Tree Based Indexes vs. Bitmap Indexes: A Performance Study

Marcus Jürgens
Institute of Statistics and Econometrics
Institute of Computer Science
Freie Universität Berlin, Germany
juergens@inf.fu-berlin.de

Hans-J. Lenz
Institute of Statistics and Econometrics
Freie Universität Berlin, Germany
hjlenz@wiwiss.fu-berlin.de

Abstract

Data warehouses are used to store large amounts of data. This data is often used for On-Line Analytical Processing (OLAP). Short response times are essential for on-line decision support. Common approaches to reach this goal in read-mostly environments are the precomputation of materialized views and the use of index structures. In this paper, a framework is presented to evaluate different index structures analytically depending on nine parameters for the use in a data warehouse environment. The framework is applied to four different index structures to evaluate which structure works best for range queries. We show that all parameters influence the performance. Additionally, we show why bitmap index structures use modern disks better than traditional tree structures and why bitmaps will supplant the tree based index structures in the future.

1 Introduction

Data warehouse and OLAP applications differ very much from the traditional database applications. Traditional database systems are OLTP oriented and access few tuples for read/write access at the same time. In data warehouse environments the data is used for decision support and large sets of data are read and usually not changed. Therefore, it is possible to materialize views in advance [GHRU97] and use different access paths to support fast access to multidimensional data [OQ97]. One of the most popular multidimensional index structures is the R-tree [Gut84]. During the last 15 years many scientists improved the R-tree resulting in structures like the R^*-tree [BKSS90], STR-tree [GLE97], packed R-tree [RL85], Hilbert R-tree and [IK94]. In case of range queries on aggregated data, the R_a^*-tree proved to be a promising structure [JL98]. However, tree structures have one drawback. It is a well-known fact that tree structures degenerate when the number of dimensions is increased. Usually in data warehouse and OLAP applications many dimensions have to be considered. Another class of index structures, the bitmap indexes, try to overcome the problem to generate when the number of dimensions is increased by storing the data of each dimension separately and allowing fast access to the dimensions that are needed to answer certain queries. Bitmap indexing techniques are implemented in some commercial available database management systems (e. g. from Informix [Inf97] and Sybase [Syb97]). The question arises as to which structure is best suited for certain applications.

In this paper a new framework is described to support the comparison of different index structures. Most investigations of index structures only consider one or two parameters. However, the performance of index structures depends on many different parameters. Here, we concentrate on a set of nine different parameters to compare index structures used for a database management system. The framework is applied to investigate four different index structures. Two tree based index structures and two bitmap structures are used to evaluate which index structure works best for given range queries. Our investigations show that all presented parameters influence the performance of the used index structures. We show that taking just one or two dimensions into account is not sufficient for a thorough comparison of different indexing techniques. Furthermore, we show that bitmap index techniques will outperform the tree based index structures in many cases with future disk technology.

The rest of the paper is organized as follows: Section 2

Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'99)
Heidelberg, Germany, 14. - 15.6. 1999
(S. Gatziu, M. Jeusfeld, M. Staudt, Y. Vassiliou, eds.)
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-19/

defines the used parameters that influence the performance of index structures. Section 3 explains the framework and defines the rules of how the high dimensional data is aggregated to two-dimensional data. In Section 4, four different index structures are modeled, and it is shown how these structures are applied in the framework. Section 5 describes the experiments and results of the application of the framework. The last section summarizes the approach and gives an outlook.

2 Input parameters

There are nine parameters which influence the performance of index structures. We group them into four different categories: data specific, query specific, system specific, and disk specific parameters. First we describe the different groups with their parameters.

2.1 Data specific parameters

Data specific parameters describe the data that has to be indexed. The three important parameters are:

Number of dimensions. The dimensionality $d \in \mathbb{N}$ of the data denotes the number of attributes. This parameter is important for the performance of a system. For the task of indexing eight dimensional data a different structure is better suited than for indexing one or two-dimensional data.

Number of stored tuples. The number t of tuples influences the performance of different structures. In [JL98] it is shown that the R_a^*-tree improves in comparison with the R^*-tree if the number of indexed tuples is increased.

Cardinality of data space. The cardinality of the range of an attribute c is the number of different values an attribute may have. It is obvious that for attributes like gender, where there are at most three possible values (male, female, NULL) different index structures are better than for attributes like social security number or telephone number, where the attributes may have millions of different values. In the following, we assume that each attribute can have c different real numbers in the range of $[0, 1)$. It is assumed that the attribute cardinality is the same in all dimensions. This assumption makes the model simple and can be relaxed later to make the experiments more realistic. The cardinality of the range of the jth attribute is denoted by c_j.

