3. Data Storage and Index Structures

640 K ought to be enough for anybody. Bill Gates

One important property of DBMSs is their ability to handle large amounts of data. In order to store and retrieve this data efficiently different techiques are applied. This chapter introduces basic methods that are relevant to save and search large sets of data as they are typical in data warehouse systems.

3.1. Introduction

Data warehouses store huge sets of multidimensional data. The databases of such a system can be as large as TB of data. Data of this size does not fit into main memory and is therefore stored on secondary or even on tertiary memory. To process the data and use it for computations within the CPU the data is transfered through different components of a computer system. These components are reviewed in a memory hierarchy. We describe the memory hierarchy that is typical for today's computer systems. Fast accesses to the data stored on secondary memory have to be provided to retrieve the data. The mechanics of the disks influence the performance of the index structures and is therefore investigated here.

We map the multidimensional tuples stored in the data warehouse into a multidimensional data space. We define the data space and the typical queries processed on the data. In order to process the specified queries efficiently different access structures have been designed in academic research. We present different kinds of structures and concentrate on tree-based based index structures and bitmap indexes.

3.2. Memory hierarchy

The major characteristic of such computer systems which the DBMSs run on is the memory hierarchy consisting of five levels [Gray and Reuter, 1993], [Härder and Rahm, 1999], [Garcia-Molina et al., 1999], [Saake and Heuer, 1999].

CPU registers are the fastest storage devices of a computer system and store the operands of computations. The next level is the cache. For read operations each value in a cache is a copy of some value in the main memory. There are often two kinds of cache. One small cache is integrated on the processor chip. One other larger

second level cache is placed on a separate chip. The main memory is a volatile storage and ranges from 128 MB to more than 10 GB for typical machines. Main memory is accessed randomly and access time ranges between 10^{-8} and 10^{-7} seconds.

The secondary memory is a non-volatile storage. The random access to disk is usually more than 10^5 times slower than the access to main memory. In database systems, transfering the data from the disk into main memory is the main performance bottleneck. The cost models in this thesis are based on the accesses to disks.

Tertiary memory has larger capabilities than secondary memory, but slower access times. Typical kinds of tertiary storage systems are: ad-hoc tape storages, optical-disk juke boxes, and tape silos. Tertiary storage access is approximately 1,000 times slower than secondary memory access but can be 1,000 times more capacious. In data warehouse environments it often archives and backups data.

3.3. Mechanics of disks

The complete database of a data warehouse does not usually fit into the main memory of a computer system. Therefore, the database is stored on the next level of the memory hierarchy, the secondary memory. As previously mentioned, access to secondary memory is more than 100,000 times slower than access to main memory. Often, this access to secondary memory is the bottleneck of a DBMS.

However, not all accesses to secondary memory are equal. Reading blocks sequentially from hard disks is much faster than random access.

A disk consists of a number of platters rotating around a central spindle. Each platter has two surfaces that are covered with magnetic material. Figure 3.1 sketches a disk with three platters and six surfaces. The bits are stored sequentially in concentric circles on the surfaces that are called tracks. Each track consists of a fixed number of sectors.

When data is requested from the disk complete sectors of data are read. Physical sectors are mapped to logical blocks. Often the terms “pages” or “chunks” are used in the same context. In this thesis we assume that one block is mapped to a fixed number of sectors and we use the words block and sectors interchangeable. The size of a block in KB is denoted by blocksize b.

As previously mentioned, the access to a block on secondary memory can be performed in two different modes. The time t_s for a sequential block access (transfer_time) is calculated by [Härder and Rahm, 1999]:

$$t_s = \text{transfer\_time} = \frac{b * 1024}{\text{transfer\_rate}} \tag{3.1}$$

The time t_r for a random block access is calculated by:

$$t_r = \text{seek\_time} + \frac{\text{rotation\_time}}{2} + \text{transfer\_time} \tag{3.2}$$

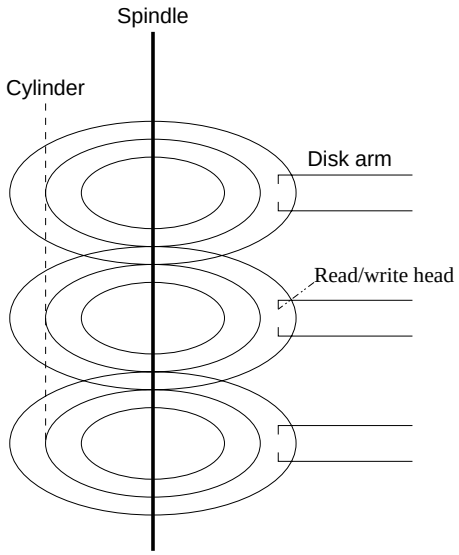


Figure 3.1.: Disk with three platters and six surfaces

With today's hard drives and block_sizes of 4 KB the time t_r for a random block access is approximately 10 to 20 times larger than t_s. This difference in access time increases the performance of structures which access large contiguous portions of data at the same time.

Experimental studies show that the seek_time and the rotation_time decrease by only 8 % per year, while the transfer_rate increases by 40 % per year [Patterson and Keeton, 1998], [Bitton and Gray, 1998]. Therefore, the ratio between the time for a random block access t_r and the time for a sequential block access t_s increases significantly. Sequential disk accesses are getting disproportionately cheaper. This difference between the sequential and random access times implies two facts that become even more important in the future:

1. If more than a certain fraction (approximately 5 to 10 %) of the data has to be accessed, a full table scan is faster than the use of a tree-based index structure. This fraction is decreasing every year.

2. For a comparison of different access structures it is not sufficient to count only the block I/Os . The number of random block accesses and sequential block accesses have to be weighted differently.

In the next chapters we will consider these two facts and use them for the comparison of index structures.

