tree (UB-tree)
  - Z-curve is used to map multidimensional points to one-dimensional values (Z-values)
  - Z-values are used as keys in B'-Tree

- Concept of Z-Regions
  - To create a disjunctive partitioning of the multidimensional space
  - This allows for very efficient processing of multidimensional range queries
6.2 UB-Trees

- **Z-Regions**
  - The space covered by an interval on the Z-Curve
  - Defined by two Z-Addresses a and b
    - We call b the region address of [a : b]
  - Each Z-Region maps exactly onto one page on secondary storage
  - I.e., to one leaf page of the B*-Tree
  - E.g., of Z-Regions
    - [1:9], [10, 18], [19, 28]...

- **Z-Value address representation**
  - Calculated through bit interleaving of the coordinates of the tuple
    - $Y = 5 = 101$  $X = 4 = 100$
    - Z-value $= 110010$

- **Why Z-Values?**
  - With Z-Values we reduce the dimensionality of the data to one dimension
  - Z-Values are then used as keys in B*-trees
    - Using B*-Trees results in high node filling degree (at least 50%)
    - Logarithmical complexity at search, insert and delete
      - Guaranteed maximum node accesses to locate a key is $\log_2 \left( \frac{b}{a} \right)$
  - Z-Values are very important for range queries!

- **Range queries (RQ) in UB-Trees**
  2. The corresponding page is loaded and filtered with the query predicate
    1. Tuples 15 and 16 fulfill the predicate
  3. The next region (inside the query rectangle) on the Z-curve is calculated
    1. The next jump point on the Z-curve is 27
  4. Repeat steps 2 and 3 until the end-address of the last filtered region is bigger than $q_b$

- **The critical part of the algorithm** is calculating the jump point on the Z-curve which is inside the query rectangle
  - If this takes too long it eliminates the advantage obtained through optimized disk access
  - How is the jump point optimally calculated?
    - From 3 points: $q_a, q_b$ and the current Z-Region
    - By performing bit operations
6.2 UB-Trees

- Range Queries, UB-Trees and DW
  - In DW we have hierarchical organization of dimensions
  - No intervals for hierarchical restrictions
  - Naïve restrictions lead to many point queries instead of one interval on UB-Tree
  - This is why we need Multidimensional Hierarchical Clustering (MHC)

6.2 MHC

- With MHC UB-Trees can:
  - Artificial encode hierarchies:
    - Mapping of hierarchy restrictions to range restrictions
    - Mapping is used for physical clustering of the fact table
  - Increase computation and space efficiency
  - However, modification of query algorithms is necessary

6.2 Bitmap Indexes

- Bitmap Indexes
  - Lets assume a relation Expenses with three attributes: Nr, Shop and Sum
  - A bitmap index for attribute Shop looks like this

<table>
<thead>
<tr>
<th>Nr</th>
<th>Shop</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Saturn</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>Real</td>
<td>65</td>
</tr>
<tr>
<td>3</td>
<td>P&amp;C</td>
<td>160</td>
</tr>
<tr>
<td>4</td>
<td>Real</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>Saturn</td>
<td>350</td>
</tr>
<tr>
<td>6</td>
<td>Real</td>
<td>80</td>
</tr>
</tbody>
</table>

6.2 MHC

- Mapping of hierarchy restrictions to range restrictions
- Mapping is used for physical clustering of the fact table

6.2 Bitmap Indexes

- A bitmap index for an attribute of relation is:
  - A collection of bit-vectors
  - The number of bit-vectors represents the number of distinct values of the attribute in the relation
  - The length of each bit-vector is called the cardinality of the relation
  - The bit-vector for value v has 1 in position i, if the i^{th} record has v in attribute A, and it has 0 otherwise
- Records are allocated permanent numbers
- There is a mapping between record numbers and record addresses
6.2 Bitmap Indexes

**Advantages**
- Very efficient when used for partial match queries
- They offer the advantage of buckets
  - In our example each index vector is a bucket
    - E.g., the Saturn bitmap vector is a bucket of 2, telling us that records having value Saturn in attribute Shop are first and 5th record in the table
- They can also help answer range queries
- Efficient hardware support for bitmap operations (AND, OR, XOR, NOT)