Another parameter may be the distribution type of data (e. g. normally distributed or uniformly distributed data). To keep the framework simple, this parameter is not considered here, but the framework can easily be extended to take the actual distribution of data into account. In this case the models for the tree structures can be changed according to the PISA model [JL99]. In the following, uniformly distributed data is assumed. The models for the bitmaps are not affected by other distributions of data.

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 can be expressed as range queries with query box size $q_s = 0$. In order to have only scalar values as parameters, we assume range queries that can be described with the two scalar values:

Query box size. The size of the query box is given proportionate to the size of the data space and is denoted by q_s ($0 \leq q_s \leq 1$). A value of $q_s = 0.04$ means that the query box fills 4 percent of the data space.

Query box dimensions. The query box dimension parameter q_d denotes the number of attributes occurring in a given range query. Assume a five dimensional index is built ($d = 5$), but the query is only restricted to two dimensions. In this case q_d is set to 2. The size of the query box in the first q_d dimensions is $q_i = q_s^{\frac{1}{q_d}}$ for all $1 \leq i \leq q_d$. The size of the query box in the remaining dimensions will be set at $q_i = 1$ for all $q_d < i \leq d$. This means there are no restrictions or predicates in the remaining dimensions. Assuming that the size of the query box is $q_s = 0.04$ and the query box dimensions is $q_d = 2$ 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 positions of the query boxes are uniformly distributed over the data space.

2.3 System specific parameters

System specific parameters are parameters that can be 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 bytes that is read with one disk access.

Scale factor. Due to the fact that the access time, and not the available disk space, is the limiting factor, redundancy of stored data is accepted in order to be more time efficient. This is especially true in the context of data warehouses where materialized views occupy a large portion of disk space. The more data is materialized the more queries can be answered without accessing base data and the faster the systems are. Some

index structures have the same property that they can trade space and time. In this approach we use bitmap indexes that are *time optimal under given space constraints*. Here, more space is added to the bitmap indexes than space is occupied by the investigated tree structures. E. g. a scale factor of $sf = 2$ means that the bitmaps can occupy twice the space used by the trees.

2.4 Disk specific parameters

Index structures are usually not stored in main memory, but in the secondary memory and have to be read from secondary memory and transfered into main memory. In contrast to many approaches that compare index structures and only count the number of external I/Os and neglect the fact that blocks can be read sequentially much faster than reading blocks randomly, we take this fact into account and model it. Here the behavior of disks are modeled by two parameters:

Bandwidth. The bandwidth bw of a disk is the speed [MB/s] in which the disk can read data and transfer it into main memory. Due to the fact that the disks are getting (physically) smaller and the data is stored more densely on the disks, this speed increases by approximately 40 % per year [BG98], [PK98].

Latency time. The second parameter is the average time the read/write heads need to get to a certain position and to start reading the desired data. This time is the latency time t_l of a disk system. On average, this is the SeekTime+RotationTime/2. This parameter depends mainly on the rotation speed of the disk. The rotation speed does not increase with the same rate as the bandwith bw. It increases only by approximately 8 % per year [BG98], [PK98].

From the bandwidth bw and the blocksize b, the time t_s for one sequential access can be calculated as the time for reading and transferring one block of data into main memory

$$t_s = \frac{b}{bw} \tag{1}$$

The time t_r for a random block access is calculated as the time for bringing the read/write head to a certain position plus the time to read one block of data and transferring it into main memory

the gap between tl and b/bw is widening

$$t_r = t_l + \frac{b}{bw} \tag{2}$$

The fact that the bandwidth bw is increasing much faster than the latency time t_l is decreasing, widens the gap between a sequential and a random access. With todays (1999) disks it is (with a reasonable large block size) ten to twenty times faster to read a sequential block than a random block. In five years, this factor will probably be increased to 36 to 72. One could argue that by this time index structures will only be of limited use, because sequential scans will be faster for most queries than the use of index structures. This is true if the amount of data is kept fixed. But the capacity of disks (and the amount of stored data) is increasing even faster than the bandwidth. Therefore, the time for scanning a whole disk will increase, and it will still be necessary to index the data. However, the index structures in the future have to be better according to the changed parameters than structures used today. The described framework supports the investigation as to which index structure is best suited for a certain application and given parameters.

bw is increasing faster than tl, and therefore sequential scans are much better random scans on disk.

3 The Framework

In the following section, a framework is described to compare different index structures in respect to all parameters.