3.4. Data space and queries

We assume the data of a fact table introduced in Section has to be indexed. The fact table is the relation schema $R(a_1, a_2, \cdots, a_n, s)$, where a_j are the key attributes and the names of the *dimensions*. The fact attribute is denoted as s. Each dimension a_j has the domain A_j and s is a summary value.

3.4.1. Data space

Without loss of generality we assume that an index is built on the first $d \in \{1, \cdots, n\}$ attributes. The index structure does not consider the other $(n-d)$ attributes. Pointers (tid) are stored in the leaf nodes of the index structure to point to the locations where the complete tuples are stored [Härder and Rahm, 1999]. We assume that all indexed attributes are discrete values. The cardinality of the different domains A_j is given by $c_j = |A_j|$ for all $j \in \{1, \cdots, d\}$. Each set A_j is coded to a set of non-negative integers $O_j = \{0, \cdots, c_j - 1\}$.

Definition 3.1 The d-dimensional data space is defined as the set $O = O_1 \times \cdots \times O_d$.

For each tuple of relation R one index entry (p, tid) is created, where $p \in O$ and $tid \in TID$ is a unique Tuple IDentifier and TID is the set of all correct Tuple IDentifiers. The set of all index entries is $A \subset O \times TID$ where $\forall (p_1, tid_1), (p_2, tid_2) \in A : tid_1 = tid_2 \Rightarrow p_1 = p_2$.

For storing set A, this set is partitioned into subsets which store the elements of each subset on one block. When partitioning A into subsets the tuples, that have points which are close to each other, should be in the same block. Each block is represented often by a minimum bounding box or minimum bounding rectangle that all tuples include. These regions are represented as d-dimensional hyper-rectangles. Definition 3.2 specifies these d-dimensional hyper-rectangles.

Definition 3.2 A d-dimensional hyper-rectangle I of the data space O is defined as $I = [l_1, u_1] \times \cdots \times [l_d, u_d] \subset O$ where $l_j \in \{0, \cdots, c_j - 1\}$ is the lower limit in the jth dimension and $u_j \in \{0, \cdots, c_j - 1\}$ is the upper limit in the jth dimension $(l_j \leq u_j) \forall j \in \{1, \cdots, d\}$.

3.4.2. Queries

An index structure processes different kinds of queries. We define the most important query types below.

Point queries. Point queries retrieve all elements of a specific point p.

Definition 3.3 A point query $PQ : O \rightarrow 2^A$ with $PQ(p) = \{(p, tid) \in A\}$ is a query that returns all elements having exactly the value of point p.

Range queries. Range queries retrieve all elements that are contained in a d-dimensional hyper-rectangle I.

Definition 3.4 A range query $RQ : 2^I \rightarrow 2^A$ with $RQ(I) = \{(p, tid) \in A | p \in I\}$ is a query returning all elements where p is included in the range of the hyper-rectangle I.

The size of a query box is given as $q = (q_1, \cdots, q_d) := (u_1 - l_1, \cdots, u_d - l_d)$. A partial range query is a query with a query box where some dimensions are not restricted. Each partial range query can be simulated by a range query. For each dimension i, for which the partial range query is not restricted, the hyper-rectangle of the range query is set to $l_j = 0$ and $u_j = c_j - 1$. Therefore, we shall only use the term range queries in this thesis.

Nearest neighbor. Nearest neighbor queries are important for similarity search and they retrieve the closest data items to some specified point p.

Definition 3.5 A nearest neighbor query is $NNQ : O \rightarrow 2^A$, where $NNQ(p) = \{p' | \forall p'' : dist(p, p') \leq dist(p, p'')\}$

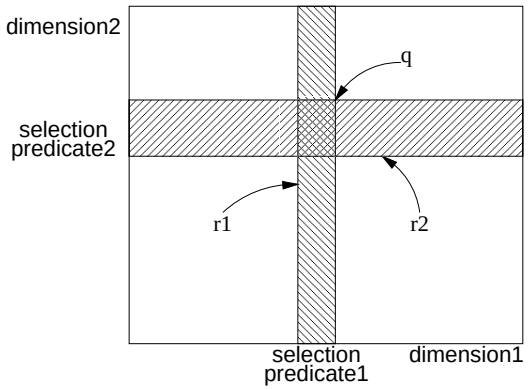
The distance between two points is calculated by a function $dist$. Different metrics can be used here, such as Euclidean metric, Manhattan metric, or maximum metric.

The previous definitions characterize the data that is indexed and the queries that are executed. In the context of data warehouses range queries on aggregated data are of main interest. Therefore, we investigate what kind of index structures efficiently support these queries for a given set of data by using the hardware described in Section 3.2 and Section 3.3.

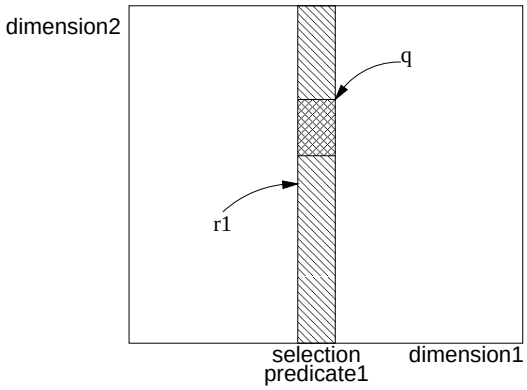
3.5. Tree-based indexing

In the beginning there was the B-tree. The B-tree [Bayer and McCreight, 1972] is a widely used one-dimensional tree-based index structure in DBMSs. It is proven that there is no better one-dimensional index structure with the same generality and flexibility than the B-tree. However, there is no general solution on how to apply the B-tree for indexing multidimensional data.

One method is to generate one B-tree for each attribute. If there is a range query in more than one dimension, the B-trees for all selected dimensions are applied. If $d = 2$, the result sets r_1 and r_2 of the indexes are calculated and intersected. The left side of Figure 3.2 shows this approach. The result sets of $predicate_1$ and $predicate_2$ of query q is calculated by selecting all tuples that are in both result sets.



