7. Techniques for Comparing Index Structures

It is evident from the preceeding chapters that the various index structures accelerate query processing differently. For some kinds of queries and data certain classes of index structures perform fast, while for others other classes of index structures are more efficient. Sarawagi makes qualitative propositions for a comparison of index structures [Sarawagi, 1997]. The performance of query processing depends on various parameters which influence the query execution time. We focus on a set of nine parameters. Two approaches are presented to support the decision making process which index structure should be applied. The first approach is based on classification trees. The second approach uses an aggregation method. Both approaches are applied to two classes with altogether four distinct index structures: a tree-based index structure without aggregated data, a tree-based index structure with aggregated data and two bitmap index structures. This chapter closes with results of a detailed performance study.

7.1. Introduction

Most performance investigations of index structures only consider one or two parameters at at time such as the blocksize b or the number of dimensions d. However, the performance of index structures depends on more than one or two parameters and there exist interactions between them. However, this chapter concentrates on a set of nine different parameters to compare index structures for processing range queries. The nine parameters are carefully chosen to describe data, queries, system behavior, and disk technology. We think that the nine parameters describe the experimental setup sufficiently precise. However, if more parameters should be needed, the described approaches can be extended easily to consider additional parameters.

7.2. Experimental parameters

7.2.1. Data specific parameters

Data specific parameters are described below:

Dimensionality. Dimensionality $d \in \mathbb{N}$ of the data denotes the number of attributes. For the task of indexing eight-dimensional data a different index structure may be better suited than for indexing one or two-dimensional data.

Number of stored tuples. The number $t \in \mathbb{N}$ of tuples influences the performance of different structures. The R_a^*-tree improves in comparison with the R^*-tree if the number of indexed tuples increases [Jürgens and Lenz, 1998].

Cardinality of data space. The cardinality c of the range of an attribute is the number of different values an attribute may have. Gender has three possible values (male, female, NULL). Attributes like social security number or telephone number may have millions of different values. For attributes like gender different index structures are better suited than for attributes like social security number or telephone number. We assume that the attribute space is normalized in the range $[0, 1)$ and that the attribute cardinality is the same in all dimensions. This assumption simplifies the model. It is relaxed later to make the experiments more realistic. Then the cardinality is denoted by c_j, ($j \in \{1, \cdots, d\}$).

The distribution type of data may be another parameter (*e.g.* uniform versus not uniform distribution). This Chapter does not consider this parameter. However, the described techniques can easily be extended to take other distributions into account. In this case the models for the tree structures are changed according to the PISA model (cf. Chapter 6). This chapter assumes uniformly distributed data. Note, that models for bitmaps are not affected by non-uniform distributions of data.

7.2.2. Query specific parameters

Query specific parameters hold information about the queries processed by the system. In our approach we concentrate on range queries. Point queries are expressed as range queries with query box size $qs = 0$. In order to have only scalar values as parameters we assume that there are only range queries which can be described with the two scalar values query box size qs and query box dimensions qd:

Query box size. The size of the query box $qs \in \mathbb{R}$ is a fraction of the data space and is denoted by $qs \in [0, 1]$. A value of $qs = 0.04$ means that the query box fills 4 percent of the data space.

Query box dimensions. The query box dimension parameter $qd \in \mathbb{N}$ denotes the number of attributes occurring in a given range query. Assume that a five-dimensional index is built ($d = 5$) but that the query is restricted to two dimensions. In this case qd is set to 2. The size of the query box in the first qd dimensions is set to $q_i = qs^{\frac{1}{qd}}$ for all $1 \leq i \leq qd$. The size of the query box in the remaining dimensions is set at $q_i = 1$ for all $qd < i \leq d$. This means there are no restrictions or predicates in the remaining dimensions. Example:

Assuming that the size of the query box is $qs = 0.04$ and the query box dimensions parameter is $qd = 2$. Then the shape of the query box is calculated with the above given rule as $q = (0.2, 0.2, 1, 1, 1)$. This limits the model to certain shapes of query boxes, but it allows the model to work with scalar values as input parameters.

It is assumed that the locations of the query boxes are uniformly distributed over the data space. If this assumption does not hold the model can be adapted as presented in the PISA model in Chapter 6.

7.2.3. System specific parameters

System specific parameters are parameters which are chosen by the database administrator (DBA) of a system. We assume that the DBMS has its own access to the disk system and does not use the I/O functions of the operating system:

Blocksize. The blocksize $b \in \mathbb{N}$ is the number of KB that are read with one disk access. Whenever data is requested from secondary memory at least one whole block is read and is transfered. The size of this block is given by the blocksize b in KB.

Scale factor. Due to the fact that the access time, and not the available disk space, is the limiting factor, controlled redundancy of stored data is accepted in order to be more time efficient. This is especially true in the context of data warehouse systems where materialized views occupy a large portion of disk space. In general, the more data is materialized the more queries are answered without accessing base data and the faster the system responses become. Some index structures have the same property in that they can trade space and time. For example, the more space is occupied by bitmap indexes, the more time efficient structures are generated. We use bitmap indexes that are *time optimal under given space constraints* [Chan and Ioannidis, 1998]. Bitmap indexes are characterized by a scale factor $sf \in \mathbb{R}$ times the space occupied by the tree structures. *E.g.* a scale factor of $sf = 2$ means that the bitmap indexes can occupy twice the space used by trees.

7.2.4. Disk specific parameters

Besides inner nodes of R^*-trees index structures are usually not stored in main memory but in secondary memory. If a query is processed with an index structure, that structure (or part of it) has to be read from secondary memory and transfered into main memory. Many approaches compare index structures by only counting the number of external I/Os. Our approach does not neglect the fact that reading blocks sequentially is much faster than reading blocks randomly.

The behavior of disks is modeled by two parameters:

Bandwidth. The Bandwidth $bw \in \mathbb{R}^+$ of a disk is the speed [MB/s] with which the disk can read data and transfer it into main memory. Since data is stored more and more densely on the disks over time, this speed increases by approximately 40 % per year [Bitton and Gray, 1998], [Patterson and Keeton, 1998]).

Latency time. The second parameter is the average time t_l of positioning the read/write heads and to start reading the needed data. This time is the latency time $t_l \in \mathbb{R}$ of a disk system. On page 16 we defined latency time=SeekTime+RotationTime/2. This parameter depends mainly on the rotation speed of the disk. The rotation speed does not increase at the same rate as the bandwidth bw. It increases only by approximately 8 % per year [Bitton and Gray, 1998], [Patterson and Keeton, 1998].