3.1 Configuration

Grouping the above defined parameters together, we get a vector of nine scalar parameters

$$e = (d, t, c, q_s, q_d, b, sf, bw, t_l) \tag{3}$$

to characterize a certain experimental setup. We call a specific vector e a *configuration*. There is one dependency between the parameters. The number of dimensions of the data space must be greater than or equal to the number of dimensions in that the query box is restricted. In the rest of the paper, we will consider only configurations in which $q_d \leq d$.

For each parameter a set of values is defined. E. g. for the blocksize, a set of values is defined as $B = \{2048, 4096, 8192, 16384\}$. For the other parameters, sets are defined similarly and denoted by capitalized letters. The set of all configurations is defined as

$$E = \{(d, t, c, q_s, q_d, b, sf, bw, t_l) \in$$
$$D \times T \times C \times Q_s \times Q_d \times B \times SF \times BW \times T_l | q_d \leq d\}$$

In the following experiments the parameters bw and t_l are assumed to be constant for one experiment. For each index structure that has to be investigated a function is needed, to estimate the time used for processing a given range query for a given configuration. Assuming s index structures have to be investigated, s different functions $t_i : E \to \mathbb{R}, i \in \{1, \cdots, s\}$ have to be defined to estimate the time that is needed to process a query with a given configuration.

$t_i(e)$ = time for processing range query in configuration e with index structure i

Having defined the s functions, the framework works in the following way. The two parameters bw and t_l are assumed to be fixed for one experiment and for each structure

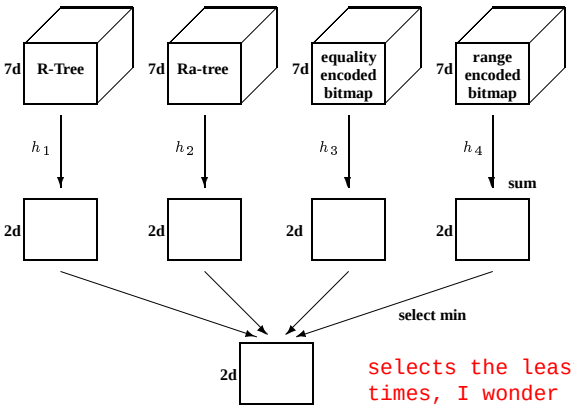


Figure 1: Framework for sum aggregation

a seven dimensional cube is built using the above defined functions t_i. In each cell, the time that is needed to process a range query for a given configuration is stored. Seven dimensional data is difficult to visualize. Therefore, the seven dimensional data is mapped to two-dimensional data by using aggregation functions. The aggregate functions: *sum*, *min*, *max*, *count*, and *median* can be used. Due to space limitations only the *sum* aggregation is discussed here.

3.2 Sum aggregation

Figure 1 shows how the sum aggregation works. For each of the s different seven dimensional data cubes, a two-dimensional data cube is generated by applying the aggregation function *sum*. Two dimensions have to be selected in which the data should be projected. Assume that the number of dimensions d and blocksize b are kept fixed. The aggregation is done by functions h_i, $i \in \{1, \cdots, s\}$

$$h_i(d', b') = \sum_{(e \in E) \wedge (d=d') \wedge (b=b')} t_i(d, t, c, q_s, q_d, b, sf, bw, t_l)$$

is it grouping by these dimensions to turn it into a 2d tbl

The *sum* function calculates the average case, because the number of cases is equal for all index structures. From the s two-dimensional data cubes one two-dimensional cube s_{min} is computed which gives the number of the index structure with the smallest value. Function $s_{min} : D \times B \rightarrow \{1, \cdots, s\}$ has to satisfy the following condition:

$$\forall i \in \{1, \cdots, s\} : h_{s_{min}(d',b')}(d', b') \leq h_i(d', b') \quad (4)$$

Functions like *min* or *max* can be used similarly. If the user is very optimistic, she may use the *min* function. If the user is very pessimistic, the *max* function should be applied.

Two other aggregation approaches are investigated. In one approach, the data is aggregated by counting the number of cases in which each index structure is the best (fastest) one. Another approach is to calculate the median. The results of the two other approaches are just a little bit different from the *sum*-aggregation method. Therefore, and because of space limitations, the other approaches are not further presented here.

4 Application of the framework to compare four index structures

To show how the framework works, the framework is applied to compare different index structures. The space used and the time t_i used for processing a range query by each index structure is computed.

4.1 Estimators for the tree based index structures

Here, we calculate the space needed for building up a tree based index structure. Due to the fact that there are many more leaf nodes than inner nodes (inner nodes occupy less than 2 % 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 neglect aggregates in the inner nodes.