Two B-trees: Result sets r_1 and r_2 have to be intersected



One B-tree: Result set r_1 has to be evaluated completely

Figure 3.2.: Use of B-tree for multidimensional data ($d = 2$)

Another possibility is the use of just one B-tree. For example, the B-tree in dimension 1 is used. Then each tuple that belongs to the result set r_1 is loaded and we evaluate if the tuple satisfies $predicate_2$. This idea is sketched on the right side of Figure 3.2. However, this approach has the main drawback that much more tuples are loaded and evaluated than actually belong to the result set.

A third alternative to index multidimensional data with one-dimensional B-trees uses compound indexes. For each permutation of a set of dimensions a one-dimensional index is built. In our example the indexes a_1, a_2 and index a_2, a_1 are created. The number of necessary indexes is the main disadvantage of this approach. For d-dimensional data $d!$ indexes must be created and must be maintained. This approach is not feasible for high dimensional data.

A fourth option is the mapping of the multidimensional data space into a one-dimensional data space with a space filling function. A space filling function defines a one-dimensional ordering for a multidimensional space [Markl, 1999], see Section 3.5.7.

The remainder of this chapter discusses some basic properties of multidimensional index structures. We cannot give a complete survey; we rather simply sketch some ideas that help us to understand what index structures for secondary memory are used in the context of indexing data warehouse data efficiently. For further details we refer to standard references [Samet, 1989], [Samet, 1990], [Gaede and Günther, 1998].

.

3.5.1. Top-down, bottom-up, and bulk loading

For a given set of data there are two approaches in creating index structures. The first approach is the top-down method; that is the data is added from the root to the

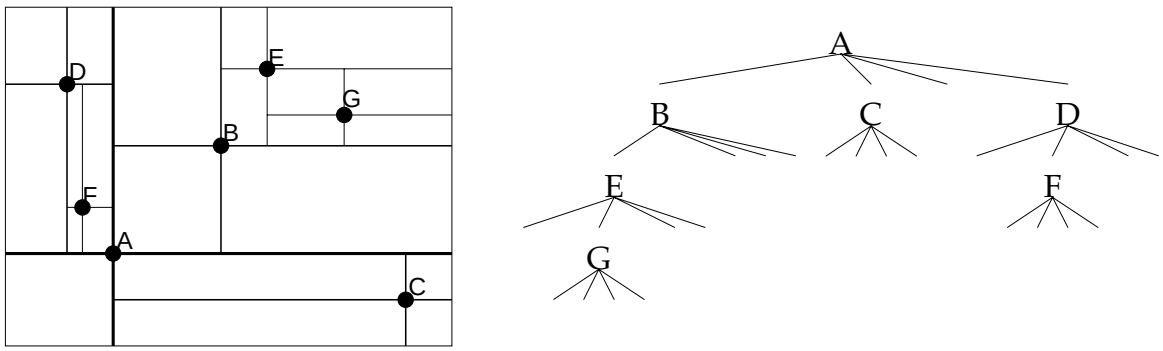


Figure 3.3.: Point quadtree: Data space and tree representation

leafs in the tree. The top-down approach should only be applied, if all of the data is not known in advance and data has to be inserted and deleted frequently. The second approach is the bottom-up method. Here, the leafs are created first. Then the upper levels of the tree are built successively from the bottom to the top. The bottom-up approach works best, if all data is known in advance and no changes are made after the create phase. Hybrid techniques can be applied. In this case the index is created efficiently with a bottom-up technique and changes are propagated with the top-down method.

Inserting data in a multidimensional index structure incrementally is expensive. An approach to alleviate this problem is to attach a buffer to each node [van den Bercken et al., 1997]. If the node exceeds its capacity tuples are loaded into the buffer. This allows inserting many tuples at the same time and postponing the expensive split operations. Related techniques are the small-tree-large-tree approach [Chen et al., 1998] and the buffer tree [Arge, 1995].

3.5.2. Point quadtrees

Point quadtrees [Finkel and Bentley, 1974] are multidimensional extensions of binary trees. Two-dimensional point quadtrees ($d = 2$) divide the data space into non uniformly sized cells. Each cell stores up to a fixed number of data points. Once the capacity is exceeded, the cell is split into the $2^d = 4$ sub-cells: NE, SE, SW, and NW clockwise in this order. The left side of Figure 3.3 represents some cells in a plane while the right side of the same figure shows the structure of the quadtree.

The main drawback of this structure is that each node that exceeds its capacity is split into four sub-nodes in the two-dimensional case. Therefore, the minimal number of entries per node can be as low as $\frac{1}{4}$ of the maximum capacity. For the three-dimensional case this structure is also called octree. If a split occurs in the octree, each node is split into eight sub-nodes. In general, for a d-dimensional tree each split creates 2^d descendants. Therefore, the average space utilization might be low for a high number of dimensions.

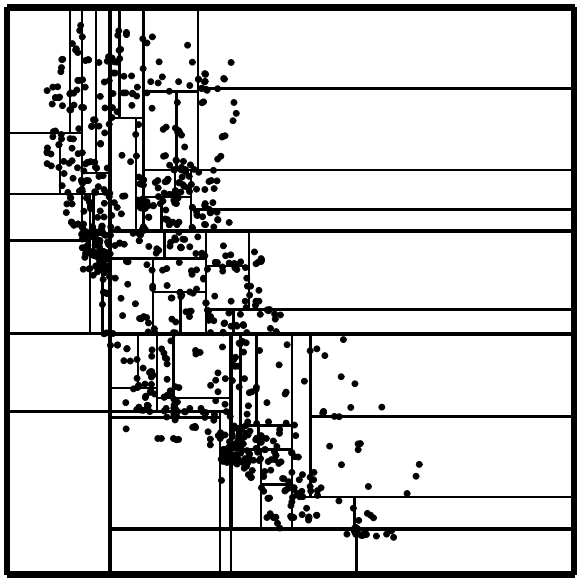


Figure 3.4.: kdb-tree with 1,000 locations in California

3.5.3. kd-tree

The kd-tree [Bentley, 1975] is in contrast to the point quadtree a binary tree. A node split in a kd-tree is processed by splitting a node into two child nodes according to one dimension. The dimension for that the split is chosen either randomly or according to some rules. The kd-tree guarantees that each leaf nodes is filled by at least 50 %. However, the top down approach of the kd-tree is an unbalanced index structure.