The time for a random disk access t_r and the time for a sequential disk access is calculated from the above parameters as shown in Section 3.3 on page 16. The fact that the bandwidth bw increases much faster than the latency time t_l, decreases, widens the gap between a sequential and a random block access. With today's (2000) disk technology, using a reasonable large blocksize it is ten to twenty times faster to read a sequential block than a random block. In five years, this factor will probably be increased by 30 to 70. One can argue that by then index structures will only be of limited use, because sequential scans will be faster for most queries than using index structures. This is true if the amount of data is kept fixed. But the capacity of disks (and the amount of stored data) increases even faster than the bandwidth. Therefore, the time for scanning a whole disk increases, and it will still be necessary to index data. However, the index structures will have to adapt according to the changed parameters.

7.2.5. Configuration

For our experimental setups we group the above defined nine parameters $d, t, c, qs, qd, b, sf, bw, t_l$ together to a vector of nine parameters

$$e = (d, t, c, qs, qd, b, sf, bw, t_l) \tag{7.1}$$

Let us call a specific vector e a *configuration*. There are two restrictions between the parameters. The number of dimensions d of the data space must be larger than or equal to the number qd of dimensions in which the query box is restricted. The number of indexed tuples must be larger or equal than the cardinality c of attributes. In the remaining part of this chapter only configurations in which the restrictions $qd \leq d$ and $c \leq t$ are considered.

For each parameter in e a set of values is defined. *E. g.* for the blocksize b, let $B = \{4, 8, 16\}$. For the other parameters, value sets are defined similarly and denoted by capitalized letters. The set of all configurations is defined as:

$$\begin{aligned} E = & \{(d, t, c, qs, qd, b, sf, bw, t_l) \\ & \in D \times Nt \times C \times Q_s \times Q_d \times B \times SF \times BW \times T_l | (qd \leq d) \wedge (c \leq t)\} \end{aligned} \tag{7.2}$$

In the following experiments the parameters bw and t_l are kept constant for one set of parameter values. The set BW and the set T_l contains one value each.

7.3. Index structures and time estimators

Our goal is to develop techniques for the comparison of index structures. This Section describes four for index structure models which are used for the comparison. We apply two tree-based index structures and the two bitmap index structures. We limit ourself firstly to the size of the index structure if the number of tuples and the cardinality of data which has to be indexed is given. Then we calculate the time which is needed to process range queries. For each index structure we define a function to estimate the time used for processing a given range query in a given configuration. This chapter applies ($s = 4$) index structures. Therefore s different functions $t_i : E \rightarrow \mathbb{R}^+, i \in \{1, \cdots, s\}$ are defined to estimate the time for query processing:

$$t_i(e) = \begin{array}{l}\textbf{time for processing range query in configuration } e \\ \textbf{with index structure } i\end{array}$$

Next we show how the functions t_i are calculated for the four index structures which we apply in this chapter.

7.3.1. Time Measures for tree-based index structures

First we apply a tree-based index structure without aggregated data in the inner nodes. We calculate the space needed for storing a tree-based index structure. Since there are much more leaf nodes than inner nodes (inner nodes occupy less than 2 % of the disk space in our experiments) we consider only leaf nodes. The number of leaf nodes is the same for structures that use aggregated data and for structures that omit materialized aggregates in the inner nodes. The space s_{data} (in Byte) needed by one entry of a leaf node depends on the cardinality of the attribute c_j and the dimensionality d of the data. In addition, a pointer (*Tuple IDendtifier*= 4 Bytes) to the data itself is stored:

$$s_{data} = \frac{\sum_{j=1}^{d} \lceil \log_2 c_j \rceil}{8} + \underbrace{4}_{tid} \tag{7.3}$$

The maximum fanout of data pages or leaf pages depends on the chosen blocksize b and on the size of the data entries s_{data}. The greater the blocksize, the more data entries are stored on each block. This maximum number of entries B_{leaf} per block is given by:

$$B_{leaf} = \left\lfloor \frac{b * 1024}{s_{data}} \right\rfloor \tag{7.4}$$

We define the number of data nodes n (leaf nodes) necessary to store all data entries as the quotient of the number of tuples t that are indexed and the number of data entries B_{leaf} fitting into one block:

$$n = \left\lceil \frac{t}{B_{leaf}} \right\rceil \tag{7.5}$$

Here we assume that all nodes are filled with the maximum fanout. This is achieved if a bottom-up structure like the STR-tree [Leutenegger et al., 1997] or packed R-tree [Roussopoulos and Leifker, 1985] is used. Therefore, for other structures like the R^*-tree B_{leaf} has to be set to the average number of entries per leaf node. For the R^*-tree this is approximately 70 % of the maximum fanout.

We assume that blocks are not stored on the disk in a specific ordering. Each page access to disk requires one random access. The number of necessary disk accesses can be computed by measures presented in Chapter 6. The time estimation $t_1 : E \to \mathbb{R}$ for the tree *without* aggregated data is the expected number of necessary page accesses $Inter(q)$ multiplied by the time needed for one random access t_r. The calculation of the form of a range query q depends on parameters qs and qd of configuration e as presented on page 90. Given this vector q and the other parameters of the tree-based index structure we define the time from the given configuration e as the product of expected number of disk accesses times random access time:

$$t_1(e) = Inter(q) * t_r \tag{7.6}$$

Chapter 5 described the idea of aggregated data in the inner nodes of an index structure in detail. If aggregated data is used, there is no access necessary to rectangles completely contained in the query box. $Border(q)$ is the number of leaf nodes which have to be accessed when *aggregated* data is used. Then the expected time $t_2(e)$ for the tree-based structure with use of aggregated data is the product of the number of accessed pages and the time for a random block access t_r:

$$t_2(e) = Border(q) * t_r \tag{7.7}$$

We use the two measures t_1 and t_2 to estimate the time for a given configuration to process a query with tree structures. There are approaches in which blocks are stored in some ordering (*e. g.* Hilbert-curve or the Z-curve). For few dimensions (two to three) this reduces the number of random block accesses because blocks are read sequentially. However, this effect is only effective for few dimensions and for a high number of dimensions this effect can be neglected.

7.3.2. Time measures for bitmap indexing techniques

This section investigates bitmap indexing techniques. Bitmaps indexes perform differently from tree-based indexing techniques. Details of bitmap indexing techniques are presented in Section 3.6. This chapter uses multi-component equality encoded bitmap indexes and multi-component range based encoded bitmap indexes. For simplicity, we use the terms equality encoded and range encoded here.

