5. Aggregated Data in Tree-Based Index Structures

I never waste memory on things that can easily be stored and retrieved from elsewhere.
Albert Einstein

This chapter describes an approach where aggregated data is materialized in the inner nodes of an index structure. These data could also be calculated from data stored in its successor nodes, but the access to aggregated data in the inner nodes is faster than accesses to successor nodes. This concept of aggregated data in the inner nodes is generic and is applicable to most tree-based index structures. This chapter describes this concept of aggregated data in the inner nodes in detail, and it shows how the algorithms are modified to maintain and use the aggregated data. At the end of this chapter we present results of experiments where the extended structure with aggregated data in the inner nodes is compared with the standard structure without materialized aggregated data.

5.1. Introduction

In this chapter we focus on range queries on aggregated data. Range queries on aggregated data are very common in database applications such as data warehousing. Standard index structures perform poorly for these kinds of queries. We investigate an extension that improves the performance of index structures for most range queries on aggregated data. The processing of point queries will not be improved. A generic extension stores aggregated data in the inner nodes of standard index structures and can be applied for the majority of tree-based index structures. The main idea was presented first as TBSAM for the case of the B-tree [Srivastava et al., 1989].

We assume that a tree-based index structure consists of two different kinds of nodes. Leaf nodes at the bottom of the structure store the indexed attribute values and the references to the data itself (tid-concept) [Härder and Rahm, 1999]. Non-leaf nodes (inner nodes or directory nodes) are all other nodes including the root node. Each leaf node can hold a (not necessarily fixed) number of leaf node entries. The directory nodes of the tree in Figure 5.1 store between two and four directory entries and the leaf nodes can store between two and four data entries. This chapter assumes that the maximum number of directory entries is fixed.

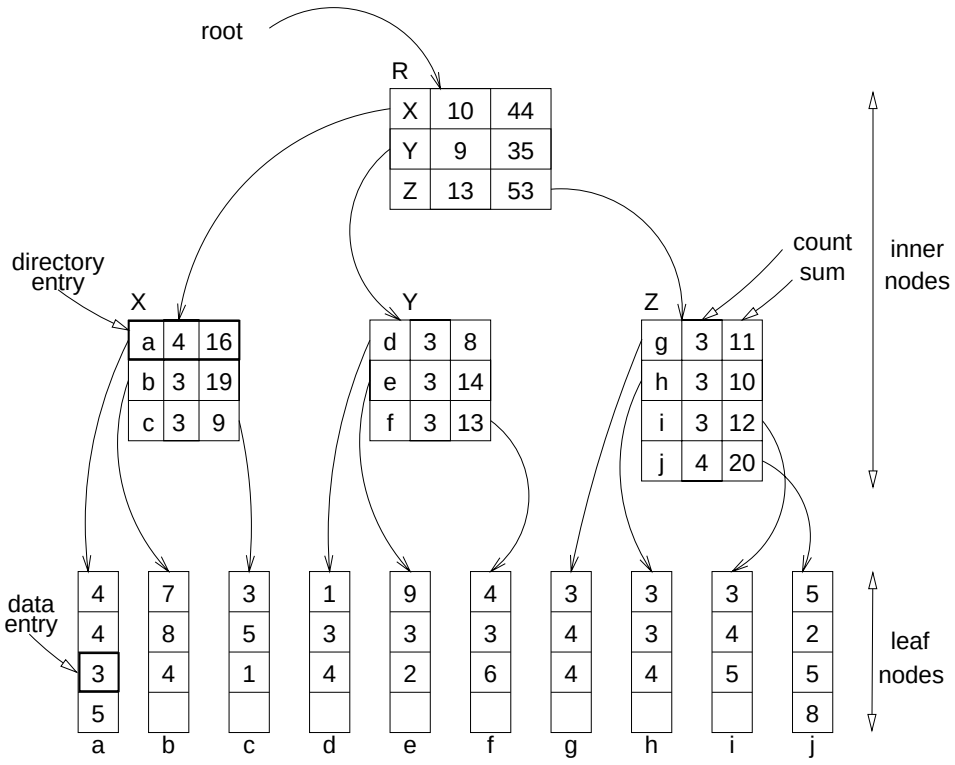

Figure 5.1.: Example of an R^*_a-tree for data in Figure 5.2

We describe how the structure is extended to speed-up the range queries on aggregated data. Only non-leaf nodes respectively inner nodes are modified but this extension does not affect entries on leaf node level. The leaf node entries keep the form $(region, ptr)$. Each non-leaf node can hold a number of directory entries. As described in Chapter 3 in standard index structures without aggregated data, the inner nodes store entries of the form $(region, ptr)$, where ptr stands for pointer and stores the link to the successor node of this entry. The variable $region$ denotes the geometric region that covers all regions of the successor node of ptr. In structures like the R-tree [Guttman, 1984] the regions are d-dimensional hyper-rectangles.