3.5.4. kdb-tree

The **kdb**-tree [Robinson, 1981] combines the balanced structure of the B-tree and the multidimensional features of the kd-tree. The path from the root to the leafs has always the same length; some pages may be split without any overflow. This effect is called *cascade splitting* and might generate some nodes without any entries. Figure 3.4 shows a kdb-tree in which coordinates of cities in California are inserted. In the lower left corner *cascade splits* generate empty pages.

3.5.5. R-tree

The R-tree [Guttman, 1984] is a multidimensional generalization of the B-tree. In contrast to the B-tree which uses one-dimensional intervals as atomar elements, the R-tree applies multidimensional rectangles to represent multidimensional intervals. The R-tree consists of two different types of nodes. *Leaf* nodes and *non-leaf* nodes. *Leaf* nodes contain entries of the form $(tid, rectangle)$ with tids referring to records in

the database. *Non-leaf* nodes contain entries of the form $(cp, rectangle)$ with cp being

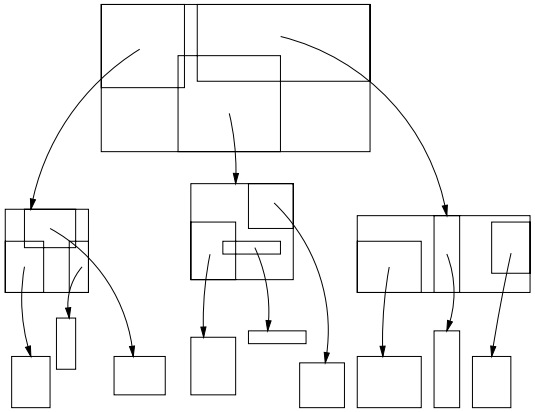


Figure 3.5.: R-tree with rectangles as atomar elements

a pointer to a child node of the R-tree and $rectangle$ being the minimum bounding rectangle of all rectangles which are entries of that child node (cf. Figure 3.5). Each rectangle is represented as a hyper-rectangle respectively a hyper-interval I as defined in Definition 3.2 on page 18. *Non-leaf* nodes are used to direct the path to the *leaf* nodes. Therefore, they are also called directory nodes.

In contrast to the B-tree and the other index structures, the R-tree contains overlaps of regions. This implies, that no worst case analysis is possible. For a point query there can be ambiguous ways that have to be traversed. The R^+-tree [Sellis et al., 1985] is an approach to overcome this problem by using clipping to prevent overlaps. There are several other extension of the R-tree, some of which are briefly discussed in the next two sections.

3.5.6. R*-tree

The R^*-tree [Beckmann et al., 1990] provides a better insertion algorithm than the R-tree. It uses a *forced reinsert* mechanism to reorganize the structure. This mechanism enables the structure to adapt to data distributions and not to suffer from rectangles inserted previously. Experimental comparisons of the R-tree family show that the R^*-tree performs faster than the other structures [Beckmann et al., 1990].

The maximum fanout B and the minimum fanout b of the nodes are important parameters for trees. Let B_{dir} denote the maximum number of directory entries fitting in one node and let b_{dir} be the minimum number of entries in a directory node. The parameter b_{dir} satisfies the following condition: $2 \leq b_{dir} \leq \frac{B_{dir}}{2}$. The parameters B_{leaf} and b_{leaf} are defined in the same way. The ratio between b_{leaf} and B_{leaf} respective b_{dir} and B_{dir} influences the performance of the R^*-tree. A high value of b_{leaf} yields to less leaf nodes, but to more overlaps between the regions of different nodes on the same level. Beckmann *et al.* got the best tradeoff for $\frac{b_{dir}}{B_{dir}} = \frac{b_{leaf}}{B_{leaf}} = 0.4$

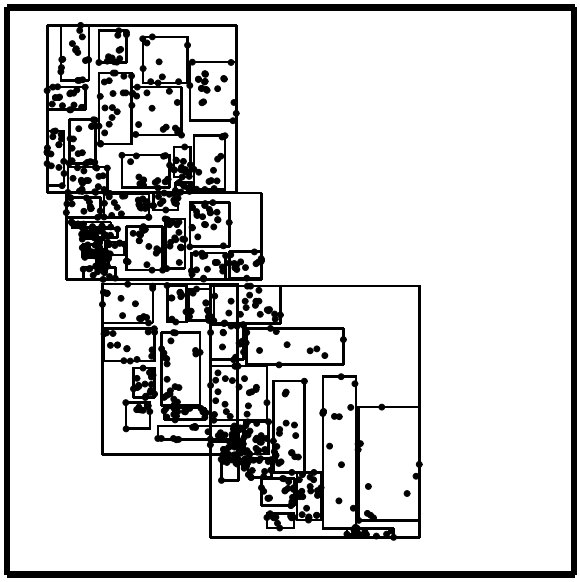


Figure 3.6.: R^*-tree with 1,000 locations in California

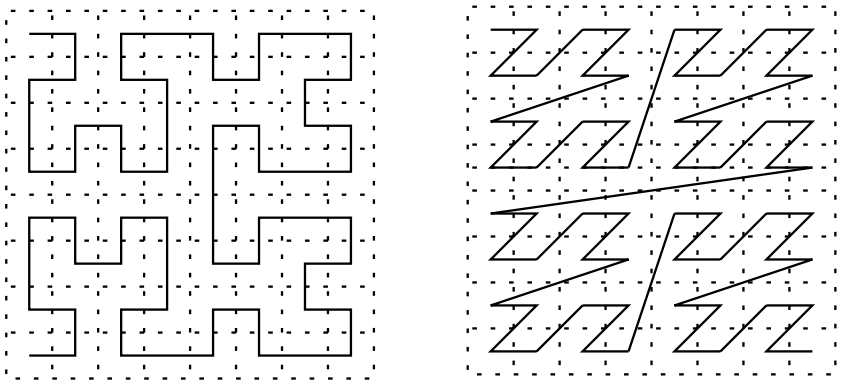
[Beckmann et al., 1990]. In experiments later presented in this thesis this value is used as well.