Here, we concentrate on *time optimal bitmap indexes under a given space constraint* [Chan and Ioannidis, 1998]. To compare bitmap index structures with tree structures, we assume that the space constraint depends on the space the tree structure needs multiplied by scale factor sf. First the size of each bitmap vector (*e.g.* size of B_1 in Figure 3.10 on page 28) is determined. The number of blocks necessary for storing one bitmap vector depends on the number of tuples t and on the blocksize b:

$$v = \left\lceil \frac{t}{8 * 1024 * b} \right\rceil \tag{7.8}$$

This model assumes that the size of the space which is occupied by bitmap vectors is proportional to the space needed by tree structures. Let m denote the number of bitmap vectors that are stored by the system. This value of m depends on the blocks allocated by the tree structure and a scale factor *sf*, which is one of our input parameters:

$$m = \left\lfloor \frac{n * sf}{v} \right\rfloor \tag{7.9}$$

The space constraint m gives the maximum number of bitmap vectors for all dimensions together. We split the *global* m into separate m_j for each dimension $j = 1, 2, \cdots, d$ with $\sum_{j=1}^{d} m_j \leq m$ weighted by the cardinality c_j:

$$m_j = \left\lfloor m \frac{\log c_j}{\sum_{j=1}^{d} \log\ c_j} \right\rfloor \tag{7.10}$$

The m_j's are used to calculate the base of the encoded bitmap indexes in each dimension. For equality and range encoded bitmap indexing techniques we get different structures. Therefore, we have to distinguish between the bases for the two distinct bitmap indexing techniques. Having the m_j and c_j at hand, we calculate the bases of the equality encoded bitmap index as shown in Figure 3.12 on page 30.

Estimator B_{equal} in Equation 3.9 on page 33 calculates the number of bitmap vectors which are read when processing a range query with equality encoded bitmap indexes. The first block access is a random block access while the other block accesses are read sequentially. The time for processing the range query is calculated by:

$$t_3(e) = (t_r + (v-1)t_s) * B_{equal} \tag{7.11}$$

Figure 3.15 on page 32 shows the calculation of the base of the bitmap index with given m_j for range encoded bitmap indexes. Equation 3.17 on page 33 defines for a given base the number of bitmap indexes which are read. Given the estimator for the

Table 7.1.: Functions to estimate the processing time using various index structures

Function	Index Structure
t_1	R-tree without aggregated data
t_2	R-tree with aggregated data
t_3	Equality encoded bitmap index
t_4	Range encoded bitmap index

range encoded bitmap vectors as B_{range}, we calculate the time needed for the range encoded bitmap index as:

$$t_4(e) = (t_r + (v-1)t_s) * B_{range} \tag{7.12}$$

This section defined four functions t_1, t_2, t_3, and t_4 to calculate the time for processing a configuration e with an index structure. In Table 7.1 the four functions are summarized. The functions t_1 and t_2 provide functions of the expected time for processing range queries using trees. The functions t_3 and t_4 define the expected time for access the data with bitmap indexing techniques.

7.4. Classification trees

This section applies classification trees to get information about the most important parameters which influence the performance of index structures. Classification trees are important tools in detecting latent structures in data [Venables and Ripley, 1994]. Classification trees use a set of objects or tuples, a set of one or more classification variables, and one response variable [Breimann et al., 1984]. A classification tree can be reviewed as a hierarchical collection of rules: For example, for tuple $a \in O$ the rule set may look like:

`if` $(x \leq 3)$ `and` $(y \leq 5)$ `then` a belongs most likely to group A
`if` $(x \leq 3)$ `and` $(y > 5)$ `then` a belongs most likely to group B
`if` $(x > 3)$ `then` a belongs most likely to group C

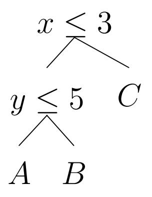


Figure 7.1.: Simple example of a classification tree

In this example are x and y the classification variables, a is a tuple or data item and the response variable can take values in $O = \{A, B, C\}$. Figure 7.1 displays the rules in the form of a binary tree. When constructing a classification tree there are two contradicting objectives. First, a classification tree should be rather small in terms of terminal nodes. This objective yields to too general trees. The extreme case is a generated tree consisting of only one terminal node. The second objective is to classify as many data items correct as possible. This objective leads to rather large trees. In the extreme case a tree has one different terminal node for each data item. Such extreme classification trees are rather useless. The main goal is to find an appropriate tradeoff between the size of the tree and its accuracy measured as a misclassification rate.

7.4.1. Applied methods

This section describes how classification trees are applied for comparing index structures. For each configuration $e \in E$, the expected times are computed for processing the specified query for s tree structures. This is done by applying the functions $t_i : E \rightarrow \mathbb{R}$ defined in Section 7.3. The best (fastest) index of all structures is selected by a function $s_{min} : E \rightarrow \{1, \cdots, s\}$ which is defined as:

$$s_{min}(e) = \min\{i \in \{1, \cdots, s\} | t_i(e) \leq t_j(e) \quad \forall j \in \{1, \cdots, s\}\} \tag{7.13}$$

For each configuration $e \in E$, function $s_{min} : E \rightarrow \{1, \cdots, s\}$ selects the fastest index structure. The input set for the generation of the classification tree consists of ten-dimensional vectors. The first nine values are the parameters defined by e. The last value is the response variable and corresponds to the index of the fastest index structure. Formally we define this set G for constructing the classification tree as:

$$G = \{(\underbrace{d, t, c, qs, qd, b, sf, bw, t_l}_{e}, s_{min}(e)) | e \in E\}$$

Table 7.2 shows an example of the set G. This set G is the input for the statistical software package S-Plus.

7.4.2. Value sets of Parameters

Table,7.3 shows the different experimental parameter sets. Altogether there are 42,336 cases considered for building a classification tree. This are fewer cases than for the aggregation technique presented in Table 7.4 because the software package S-Plus cannot handle such a large number of cases in a reasonable time.

We fix some parameters before running the software. The minimum number of elements of a node of a classification tree before the node is split is set to minsize=10. The minimum number of elements per node after a split is set to mincut=5.

Table 7.2.: Set G for generation classification tree

d	t	c	qs	dq	b	sf	bw	t_l	$s_{min(e)}$
1	10^6	3	10^{-7}	1	4	1	60	6	0
1	10^6	3	10^{-7}	1	4	2	60	6	0
1	10^6	3	10^{-7}	1	4	3	60	6	0
1	10^6	3	10^{-7}	1	8	1	60	6	0
1	10^6	3	10^{-7}	1	8	2	60	6	0
1	10^6	3	10^{-7}	1	8	3	60	6	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 7.3.: Parameter sets for experiments

Name	Variable name	Set name	Set of different values
Dimensions	d	D	$\{1,2,3,4,5,6,7\}$
Tuples	t	N_t	$\{10^6,10^7,10^8,10^9\}$
Cardinality	c	C	$\{3,10,100,10^3,10^4,10^5,10^6\}$
Query box size	qs	Q_s	$\{10^{-7},10^{-6},\cdots,10^{-2}\}$
Query box dimensions	qd	Q_d	$\{1,2,3,4,5,6,7\}$
Block size [KB]	b	B	$\{4,8,16\}$
Scale factor	sf	SF	$\{1,2,3\}$
Bandwidth	bw	BW	today (2000): 11 MB/sec, in 5 years: 60 MB/sec
Latency time	t_l	T_l	today (2000): 6 ms, in 5 years: 4 ms

7.4.3. Results

The number of misclassified data items depends on the size of the tree. The more leaf nodes the tree consists of, the more data items are classified correctly. The smaller the tree is, the more data items are misclassified. Figure 7.2 shows the tradeoff between the number of misclassified data items and the size of the tree in number of leaf nodes. The x-axis shows the number of leaf nodes and the y-axis the number of misclassified data items. For 45 to 60 terminal nodes 6209 data items or 14.6 % of all data items are misclassified. If the tree is pruned to 32 leaf nodes, the misclassification rate increases to 16 % and for a tree with only 20 terminal nodes the misclassification rate is about 20.6 %.