The space (in bytes) used by one entry of a leaf node depends on the cardinality of the attributes c_j and the number of dimensions d that have to be stored. In addition, there has to be a pointer (TID) to the data itself.

size is sum of 1 to last dim's spaces in bits required

$$s_{data} = \frac{\sum_{j=1}^{d} \lceil \log_2 c_j \rceil}{8} + \underbrace{4}_{pointer} \quad (5)$$

The maximum fanout of data pages depends on the chosen blocksize b and on the size of the data entries s_{data}. The bigger the blocksize, the more data entries can be stored on each block. The exact quantity M_{data} is calculated by

$$M_{data} = \left\lfloor \frac{b}{s_{data}} \right\rfloor \quad (6)$$

The number of data nodes N (leaf nodes) that are necessary to store all data entries can be calculated as the quotient of the number of tuples that have to be indexed and the number of data entries that fit into one block

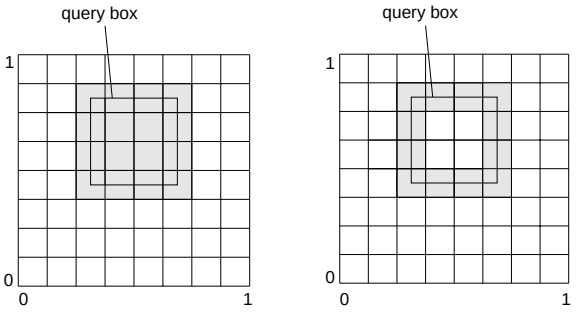
$$N = \left\lceil \frac{t}{M_{data}} \right\rceil \quad (7)$$

Here we assume that all nodes are filled with the maximum fanout. This can be achieved if a bottom-up structure like the STR-tree [GLE97] or packed R-tree [RL85] is used.

Under the assumption that the points are uniformly distributed over the data space, the average length of a data rectangle $\overline{r}$ is given by (cf. [JL99])

$$\overline{r} = \frac{1}{\sqrt[d]{N}}, \quad (8)$$

In the rest of the paper, the GRID Model with uniformly distributed data as described in [JL99] is used. The number



a) Without use of aggregated data

b) With use of aggregated data

Figure 2: $d = 2$, $N = 64$, and $\overline{r} = \frac{1}{8}$

of pages that have to be accessed on leaf node level can be calculated as

$$E_w = \prod_{j=1}^{d} \min\left\{\frac{q_j}{\overline{r}} + 1, \frac{1}{\overline{r}}\right\} \tag{9}$$

where the query box is expressed as a tuple $q = (q_1, q_2, \cdots, q_d)$ as described before.

In Figure 2a), there are 64 data nodes assumed to be uniformly distributed over the two-dimensional data space $[0,1)^2$. Therefore, the length of one rectangle is: $\overline{r} = \frac{1}{8}$. The rectangle q represents the query box $q = (0.5, 0.5)$. All gray shaded rectangles in Figure 2a) represent leaf nodes that have to be accessed when an index structure like the STR-tree is used. In Figure 2a), these are 16 blocks. The pages on the leaf node level are stored in random order on the disk. In the best case, they are stored according to one dimension or according to a space filling curve [IK94]. These savings decrease dramatically when the number of dimensions increases. We assume that the blocks are not ordered. For each page access to disk, one random access is necessary. The time estimation t_1 for the tree *without* aggregated data is the number of necessary page accesses multiplied by the time used for one random access

$$t_1(e) = E_w t_r \tag{10}$$

The idea of aggregated data in the inner nodes of an index structure is described in detail in [JL98]. The inner nodes store in addition to the reference to its successors some aggregated data about their successors (count and sum in this example, cf. Figure 3). If aggregated data is used, there is no access necessary to rectangles completely contained in the query box. This number is calculated by

$$E_c = \prod_{j=1}^{d} \max\left\{\frac{q_j}{\overline{r}} - 1, 0\right\} \tag{11}$$

All pages that contain data that is completely contained in the query box do not have to be accessed. In the example in Figure 2b), access to four blocks is saved. The number

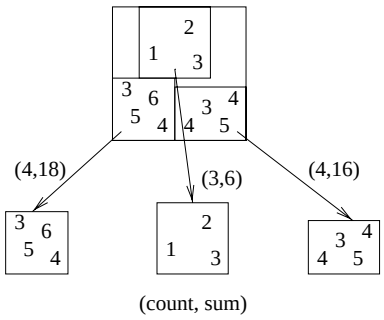


Figure 3: Aggregated data in index structure

E_a of leaf nodes that have to be accessed when *aggregated* data is used is

$$E_a = E_w - E_c \tag{12}$$

The time estimation for the tree with use of *aggregated* data is the product of the number of accessed pages and the time for each random block access

$$t_2(e) = E_a t_r \tag{13}$$

The two time estimators t_1 and t_2 are used in the framework to estimate the time for a given configuration to process a query if uniformly distributed data is assumed.