3.5.7. Other relatives of the R-tree family and other tree structures

There are many different extensions besides the R^+-tree and the R^*-tree to improve the performance of the R-tree. This section discusses some approaches briefly.

The packed-R-tree [Roussopoulos and Leifker, 1985] is a bottom-up structure and clusters data items together in a data node according to a nearest neighbor function. The idea of the Hilbert R-tree [Kamel and Faloutsos, 1994] is to cluster the data together in nodes of the same level according to a Hilbert curve (cf. left part of Figure 3.7). The Hilbert curve is applied to calculate a one-dimensional Hilbert value for multidimensional points/rectangles. The points are then clustered according to their Hilbert values. The rectangles are clustered according to the Hilbert value of their center. The HG-tree [Kuan and Lewis, 1999] is a multidimensional tree structure designed for point data based on the Hilbert R-tree. The Simple Tile R-tree (STR-tree) [Leutenegger et al., 1997] is a bottom-up structure like the packed R-tree, but it applies the Sort-Tile-Recursive algorithm to cluster rectangles. The X-tree [Berchtold et al., 1996] is an R^*-tree with variable size of nodes. The size of a node can be enlarged to prevent node splitting that would yield to nodes with large overlaps.

Rectangles are not well suited for nearest neighbor queries. The SS-tree [White and Jain, 1996] uses spheres instead of rectangles. Figure 3.8 shows an



Hilbert curve Z-Ordering

Figure 3.7.: Space filling curves

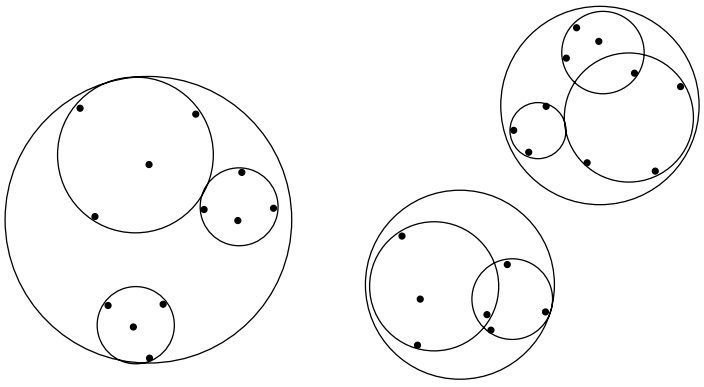


Figure 3.8.: SS-tree width spheres as atomar elements

example to organize the data. A benefit of this structure is that a region is defined by storing one d-dimensional point and the radius. Only $(d + 1)$ numbers are stored for each region, whereas the R-tree defines its regions with $2d$ coordinates. The SR-tree [Karayama and Satoh, 1997] is a combination of the SS-tree and R-tree. Weighted dimensions [Großer, 1997] apply a priority scheme to split an R^*-tree more often in selected dimensions than in other dimensions. It is advantageous to split in these dimensions more frequently in which the query boxes are restricted mostly. Another way of improving the performance is achieved by executing R-trees in parallel [Schnitzer and Leutenegger, 1999].

A flexible tree structure for indexing spatial objects is the cell tree [Günther, 1989]. Figure 3.9 shows an example of the cell tree. The cell tree

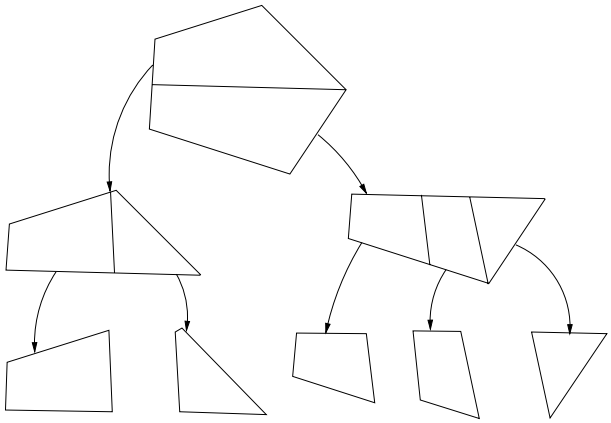


Figure 3.9.: Cell Tree with polygons as atomar elements

does not use hyper-rectangles as atomar data items, but more general polygons.

The idea of the UB-tree [Bayer, 1996], [Bayer and Markl, 1998], [Markl, 1999] is to sort the multidimensional data according to the Z-ordering (cf. right part of Figure 3.7). For each multidimensional point the corresponding Z-value of the square in which the point is contained is calculated. The corresponding Z-values are then indexed using a B-tree.

3.5.8. Generic tree structures

During the last years many different index structures have been developed. Some of them are presented in this chapter. New index structures differ often just in a few details from existing ones. Therefore, some approaches are developed to provide generic frameworks where the common part of the index structures is given by the system and only the differences are specified and implemented by researcher and developer. One approach is the *Generalized Search Tree* (GiST) [Hellerstein et al., 1995], [Kornacker et al., 1997], [Kornacker, 1999] where different index structures are implemented by just defining the four functions *consistent*, *union*, *penalty*, and *pick-*

split. The *Access Method DeBuging tool* (AMDB) [Kornacker et al., 1998] illustrates for a given data and query set the performance of the structure. Gurret et al. focuses mainly on spatial benchmarks [Gurret and Rigaux, 1998] and implement different spatial index structures and join algorithms in one framework. The performance therefore can be compared easily. Günther et al. discuss related techniques for comparing spatial join algorithms [Günther et al., 1998].