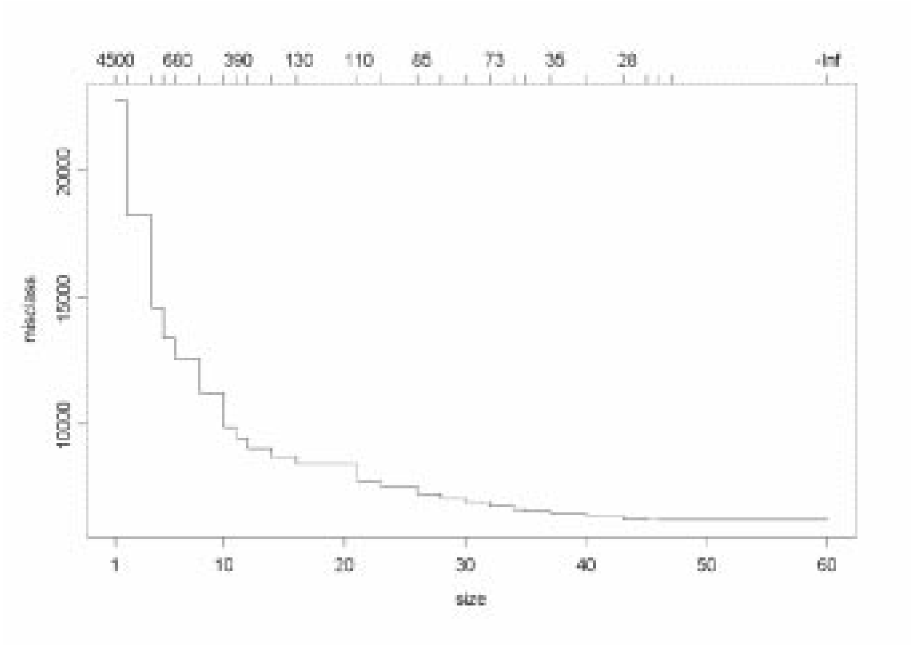


Figure 7.2.: Size of tree vs. misclassification of cases

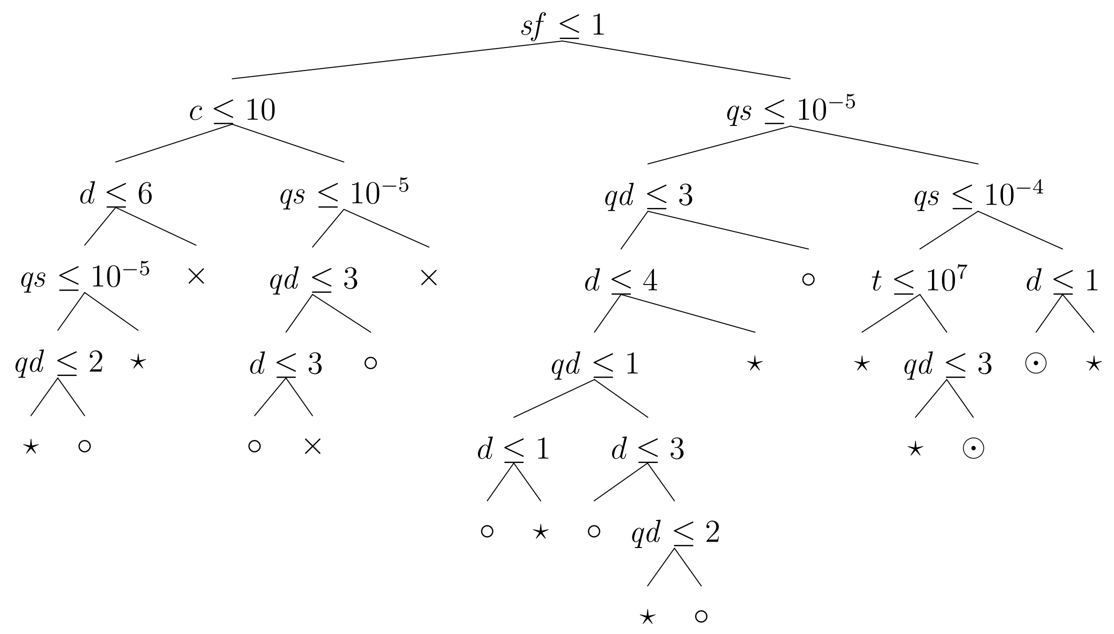
In Figure 7.3 the generated classification tree with 20 leaf nodes is plotted. The 19 inner nodes guide the path to the leaf nodes. From this type of tree we extract two kinds of knowledge.

1. Rules: For this experimental setup we get 20 rules. We pick out the following rule for example:

$$\texttt{If } sf > 1 \texttt{ and } q_s > 10^{-4} \texttt{ and } d \leq 1 \Rightarrow R_a^*\texttt{-tree} \odot$$

2. Importance of parameters: If a specific case $e \in E$ satisfies the precondition ($sf > 1$ and $q_s > 10^{-4}$ and $d \leq 1$), the R_a^*-tree is the fastest structure.