---

**Handling modification**
- Assume record numbers are not changed
- Deletion
  - Tombstone replaces deleted record
  - Corresponding bit is set to 0
  - E.g., delete the 5th record

---

**Modification**
- Change the bit corresponding to the old value of the modified record to 0
- Change the bit corresponding to the new value of the modified record to 1
- If the new value is a new value of A, then insert a new bit-vector: e.g., replace Shop for record 2 to REWE

---

**Select**
- Basic AND, OR bit operations:
  - E.g., select the sums we have spent in Saturn and P&C

---

**Insertion record is assigned the next record number**
- A bit of value 0 or 1 is appended to each bit vector
- If new record contains a new value of the attribute, add one bit-vector
  - E.g., insert new record with REWE as shop

---

**Select**
- Bitmap indexes should be used when selectivity is high
6.2 Bitmap indexes

• Advantages
  – Operations are efficient and easy to implement (directly supported by hardware)

• Disadvantages
  – For each new value of an attribute a new bitmap-vector is introduced
    • If we bitmap index an attribute like birthday (only day) we have 365 vectors: $365/8$ bits $\approx 46$ Bytes for a record, just for that
  – Solution to such problems is multi-component bitmaps
  – Not fit for range queries where many bitmap vectors have to be read
    • Solution: range-encoded bitmap indexes

6.2 Multi-component bitmap indexes

• Advantage of multi-component bitmap indexes
  – If we have 100 (0..99) different Days to index we can use a multi-component bitmap index with basis of $<10,10>$
  – 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

6.2 Range-encoded bitmap indexes

• If the query is limited only on one side, (e.g., persons born in or after March), 1 vector is enough (NOT $A_i$)
• For point queries, 2 vector reads are however necessary!
  – E.g., persons born in March: ((NOT $A_7$) AND $A_2$)

6.2 Bitmap indexes flavors

• Combine multi-component bitmap indexes with range-encoding bitmap indexes and we have multi-component-range-encoding bitmap indexes
• Interval-encoded bitmap indexes
  – Each bitmap-vector represents an interval
  – It also needs to read at most 2 vectors, but the storage is half, compared to range-encoded bitmap indexes
6.2 Grid Files

- **Partitions** the range of key values for each key into several buckets

- Dynamic structure using a **grid directory**
  - **Grid array**: a 2 dimensional array with pointers to buckets (this array can be large, disk resident) \( G(0, \ldots, n_x, \ldots, n_z) \)
  - **Linear scales**: two 1 dimensional arrays that used to access the grid array (main memory) \( X(0, \ldots, n_x), Y(0, \ldots, n_y) \)

- **Properties**
  - Supports multi-dimensional data, but **not high number** of dimension
  - Every key is treated as primary key
  - The index structure adapts itself dynamically to maintain storage efficiency
  - Guarantee two disk accesses for point queries
  - Values of key must be in linearly-ordered domain

- **Range Queries**
  - E.g., Find \( 5 < X < 9 \) AND “Mat” < “Robot”

- **Exact Match Search**: at most 2 I/Os assuming linear scales fit in memory
  - First use linear scales to determine the index into the cell directory
  - Access the cell directory to retrieve the bucket address (may cause 1 I/O if cell directory does not fit in memory)
  - Access the appropriate bucket (1 I/O)

- **Insert**
  - Determine the bucket into which insertion must occur
    - If space in bucket, insert
    - Else, split bucket
    - If bucket split causes a cell directory to split do so and adjust linear scales
  - Insertion of these **new entries** potentially requires a complete reorganization of the cell directory… **expensive!!!**
6.2 Grid Files: Insert

- a)
- b)
- c)
- d)
- e)
- f)

6.2 Grid Files: Delete

- Delete the data node, and
  - Merge data pages/blocks if possible
  - Merge directory pages if possible

Next lecture

- Optimization
  - Partitioning
  - Joins
  - Materialized Views