3.6. Bitmap indexing

Bitmap indexing is rather different from the tree-based indexing considered previously. One of the main benefit of bitmap indexing techniques is that they are easy to implement. In addition, the operations of the bitmap indexing techniques are mostly reading large blocks of bits and Boolean operations on vectors of bits (bitmap vectors). These operations are performed very efficiently. This is one reason, why bitmap indexing techniques are implemented in commercial database systems, *e. g.* Oracle [Christiansen et al., 1998], Sybase [Sybase, 1997], and Informix [Informix, 1997]. Among the disadvantages of bitmap indexing techniques are the facts that they can be very space consuming and that the insert / update operations are more expensive than for tree-based structures.

O'Neil et al. compare different indexing techniques in a rather qualitative approach [O'Neil and Quass, 1997]. Equality encoded and range encoded indexing techniques [Chan and Ioannidis, 1998] are promising structures for *read-mostly* environments. Interval encoded bitmap indexing techniques [Chan and Ioannidis, 1999] are optimal under certain conditions. In the presence of hierarchies of the attributes special bitmap structures are used by some bitmap indexing techniques [Wu and Buchmann, 1998].

3.6.1. Standard bitmap indexing

The idea of bitmap indexing is simple. First, we treat each dimension separately. For each attribute a_j a number of c_j bitmap vectors are generated, with $c_j = |A_j|$. Each bitmap vector has a length of t bits, where t is the number of tuples it indexes. The jth bit of the kth bitmap vector is set to 1, if the jth tuple corresponds to the kth value and it is set to 0 otherwise. The left side of Figure 3.10 shows the first five rows of the projection of a relation R on attribute a_1. We assume A_1 consists of 12 different values which are mapped to integer numbers between 0 and 11. Figure 3.10 presents the bitmap index for this data. Each column B_{11} to B_0 represents one bitmap vector. If all tuples have to be selected where $a_1 = 3$ the bitmap vector B^3 is read and all tuples, with the bits set to 1, are chosen ($PQ(3) \approx B^3$). Such a set of bitmap vectors is generated for all dimensions.

If preconditions like $(a_1 = 3) \wedge (a_2 = 2)$ are evaluated, one bitmap vector for attribute a_1 and one bitmap vector for attribute a_2 are read and a Boolean AND operation is performed.

$\pi_{a_1}(R)$	B^{11}	B^{10}	B^9	B^8	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
5	0	0	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0	1	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3.10.: Original data and standard bitmap index

The size of the bitmap index depends on the number of tuples and on the cardinality of the attributes. The size of one bitmap vector given in blocks is calculated as $v = \lceil \frac{t}{8192*b} \rceil$, where b is the block_size in KB and t is the number of tuples that are indexed. For each element of the domain A_j one bitmap vector is created. The total number of bitmap vectors is calculated by:

$$Space = \sum_{i=1}^{d} c_j \tag{3.3}$$

However, bitmap indexes in the form presented here, have some drawbacks. Firstly, the cardinality of the domain of each attribute has to be known in advance. In *read-mostly* environments which are in the focus of this thesis this is a reasonable assumption. A second drawback is, if the cardinality of the domain is large, standard bitmap indexes become very space consuming.

Finally, range queries are very typical for data warehouse systems. Standard bitmaps are not very efficient for range queries. Consider the above example from Figure 3.10 and a range query like $2 \leq a_1 \leq 7$. For this query six bitmap vectors B^2 to B^7 are read and distributively combined as:

$$RQ([2,7]) \approx (B^7 \vee B^6 \vee B^5 \vee B^4 \vee B^3 \vee B^2) \tag{3.4}$$

Alternatively, the bitmaps B^0, B^1 and B^8 to B^{11} are read, distributively combined and the compliment calculated: $RQ([2,7]) \approx \neg(B^{11} \vee B^{10} \vee B^9 \vee B^8 \vee B^1 \vee B^0)$. In each case six bitmaps vectors are read.

The next sections describe extensions of bitmap indexing techniques to overcome the above mentioned problems. One approach of multi-component equality encoded bitmap indexes which increases the space efficiency of standard bitmaps is presented. A further approach investigates range encoded bitmap indexes to support range queries more efficiently. A hybrid approach combines both approaches to a space and time efficient index for processing range queries.

$\pi_{a_1}(R)$	B_1^2	B_1^1	B_1^0	B_0^3	B_0^2	B_0^1	B_0^0
5	0	1	0	0	0	1	0
3	0	0	1	1	0	0	0
0	0	0	1	0	0	0	1
3	0	0	1	1	0	0	0
11	1	0	0	1	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
x	$\underbrace{\qquad\qquad}_{y}$			$\underbrace{\qquad\qquad\qquad}_{z}$			

Figure 3.11.: Base-<3,4> equality encoded bitmap index

3.6.2. Multi-component equality encoded bitmap index

The equality encoded bitmap index [Chan and Ioannidis, 1998], overcomes the problem of being very space consuming for attributes with large domains. It could be argued, that space is no longer an issue. However, the occupied space is proportional to the time for creating such an index. Therefore, the space measures the creation time and even in *read-mostly* environments this time cannot be neglected. The main idea of compressing bitmap indexes presented here can be seen as an encoding of the values of a_1 into a different number system. For example, the values 0 to 11 from the above example are encoded into the <3,4> number system. Each value $x \in \{0, \cdots, 11\}$ is encoded by $x = (4 * y + z)$ where $y \in \{0, 1, 2\}$ and $z \in \{0, 1, 2, 3\}$, $y := \lfloor x \div 4 \rfloor$, $z := (x \mod 4)$. The values of y and z are then stored like standard bitmap indexes. Figure 3.11 shows the resulting structure. The main benefit of this approach is the reduced space consumption in comparison with the standard bitmap index. The standard bitmap index in Figure 3.10 needs 12 bitmap vectors. The <3,4> encoded bitmap vector in Figure 3.11 stores seven bitmap vectors. However, the savings in space reduce the time efficiency of the structure. For a point query, two bitmap vectors are read with the <3,4> equality encoded bitmap index. To compute the result for the query $a_1 = 3$ the term $PQ[3] \approx (B_1^0 \wedge B_0^3)$ is evaluated.