4.2 Estimators for bitmap indexing techniques

First we present the general idea of bitmap indexing and show how space and time can be traded. Assume there is a column x in a relational DBMS with 6 different values from 0 to 5. The traditional bitmaps generate one bitmap vector ($B_0 - B_5$) for each of the 6 different values as shown in Table 1. If the value of x is set to 3, the corresponding bit in vector B_3 is set to 1. Otherwise the bit is set to 0. This index structure has the great disadvantage, that one bitmap vector is necessary for each value of x. Therefore, this structure can only be used for attributes that have only few different extensions.

The next two index structures that are used for the application of the framework are equality and range encoded bitmap indexes. In this example equality encoded bitmap indexes convert the value of x in a number system with base 3 of the first and base 2 of the second digit: $x = 2 * y + z$, where y is represented by B_2^1, B_1^1, B_0^1 and z is represented by B_1^0 and B_0^0 (cf. Table 1). Range encoded bitmaps are optimized for range queries. They can be calculated from the equality encoded indexes by $\overline{B}_i^j = \overline{B}_{i-1}^j \wedge B_i^j$. In [CI98], the authors show how time and space are traded in detail. They focus on four different points of this trade: time optimal, space optimal, "knee", and time optimal under given space constraint. Here we concentrate on *time optimal bitmap indexes under a given space constraint.* To compare the bitmap structures with tree structures, we assume that

value	traditional bitmap						equality encoded					range encoded		
	B_5	B_4	B_3	B_2	B_1	B_0	B_2^1	B_1^1	B_0^1	B_1^0	B_0^0	$\overline{B}_1^1$	$\overline{B}_0^1$	$\overline{B}_0^0$
1	0	0	0	0	1	0	0	0	1	1	0	1	1	0
3	0	0	1	0	0	0	0	1	0	1	0	1	0	0
0	0	0	0	0	0	1	0	0	1	0	1	1	1	1
5	1	0	0	0	0	0	1	0	0	1	0	0	0	0
4	0	1	0	0	0	0	1	0	0	0	1	0	0	1
2	0	0	0	1	0	0	0	1	0	0	1	1	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
							$\underbrace{\quad\quad\quad}_{y}$			$\underbrace{\quad\quad}_{z}$				

Table 1: Different bitmap indexing techniques

the space constraint depends on the space the tree structure needs to store the data multiplied by scale factor *sf*. First the size of each bitmap vector (e. g. B_1) is calculated. The number of blocks needed to store one bitmap vector is given by

$$v = \left\lceil \frac{t}{8b} \right\rceil \tag{14}$$

Let M denote the number of bitmap vectors that can be stored by the system. In this model it is assumed, that the space for bitmap vectors is proportional to the space needed by tree structures. Therefore, M will be set dependent on the blocks allocated by the tree structure and a scale factor *sf*, which is one of the input parameters of our model.

$$M = \left\lceil \frac{N}{v} sf \right\rceil \tag{15}$$

The space constraint M is given for all dimensions together. We have now to split the *global* M down into separate M_j for each dimension with $\sum_{j=1}^{d} M_j \leq M$. This split is done weighted by the different c_j.

$$M_j = \left\lfloor M \frac{\log c_j}{\sum_{j=1}^{d} \log\ c_j} \right\rfloor \tag{16}$$

The M_j's can be used to calculate the base of the encoded bitmap indexes in each dimension. Note that in our examples the c_j will all have the same value, but this can not be assumed in general. For equality and range encoded bitmap indexing techniques we get different structures. Therefore, we have to distinguish between the two bitmap indexing techniques. Having the M_j and c_j at hand, the bases of the index can be calculated as shown in Figure 4.