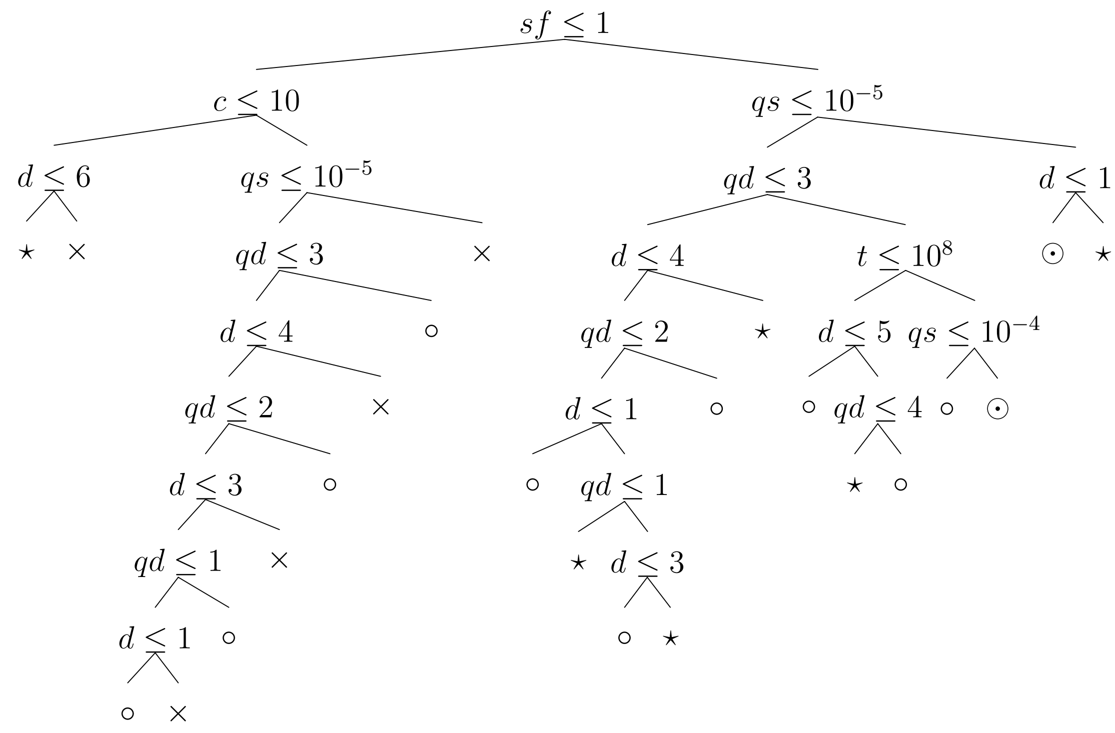
However, the classification tree contains additional information. The occurrence of the parameters in the inner nodes of the tree give a hint of to what degree the parameters influence the behavior of the tree structure. One interesting result of the



Index structure	Symbol
R-tree without aggregated data	$\circ$
R-tree with aggregated data	$\odot$
equality encoded bitmap index	$\times$
range encoded bitmap index	$\star$

Figure 7.3.: Classification tree for year 2000 data

application of the classification tree is the fact that the blocksize is not used in any nodes of the classification tree. The blocksize is assumed to be an important parameter in DBMSs. However, in these experiments the blocksize is of limited importance. The bitmaps are not very sensitive to changes of the blocksize. Therefore, a changed blocksize yields to minor changes of the performance of bitmaps. For tree structures, the blocksize is an important factor. However, if the trees are applied for very small range queries (nearly point queries) a query is processed by accessing one or very few leaf nodes. In this case the blocksize is not important. If the query box is large and several blocks are read, the bitmaps are more efficient than the trees. Therefore, we conclude that the blocksize does influence the behavior of the structures, but not change the relative performance between the structures.



Index structure	Symbol
R-tree without aggregated data	$\circ$
R-tree with aggregated data	$\odot$
equality encoded bitmap index	$\times$
range encoded bitmap index	$\star$

Figure 7.4.: Classification tree for year 2005 data

Figure 7.4 shows a classification tree which is generated with data for disks assumed to be available in five years. One interesting result of this tree is the fact that in

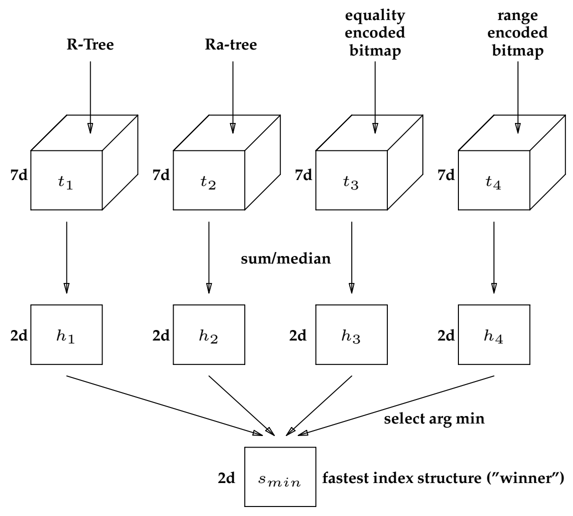


Figure 7.5.: Sum aggregation technique

more than two-third of all inner nodes the variables for the number of dimensions d or the variable for the query box dimensions qd occurs. In five years the variables d and qd influence the relative performance of the structures more than other parameters. Therefore, these parameters have to be carefully investigated.

Classification trees have one drawback. The input for the classification algorithm is the information which index structure is the fastest for a certain configuration. There is no information about the distance between the best solution and its competitors.

7.5. Statistics in two dimensions

This section describes an aggregation technique for comparing different index structures by keeping two parameters fixed [Jürgens and Lenz, 1999b]. Having defined the s functions from Section 7.3, the technique works in the following way: The two parameters bw and t_l are selected for one experiment. We create for each structure a seven-dimensional cube using the above defined functions t_i. Each cell of the seven-dimensional data cube stores the expected time for processing a range query for a given configuration. The seven-dimensional data is mapped to two-dimensional data by using statistical aggregation functions. Aggregation functions like *sum*, *min*, *max*, *count*, and *median* can be used as a measure of 'location' or 'mean' behaviour of an index structure. Next, we describe the *sum*, *median*, and *count* aggregation in detail.

7.5.1. Sum aggregation

Figure 7.5 sketches how the sum aggregation is implemented. From each of the s different seven-dimensional data cubes, we generate a two-dimensional data cube by applying the aggregation function *sum*. Two dimensions are preselected into which we aggregate the data. Assume that the number of dimensions d and blocksize b are selected and that they are kept fixed. The aggregation is done by functions h_i, $i \in \{1, \cdots, s\}$ as follows:

$$h_i(d', b') = \sum_{(e \in E) \wedge (d = d') \wedge (b = b')} t_i(\underbrace{d, t, c, qs, qd, b, sf, bw, t_l}_{e}) \forall (d', b') \in D \times B \tag{7.14}$$

The *sum* function is proportional to the average case, because the number of cases is constant for all index structures. From the s two-dimensional data cubes, a two-dimensional cube is computed. Each cell of the resulting two-dimensional cube is computed by applying the function $s_{min} : D \times B \rightarrow \{1, \cdots, s\}$. The function s_{min} selects the index of the structure with the smallest value (shortest processing time). Function s_{min} is defined as:

$$s_{min}(d', b') = \min\{i \in \{1, \cdots, s\} | h_i(d', b') \leq h_j(d', b') \ \forall j \in \{1, \cdots, s\}\} \forall (d', b') \in D \times B \tag{7.15}$$

Functions like *min* or *max* can be used similarly as the *sum* function in Equation 7.14. An optimistic user may use the *min* function. A decision based on the *min* function assumes always the best case. A pessimistic user applies the *max* function.