For larger attribute cardinality c_j the differences in space and time are even more significant. Consider an example, where $c_1 = 1,000$ different values are indexed. The standard bitmap index can be seen as a <1000> encoded structure and 1,000 bitmap vectors are created. For each of the values between 0 and 999 one bitmap vector is stored. One way of compression is to decode each digit of the values separately. This <10,10,10> multi-component equality encoded bitmap index stores only 30 bitmap vectors. The number of bitmap vectors, read for a query like $a_i = 352$ is increased from one bitmap vector of term (B^{352}) to three bitmap vectors in term $PQ[352] \approx (B_2^3 \wedge B_1^5 \wedge B_0^2)$, while the number of necessary vectors is decreased.

This example shows the time-space tradeoff. There are four interesting points of this tradeoff: time optimal, space optimal, "knee", and time optimal under given space constraint [Chan and Ioannidis, 1998]. In this example the time optimal index has the base of <1000>. This is the standard bitmap index. The space optimal

$n_j = 0$
`repeat`
$n_j = n_j + 1$
$b_j = \lfloor m_j / n_j \rfloor + 1$
$r_j = (m_j + n_j) mod\ \ n_j$
`until` $b_j^{r_j}(b_j - 1)^{n_j - r_j} \geq c_j$

Figure 3.12.: Calculation of base for multi-component equality encoded bitmap indexes for given m_j and c_j

index has the base of <2,2,2,2,2,2,2,2,2,2>. This is the binary representation of the values. The more interesting tradeoffs are somewhere in between (*e. g.* <34,33> or <10,10,10>). In the following chapters we construct index structure which are time optimal under given space constraint.

The processing of range queries with a multi-component equality encoded bitmap index is more complex. The query $2 \leq a_1 \leq 7$ is processed by evaluating the expression:

$$RQ([2,7]) \approx \underbrace{(\neg B_1^0 \vee (B_1^0 \wedge (B_0^2 \vee B_0^3)))}_{2 \leq a_1} \wedge \underbrace{\neg B_1^2}_{a_1 \leq 7} \tag{3.5}$$

Figure 3.12 sketches the algorithm to calculate the base for an equality encoded bitmap index. This algorithm performs for each $j \in \{1, \cdots, d\}$ with the input parameters cardinality of the attribute c_j and the maximum number of bitmap vectors m_j. The output are the number of components n_j and the size of each component (depends on b_j and r_j). An additional optimization step (not shown here), improves the performance of the bitmap index structures [Chan and Ioannidis, 1998]. The base in each dimension j is then given by:

$$< \underbrace{b_j - 1, \cdots, b_j - 1}_{n_j - r_j}, \underbrace{b_j, \cdots, b_j}_{r_j} > \tag{3.6}$$

In this thesis, the base in the jth dimension is denoted as:

$$< b_{j1}, b_{j2}, \cdots, b_{jn_j} > = < \underbrace{b_j - 1, \cdots, b_j - 1}_{n_j - r_j}, \underbrace{b_j, \cdots, b_j}_{r_j} > \tag{3.7}$$

For example: b_{23} denotes the base of the third component of the second attribute / dimension. The number of bitmap vectors in all dimensions is:

$$Space = \sum_{j=1}^{d} m_j = \sum_{j=1}^{d} \sum_{i=1}^{r_j} b_{ji} = \sum_{j=1}^{d} ((n_j - r_j)(b_j - 1) + r_j b_j) \tag{3.8}$$

The average number of bitmap vectors which have to be read for processing a range query [Chan and Ioannidis, 1998] is:

$$B_{equal} = \sum_{j=1}^{d} \sum_{i=1}^{n_j} E_{r_j,i}, \text{ where}$$
$$E_{r_j,i} = \begin{cases} \frac{1}{b_{ji}} \left(\left\lfloor \frac{b_{ji}}{2} \right\rfloor^2 + \left(b_{ji} - 1\right) \left(\left\lceil \frac{b_{ji}}{2} \right\rceil - \frac{b_{ji}}{2} \right) \right) & : \quad b_{ji} > 2 \\ 1 & : \quad \text{otherwise} \end{cases} \tag{3.9}$$

3.6.3. Range-based encoding

As we argued before, standard bitmap indexes do not efficiently process range queries. This section describes an approach that supports range queries more time efficient than the previous described techniques. There are different definitions of range-based encoding [Chan and Ioannidis, 1998], [Wu and Buchmann, 1998]. In this thesis we use the definition by Chan and Ioannidis.

The main idea is to set the bit of the kth bitmap vector to 1 if the value is smaller or equal the kth value. More formally, the range encoded bitmap index is calculated from the equality encoded indexes by:

$$\overline{B}^i = \begin{cases} \overline{B}^{i-1} \vee B^i & : \quad i \geq 1 \\ B^i & : \quad i = 0 \end{cases} \tag{3.10}$$

where $\overline{B}$ indicates the use of the range encoded bitmap index.