With the extension examined here, entries of the inner nodes of the tree also store aggregated data. Each directory entry is extended from a form $(region, ptr)$ to an entry of the form $(region, agg, ptr)$, where agg refers to the aggregated data. In this thesis the additional storage of the aggregated data will be called the *extension* of the index structure. For the one-dimensional case this idea is presented as TBSAM. An extension of the B-tree is described where aggregated data is stored in the inner nodes to support fast access for statistics like median or other ranking functions. For the multidimensional case this idea was presented in [Ho et al., 1997]. For multidimensional data, only the computation of a certain class of aggregate functions can be supported as we will see later.

We present an example to show how these extensions support range queries on

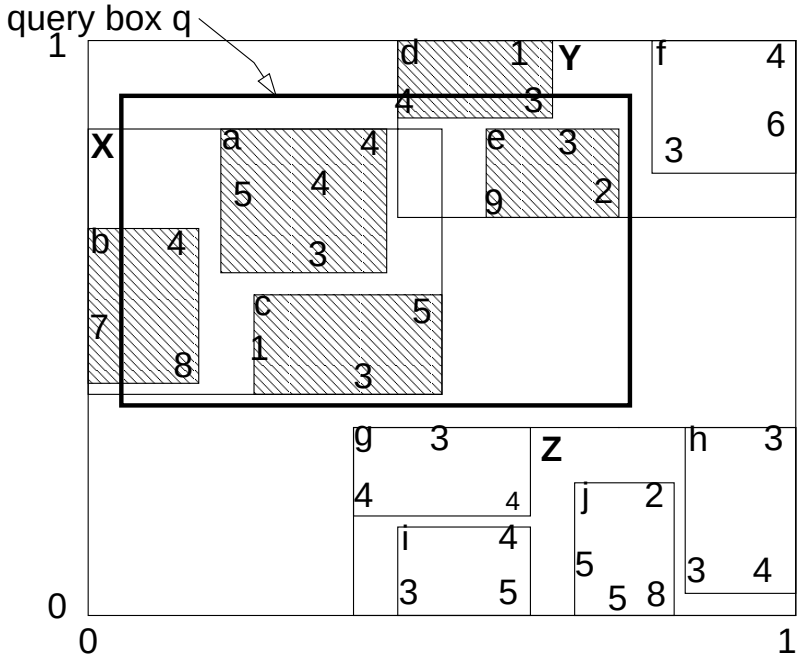

Figure 5.2.: Example data for a tree without using of aggregated data and query box q