7.5.2. Median aggregation

The median aggregation method is similar to the *sum* aggregation method. The main difference is the definition of function h_i in Equation 7.14. Instead of calculating the *sum* of all values the *median* as a less sensitive statistic (cf. [Huber, 1981]) is selected:

$$\begin{aligned} h_i(d', b') &= \text{Median}\{t_i(\underbrace{d, t, c, qs, qd, b, sf, bw, t_l}_{e}) | e \in E \wedge (d = d') \wedge (b = b')\} \\ & \forall (d', b') \in D \times B \end{aligned}$$

The second part of the aggregation methods works similar to the *sum* aggregation.

7.5.3. Count aggregation

A third aggregation technique is the count aggregation. This technique is implemented differently from the two previously described approaches. Figure 7.6 sketches the count aggregation technique. The count aggregation generates from s seven-dimensional data cubes one seven-dimensional data cube by selecting the index of the cube with the minimum value. Function s_{min} implements this selection:

$$s_{min}(e) = \min\{i | t_i(e) \leq t_j(e) \ \forall j \in \{1, \cdots, s\}\} \tag{7.16}$$

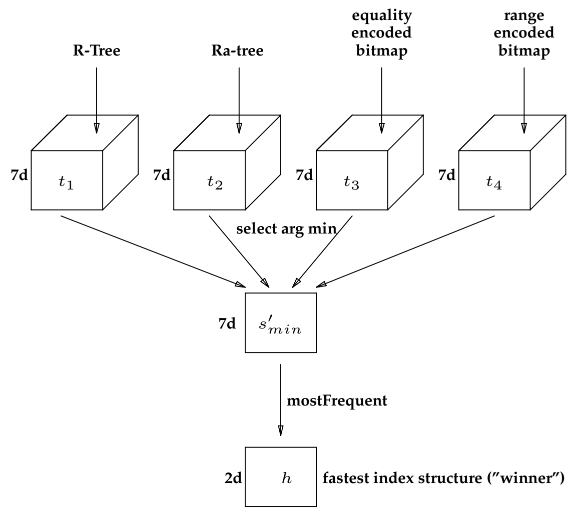


Figure 7.6.: Count aggregation technique

From this seven dimensional cube one two-dimensional cube is generated by selecting the most frequent value for each subset.

$$h_i(d', b') = \text{mostFrequent}\{s_{min}(e) | e \in E \wedge e \in E \wedge d' = d \wedge b' = b\} \tag{7.17}$$

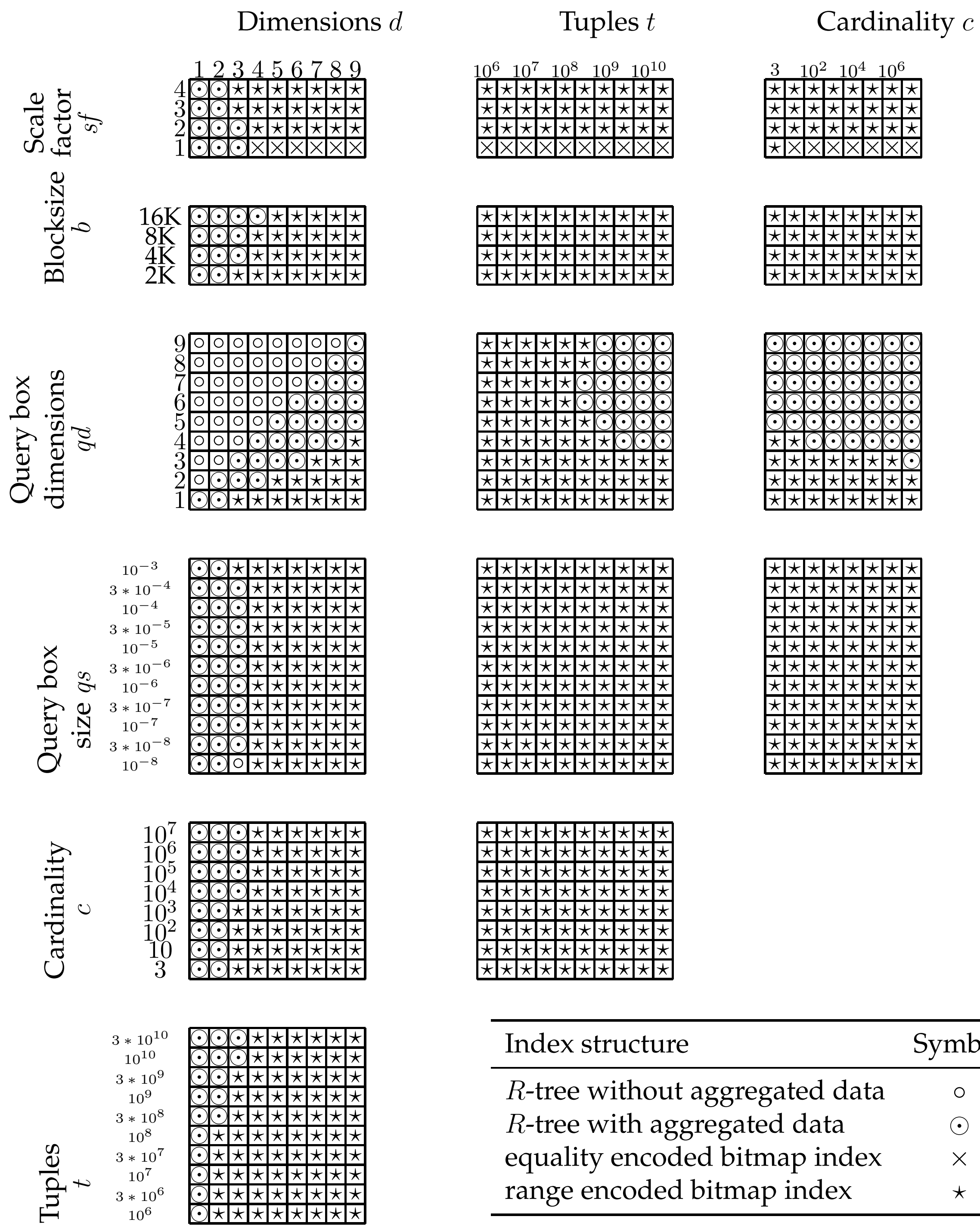
The count aggregation is less sensitive against extreme values, because all configurations are weighted equally.