For an equality encoded bitmap index, the algorithm to calculate the base is sketched in Figure 4. This algorithm has to be performed for each $j \in \{1, \cdots, d\}$. An additional optimization step (not shown here), improves the performance of the bitmap index structures [CI98]. The base in

```
(1) n_j = 0
(2) repeat
(3)         n_j = n_j + 1
(4)         b_j = ⌊M_j / n_j⌋ + 1
(5)         r_j = (M_j + n_j) mod n_j
(6) until b_j^{r_j} (b_j − 1)^{n_j − r_j} ≥ c_j
```

Figure 4: Equality encoded Calculation of base for bitmap indexes for given M_j and c_j

```
(1) n_j = 0
(2) repeat
(3)         n_j = n_j + 1
(4)         b_j = ⌊M_j / n_j⌋ + 1
(5)         r_j = (M_j + n_j) mod n_j
(6) until (b_j + 1)^{r_j} b_j^{n_j − r_j} ≥ c_j
```

Figure 5: Range encoded calculation of base for bitmap indexes for given M_j and c_j

each dimension is given by

$$< \underbrace{b_{ji} - 1, \cdots, b_{ji} - 1}_{n_{ji} - r_{ji}}, \underbrace{b_{ji}, \cdots, b_{ji}}_{r_{ji}} > \tag{17}$$

In the rest of the paper, the base is denoted as

$$< b_{j1}, b_{j2}, \cdots, b_{jn_j} > \tag{18}$$

For example: b_{23} denotes the base of the third component of the second attribute.

If range encoded bitmap indexing techniques are used, the base of the index is calculated by the algorithm in Figure 5. The algorithm in Figure 5 has to be executed for each $j, 1 \leq j \leq d$. The base in each dimension is given by

$$< \underbrace{b_i, \cdots, b_i}_{n_i - r_i}, \underbrace{b_i + 1, \cdots, b_i + 1}_{r_i} > \tag{19}$$

With the given bases for the equality and range encoded bitmap indexes it is possible to estimate the time needed by both structures. The number of bitmaps that have to be scanned for a specific configuration according to [CI98] is

$$E_e = \sum_{j=1}^{d} \sum_{i=1}^{n_j} E_{r_j,i}$$

$$E_{r_j,i} = \begin{cases} \frac{1}{b_{ji}} \left(\left\lfloor \frac{b_{ji}}{2} \right\rfloor^2 + (b_i - 1)\left(\left\lceil \frac{b_{ji}}{2} \right\rceil - \frac{b_{ji}}{2} \right) \right) & : \quad b_{ji} > 2 \\ 1 & : \quad \text{otherwise} \end{cases}$$

The details can be found in [CI98]. From the number of bitmap vectors that have to be scanned and the size of the bitmap vectors in blocks the time estimate for the equality encoded bitmap index is calculated. The first block of a vector needs a random block access, the remaining $v-1$ accesses can be read sequentially if the blocks of each bitmap vector are stored sequentially

$$t_3(e) = (t_r + (v-1)t_s)E_e \tag{20}$$

If range encoded bitmap indexing technique is used, the number of bitmap vectors that have to be scanned is given by

$$E_r = \sum_{j=1}^{d} 2 \left(\frac{(n_j - r_j)(b_j - 1)}{b_j} + \frac{r_j b_j}{b_j + 1} \right) \tag{21}$$

The time to execute one range query with range encoded bitmap vectors is given by

$$t_4(e) = (t_r + (v-1)t_s)E_r \tag{22}$$

The estimators t_3 and t_4 are used to estimate the time the different bitmap indexing techniques need to access the data. Together with the functions t_1 and t_2 this set of functions provides the base data for the framework.

5 Experiments and Results

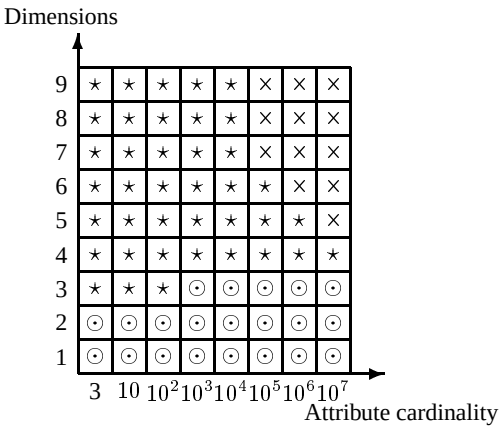
In this section results of experiments where the framework has been applied are presented. In the following experiments the bandwidth bw and the latency time t_l are set to fixed values. The remaining seven parameters are varied and all possible combinations of the sets in Table 2 are considered (under the constraint $q_d \leq d$). This yields in 475.200 different combinations for each index structure. For each of these combinations the four functions t_i are evaluated. Then the framework is applied as described in Chapter 4. There are $\frac{n(n-1)}{2} = 21$ different possibilities how to aggregate the seven dimensional space into a two-dimensional space. Due to space limitations only four two-dimensional results are presented here for todays disk systems and two results for disk system as expected in five years are shown.