In this approach range queries of arbitrary size are processed by just reading two bitmap vectors. Figure 3.13 shows an example of a range-based index structure for the same values as in Figure 3.10. The range query $2 \leq a_1 \leq 7$ from the previous example is processed by evaluating the term:

$$RQ([2,7]) \approx \underbrace{\overline{B}^2}_{2 \leq a_1} \wedge \underbrace{\neg \overline{B}^8}_{a_1 \leq 7} \tag{3.11}$$

Since bitmap vector $\overline{B}^{11}$ equals 1 for all tuples, this vector is not stored. The space (in bitmap vectors) for range-based encoding is:

$$Space = \sum_{j=1}^{d} (c_j - 1) \tag{3.12}$$

The average number of bitmap vectors that are read for a range query of arbitrary size is:

$$Time = 2 \sum_{j=1}^{d} \frac{b_j - 1}{b_j} \tag{3.13}$$

$\pi_{a_1}(R)$	$\overline{B}^{10}$	$\overline{B}^{9}$	$\overline{B}^{8}$	$\overline{B}^{7}$	$\overline{B}^{6}$	$\overline{B}^{5}$	$\overline{B}^{4}$	$\overline{B}^{3}$	$\overline{B}^{2}$	$\overline{B}^{1}$	$\overline{B}^{0}$
5	1	1	1	1	1	1	0	0	0	0	0
3	1	1	1	1	1	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3.13.: Single component range-based encoded index

$\pi_{a_1}(R)$	$\overline{B}_1^1$	$\overline{B}_1^0$	$\overline{B}_0^2$	$\overline{B}_0^1$	$\overline{B}_0^0$
5	1	0	1	1	0
3	1	1	0	0	0
0	1	1	1	1	1
3	1	1	0	0	0
11	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	

Figure 3.14.: multi-component-<4,3> range encoded index

3.6.4. Multi-component range-based encoding

The two techniques described in the Section 3.6.2 and Section 3.6.3 improve standard bitmap indexes. The multi-component bitmap index reduces the problem of low space efficiency of standard bitmap indexes for attributes with large domains. The range encoded bitmap index supports range queries more efficiently. We combine both techniques. The new structure is called a multi-component range-based encoded bitmap index. Figure 3.14 shows an example. For processing the query

$n_j = 0$
`repeat`
$n_j = n_j + 1$
$b_j = \lfloor m_j / n_j \rfloor + 1$
$r_j = (m_j + n_j) mod \ \ n_j$
`until` $(b_j + 1)^{r_j} {b_j}^{n_j - r_j} \geq c_j$

Figure 3.15.: Calculation of base for multi-component range encoded bitmap indexes for given m_j and c_j

$2 \leq a_1 \leq 7$ the term

$$RQ([2,7]) \approx \underbrace{(\neg \overline{B}_1^1 \vee (\neg \overline{B}_0^2 \wedge \overline{B}_1^0))}_{2 \leq a_1} \wedge \underbrace{(\neg \overline{B}_1^1)}_{a_1 \leq 7} \tag{3.14}$$

Figure 3.15 shows the algorithm for calculation of the base of the range encoded bitmap index. This algorithm is executed for each j, $j \in \{1, \cdots, d\}$. The result defines the base in each dimension as:

$$< \underbrace{b_j, \cdots, b_j}_{n_j - r_j}, \underbrace{b_j + 1, \cdots, b_j + 1}_{r_j} > \tag{3.15}$$

The space (in bitmap vectors) allocated by this structure is:

$$Space = \sum_{j=1}^{d} m_j = \sum_{j=1}^{d} \sum_{i=1}^{r_j} (b_{ji} - 1) = \sum_{j=1}^{d} ((n_j - r_j) b_j + r_j (b_j + 1)) \tag{3.16}$$

With the given bases for the multi-component equality encoded bitmap indexes it is possible to estimate the time needed to process queries by the structure. The number of bitmaps that have to be scanned for a specific configuration according to [Chan and Ioannidis, 1998] is:

$$B_{range} = \sum_{j=1}^{d} 2 \left(\frac{(n_j - r_j)(b_j - 1)}{b_j} + \frac{r_j b_j}{b_j + 1} \right) \tag{3.17}$$

3.6.5. Other compression techniques / combination of bitmaps and trees

In cases where the attribute cardinality is high many bits in the bitmap vector representation are 0. There are other than previous described techniques to compress a large number of zeros. Let us assume the space needed for storing one tid is 4 Bytes respectively 32 Bits. If less than $\frac{1}{32}$ of all values are 0 it is more space efficient to store a list of Tuple IDentifiers instead of a complete bitmap vector [O'Neil and Quass, 1997]. To organize the bitmap vectors and/or the tid-lists, a B-tree is used on top of the bitmap vectors/tid-lists. This technique can adapt to the actual data. If there are only a few different values, there are a few bitmap vectors. If the number of different values increases, the bitmap vectors are more sparse. Once less than a certain fraction (*e. g.* $\frac{1}{32}$) of all values are 1 it might be worth to change the index structure to a tid-list. However, the transformation from the tid-list to a bitmap representation implies some additional computation overhead.

3.7. Arrays

Arrays are efficient storage structures for dense multidimensional data. Each cell of the array holds one cell of the multidimensional data space. If the array is stored on

secondary memory extensions and reorganizations of the arrays are expensive operations. Extendible arrays overcome this problem [Rotem and Zhao, 1996]. Complete reorganizations of the arrays are avoided and new data is appended to the old data. Small structures, that can be held in main memory, allow searching and retrieving of elements in the extended structure.

The mapping of array cells to blocks on secondary memory is called tiling. This tiling influences the performance of arrays stored on disks [Marques et al., 1998]. Partial sums are stored in arrays to support the calculation of range queries on aggregated data [Ho et al., 1997]. In general, arrays are used only for small databases (up to 50 GB). The approach of storing data in multidimensional structures is called *M*ultidimensional *OLAP* or MOLAP.

3.8. Summary

Data warehouses systems store large sets of data for the purpose of interactive decision support. Special techniques are applied to support fast access to the data. Due to the fact that the data cannot be completely stored on main memory, but on the secondary memory, the memory hierarchy and the mechanics of the disks influence the performance of such systems. Multidimensional index structures are developed to access the data efficiently. In detail, we presented multidimensional structures like the *R*-tree *family* and bitmap indexing techniques. For bitmap indexing techniques we presented formulars for occupied space and query processing time.