Table 7.4.: Parameter sets for experiments

Name	Variable name	Set name	Set of different values
Dimensions	d	D	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Tuples	t	N_t	$\{10^6, 3 * 10^6, 10^7, \cdots, 3 * 10^{10}\}$
Cardinality	c	C	$\{3, 10, 100, 10^3, 10^4, 10^5, 10^6, 10^7\}$
Query box size	qs	Q_s	$\{10^{-8}, 3 * 10^{-8}, 10^{-7}, \cdots, 10^{-3}\}$
Query box dimensions	qd	Q_d	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Blocksize [KB]	b	B	$\{2, 4, 8, 16\}$
Scale factor	sf	SF	$\{1, 2, 3, 4\}$
Bandwidth	bw	BW	today (2000): 11 MB/sec, in 5 years: 60 MB/sec
Latency time	t_l	T_l	today (2000): 6 ms, in 5 years: 4 ms

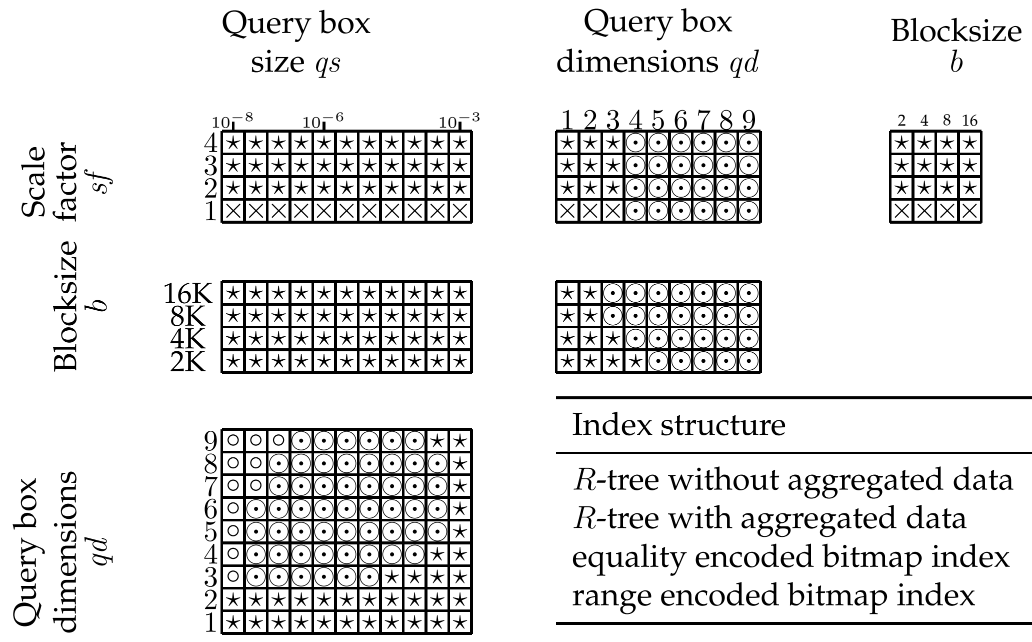
7.5.4. Results

This section presents results of experiments applying the aggregation technique. The results of the median aggregation method and count aggregation method are similar to the *sum* aggregation method. Therefore, we present only results of sum aggregation.



Index structure	Symbol
R-tree without aggregated data	$\circ$
R-tree with aggregated data	$\odot$
equality encoded bitmap index	$\times$
range encoded bitmap index	$\star$

Figure 7.7.: Results I: Sum aggregation technique (year 2000)



Index structure	Symbol
R-tree without aggregated data	$\circ$
R-tree with aggregated data	$\odot$
equality encoded bitmap index	$\times$
range encoded bitmap index	$\star$

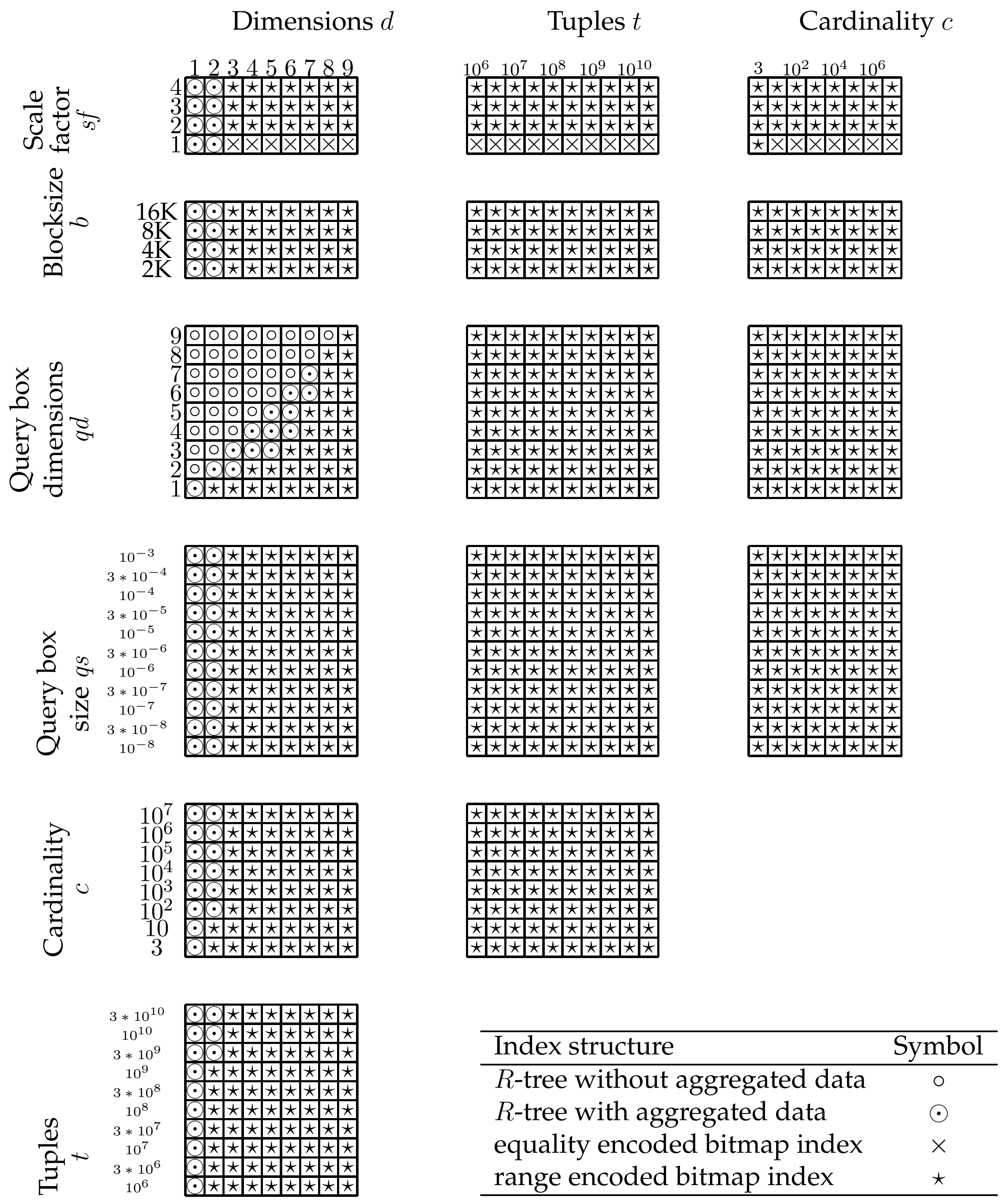
Figure 7.8.: Results II: Sum aggregation technique (year 2000)