The parameters for the disk are chosen, as using a Seagate Cheetah 18. The latency time t_l is set to 6 ms and the bandwith bw is set to 11 MB/sec [PK98]. In Figure 6a) the number of dimensions d and the attribute cardinality c are compared. It can be seen that for more than 2 to 3 dimensions the bitmaps are better than the tree structures. If the attribute cardinality is increased, it is faster to take the equality encoded bitmap index than the range encoded bitmap index. In Figure 6c) the query box size q_s and the number of query box dimensions q_d are compared. 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 can not be used and the tree without aggregated data works most efficient. When the query box size is increased, the tree with aggregated data becomes superior. For very large query boxes, the range encoded bitmap works best.

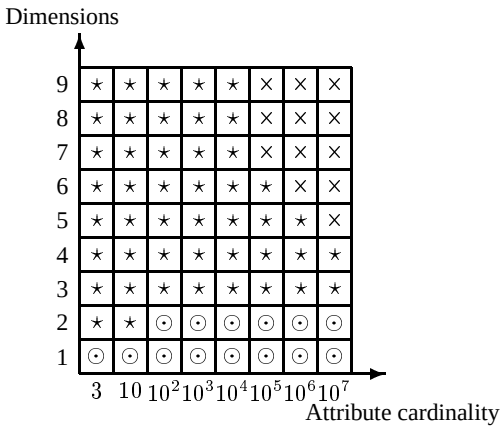
In Figure 6e) the blocksize b and the number of dimensions d are compared. As expected, the trees have some advantages when the blocksize b is increased because bigger blocks yield in a higher fanout and therefore in less random block accesses. In Figure 6f) the scale factor sf and the number of dimensions d is varied. As in the Figure 6e), the bitmaps become faster with an increasing number of dimensions. A large scale factor sf favors the range encoded bitmap index in comparison to the equality encoded bitmap index. Generally speaking, the range encoded index seems to be better than the equality encoded index. In [CI98] the authors came to the same results with different experiments. This shows that our results are really meaningful. However, our method is more general and considers more parameters.

In the field of new computer technology it is very difficult and risky to make any predictions for the future. But if we assume that the bandwidth bw is increasing by 40 % each year and the latency time t_l is decreasing by only 8 % per year (like they did during the last years), the presented models can be used to predict the performance of index structures in the future. Here we give some results for disk systems expected to be available in five years and compare them with results of todays disk systems.

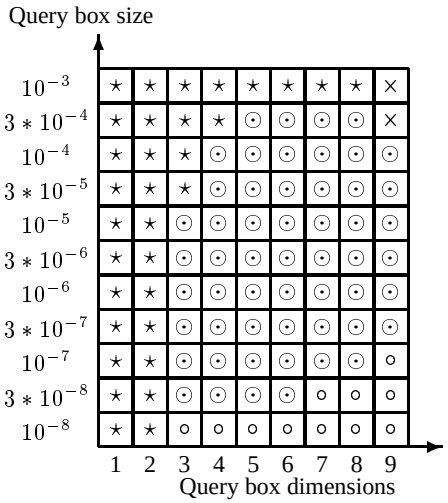
In Figure 6a) and in Figure 6b) the results of experiments are shown where only the latency time t_l and bandwith bw is changed. By comparing the two figures it can be seen that bitmaps will get superior in comparison to trees for more than 2 dimensions. Figure 6b) and Figure 6d) show the results for query box dimensions and query box size for todays disk systems and for disks available in 5 years. In this example, it can be seen that the bitmaps gain advantages over the tree based indexing techniques with the use of future disks, too. Other results show the same tendency.



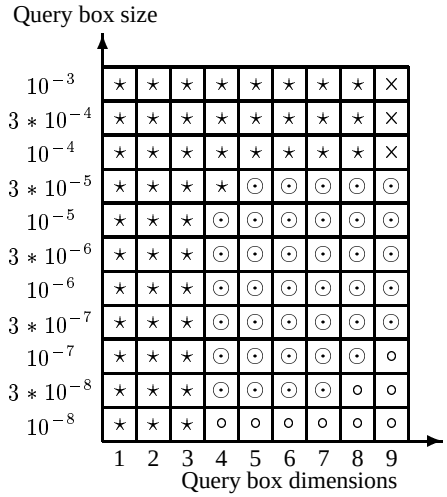
a) Attribute cardinality vs. Dimensions (now)