In the experiments the bandwidth bw and the latency time t_l are set to fixed values. We vary the remaining seven parameters and all possible combinations of the sets in Table 7.4 (under the constraint $(qd \leq d) \wedge (c \leq t)$). This yields in 617,760 different combinations for each index structure. For each of these combinations the four functions t_i are evaluated. Then we apply the aggregation as described previously. There are $\frac{n(n-1)}{2} = 21$ different possibilities, on how to aggregate the seven-dimensional data into two-dimensional data. All two-dimensional results are presented here for today's disk systems and for disk system expected in five years. The parameters for the disk are the parameters of a Seagate Cheetah 18. The latency time t_l is set to 6 ms and the bandwidth bw is set to 11 MB/sec [Patterson and Keeton, 1998]. Some of the results are discussed next.

On the very left of the topmost line of pictures in Figure 7.7 the data is aggregated according to the number of dimensions and the scale factor. This graph shows that for more than three dimensions the bitmap indexes perform faster than the tree-based index structures. For two or less dimensions the tree structures with aggregated data are best.

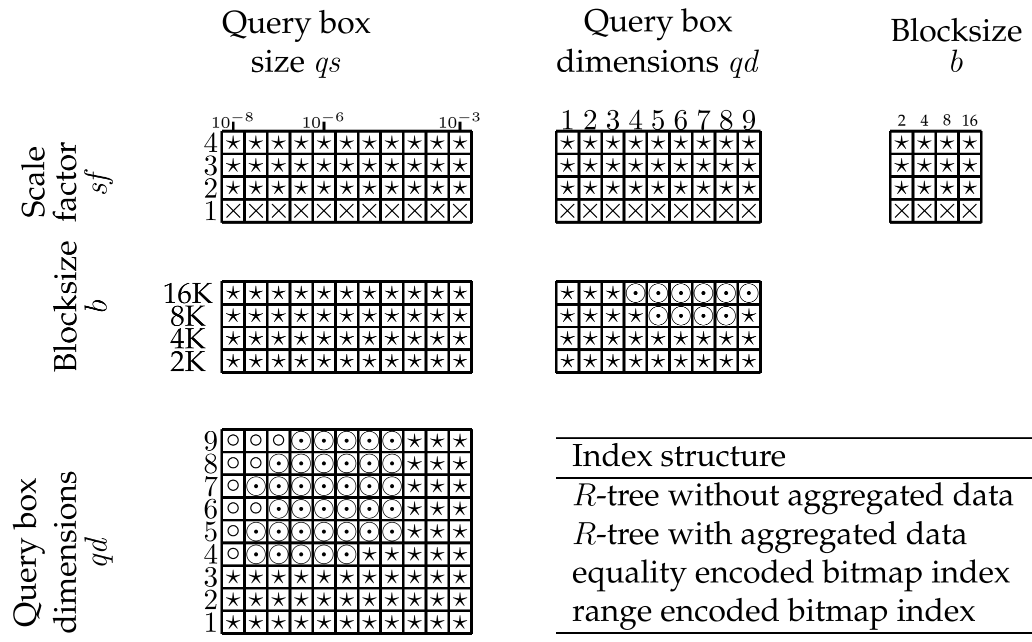
The right most picture in the third row in Figure 7.7 compares the cardinality c and the number of query box dimensions qd. This picture shows that for queries which are restricted in more than four dimensions the tree-based index structures with aggregated data are well suited. If queries are restricted in only two to three dimensions, bitmap indexes are superior.

The very left picture in the fifth row in Figure 7.7 compares the number of dimensions d and the attribute cardinality c. It can be seen that for more than 2 to 3 dimensions the bitmaps are better than the tree structures.



Index structure	Symbol
R-tree without aggregated data	$\circ$
R-tree with aggregated data	$\odot$
equality encoded bitmap index	$\times$
range encoded bitmap index	$\star$

Figure 7.9.: Results III: Sum aggregation technique (year 2005)



Index structure	Symbol
R-tree without aggregated data	$\circ$
R-tree with aggregated data	$\odot$
equality encoded bitmap index	$\times$
range encoded bitmap index	$\star$

Figure 7.10.: Results IV: Sum aggregation technique (year 2005)

In Figure 7.8 the third row compares the query box size qs and the number of query box dimensions qd. If only one or two attributes occur in the range query, the bitmap indexes outperform the trees. If the query box is very small (nearly a point query) aggregated data in inner nodes of the trees cannot be used and the tree without aggregated data works most efficiently for $qd > 2$. If the query box size is increased, the tree with aggregated data becomes superior. However, for very large query boxes, the range encoded bitmap works best.

Figure 7.9 and Figure 7.10 show the results of the same experiments as shown in Figure 7.7 and Figure 7.8, but the bandwidth bw and the latency time t_l are changed. In the area of new computer technology it is very difficult and risky to make any predictions for the future. If we assume that the bandwidth bw increases by 40 % each year and the latency time t_l is decreasing by only 8 % per year (like bw and t_l did during the last years [Bitton and Gray, 1998], [Patterson and Keeton, 1998]), the models we presented here can be used to predict the performance of index structures with new disk technology. Here, we extrapolate this trend, present results for disk systems expected to be available in five years and compare them with results of today's disk systems.

Figure 7.9 and Figure 7.10 show that the bitmaps gain advantages over the tree-based indexing techniques with the use of future disk technology. In the next years, in many more cases range encoded bitmap indexes are faster than its competitors than today. This trend becomes evident if the number of dimensions is considered. Comparing the first column of pictures in Figure 7.7 with the first column of pictures in Figure 7.9 shows that with 2000s disk drives the bitmap are better than the trees for at least three dimensions. With the expected disk technology of 2005 the bitmaps are in many cases better than tree-based index structures.

Bitmaps are gaining advantages in comparison to trees, because they read large blocks of data. Trees access only small blocks and suffer from long latency times t_l.

7.6. Summary

For data warehouses fast access to large sets of data is crucial. Index structures support query processing. Many parameters influence the performance of index structures. Here we concentrate on a set of nine parameters. We present two techniques to compare different index structures for the use in a data warehouses. Classification trees generate rules to evaluate which index structure is suited best for a specific experimental setup. Further results are information about what parameters influence the performance of index structures most. One evidence is, that the chosen blocksize for a database is only of limited influence, but the scale factor *sf* is very important, because it is first selected feature in the induced classification trees. Statistical function and aggregation methods show in which cases specific index structures outperform other index structures. In addition, we show that due to changes in disk technology, bitmap indexing techniques will gain advantages over the traditional tree-based index structures in the future.