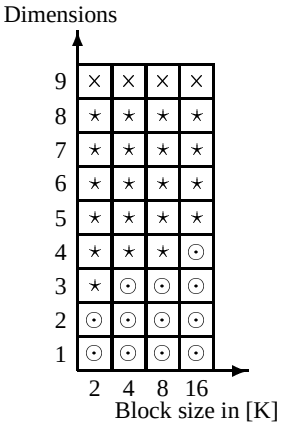
b) Attribute cardinality vs. Dimensions (in five years)



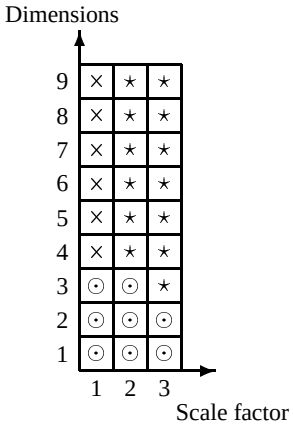
c) Query box dimensions vs. query box size (now)



d) Query box dimensions vs. query box size (in five years)



e) Block size vs. dimensions (now)



f) Scale factor vs. dimensions (now)

g) Legend

index structure	symbol in figure
STR-tree without aggregated data	∘
STR-tree with aggregated data	⊙
equality encoded bitmap index	×
range encoded bitmap index	⋆

Figure 6: Results of experiments

Table 2: Parameter sets for experiments

Name	Set name	Set of different values
Dimensions	D	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Tuples	T	$\{10^6, 3 * 10^6, 10^7, \cdots, 3 * 10^{10}\}$
Cardinality	C	$\{3, 10, 100, 10^3, 10^4, 10^5, 10^6, 10^7\}$
Query box size	Q_s	$\{10^{-8}, 3 * 10^{-8}, 10^{-7}, \cdots, 10^{-3}\}$
Query box dimensions	Q_d	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Block size [KB]	B	$\{2, 4, 8, 16\}$
Scale factor	SF	$\{1, 2, 3\}$
Latency time	BW	today: 6 ms, in 5 years: 4 ms
Bandwidth	T_l	today: 11 MB/sec, in 5 years: 60 MB/sec

6 Summary & Outlook

In the field of data warehouses fast access to large amounts of data is crucial. Many index structures support fast access to OLTP data, but perform poorly in read-mostly environments when aggregated data over large sets of data has to be calculated. We have presented a framework to compare different index structures for the use in a data warehouse environment. The framework has been applied to compare four different index structures depending on nine parameters. We have shown that all parameters influence the results and therefore should be taken into account when comparing index structures. In addition, we have shown that due to changes in disk technology, bitmap indexing techniques will probably outperform the traditional tree based index structures in the future.

Acknowledgments

Part of this research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316)

References

[BG98] Dina Bitton and Jim Gray. The rebirth of database machines. invited talk at VLDB, 1998.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331. ACM Press, New York, N. Y., 1990.

[CI98] Chee-Young Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the International Conference on Management of Data*, 1998.

[GHRU97] Himanshu Gupta, Venky Harinaryan, Anand Rajaraman, and Jeffery D. Ullman. Index selection for OLAP. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 208–219, 1997.

[GLE97] Yván J. García, Mario A. López, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 497–506. IEEE Computer Society Press, 1997.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts*, pages 47–57, 1984.

[IK94] Christos Faloutsos Ibrahim Kamell. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20st Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.

[Inf97] Informix. Indexing for the enterprise data warehouse. white paper, 1997. Available at `http://www.informix.com`.

[JL98] Marcus Jürgens and Hans-J. Lenz. The R_a^*-tree: An improved R^*-tree with materialized data for supporting range queries on OLAP-data. In *Proceedings of the International Workshop on Data Warehouse Design and OLAP Technology (DWDOT 98), Vienna*, pages 186–191. IEEE Computer Society Press, 1998.

[JL99] Marcus Jürgens and Hans-J. Lenz. PISA: Performance models of index structures with and without aggregated data. In *Proceedings of*

the 8th International Conference on Statistical and Scientific Database Management (SSDBM), 1999.

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):38–49, 1997.

[PK98] David A. Patterson and Kimberly K. Keeton. Hardware technology trends and database opportunities. invited talk at SIGMOD, 1998.

[RL85] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed R-trees. In *ACM SIGMOD (International Conference on Management of Data)*, pages 17–31, Austin, Texas, May 1985.

[Syb97] Sybase. Adaptive server IQ. white paper, 1997. availabe at `http://www.sybase.com`.