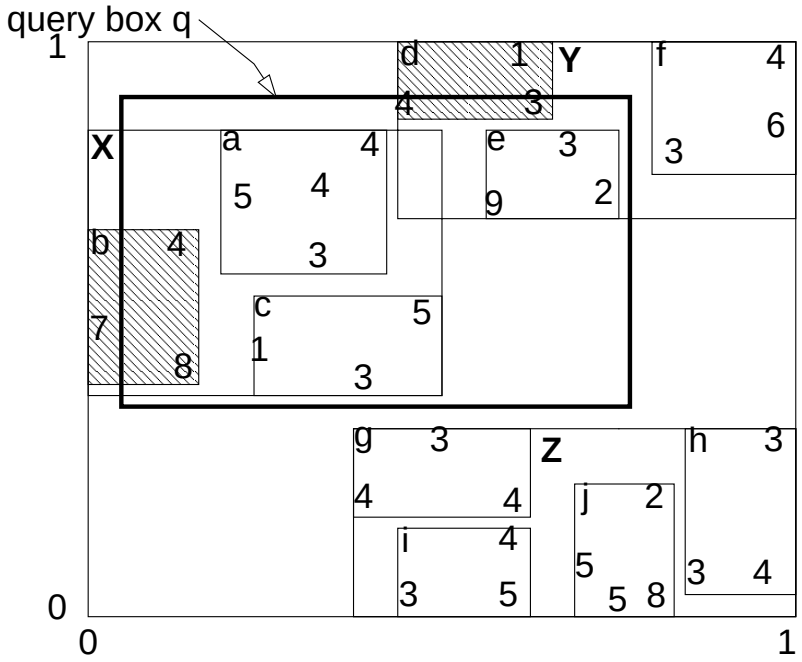

Figure 5.3.: Example data for tree using of aggregated data and query box q

aggregates. Figure 5.1 displays a tree with the aggregates `sum` and `count` attached to the directory entries. Nodes a to j are leaf nodes. Nodes X, Y, and Z are inner nodes on the first level. The root node R is on the second inner level of the tree. Figure 5.2 displays the two-dimensional data stored in this tree. Each digit represents one data item. If a directory entry refers to a *leaf* node, `count` contains the number of data entries of the referred *leaf*. If a directory entry points to a directory node d, `count` stores the `sum` of `count` values of all directory entries of the directory node d.

To see the benefits of this approach we consider range queries on aggregated data. The result of these queries are not specific tuples, but aggregated data over a set of tuples. Assume we compute the number of items contained in the query box q in Figure 5.2. The rectangles of the five hatched nodes a, b, c, d, and e intersect the query box and are accessed assuming that the data is indexed by a structure like the R^*-tree. The number of intersecting rectangles will be expressed by $Inter(q) = 5$. Rectangles a, c, and e are completely contained inside the query box. The number of these completely contained nodes will be abbreviated as $Contain(q) = 3$. The extended index structure can use the aggregated data of its inner nodes that is stored in X for leaf a and c, and in Y for node e. Only the leaf nodes b and d have to be accessed (Figure 5.3). These are the rectangles that intersect the border of the query box. The number of these rectangles is expressed by:

$$Border(q) = Inter(q) - Contain(q) = 5 - 3 = 2. \tag{5.1}$$

We assume that each access to a leaf node corresponds to one random access to secondary memory. Therefore, only 2 accesses instead of 5 accesses are necessary in this example and 60 % of disk accesses is saved.

If necessary, the aggregate `average` is computed from `sum` and `count` in this example. One could try to pre-compute and store as much data as possible in the inner nodes. Adding on more aggregated data to each directory entry increases the size of each directory entry s_{dir}. The bigger the directory entries become, the fewer entries fit onto one block. This results in a smaller fanout of the directory pages. The reduced fanout may increase the height of the trees and may reduce its efficiency and this effect is discussed later in this chapter.

Figure 5.3 shows an example when aggregated data in the inner nodes yield significant performance improvements and how it affects the index structure. The remaining part of this chapter investigates under which conditions this extension of the index structure can be applied and how it influences the index structures behavior (*e. g.* operations and space). There are four aspects that must be considered when applying the extension to an index structure.

The access methods. Which access methods are *fit for aggregation*? The extension of aggregated data in the inner nodes cannot be applied for all access methods.

The aggregation functions. Values of what kind of aggregation functions should be materialized in the inner nodes of the index structure? What kind of functions are supported to compute aggregates from stored data? Not every aggregation function that can be used to pre-compute data for storage in summary

tables can be applied for use to materialize data in the inner nodes of an index structure.

The operations. How must the operations performed on the index structure be changed? Insert, update, and delete operations maintain not only the data in the leaf nodes but also the aggregated data in the inner nodes. Therefore, these operations have to be modified and there is additional overhead. The range query algorithm is also changed to take advantage of aggregated data.

Additional space. How much additional space is needed to store materialized aggregated data? The benefit of having aggregated data in the inner nodes when performing queries needs additional space for redundant data.

5.2. "Fit for aggregation" access method

In the next definition, the requirements are stated which must be fulfilled by a tree-based index structure so that the extension of aggregated data in the inner nodes can be applied.

Definition 5.1 We call an index structure *fit for aggregation* if it fulfills the following four criteria:

1. The index structure consists of the two different kinds of nodes: leaf nodes and non-leaf nodes.

2a. Each leaf node contains data entries $data_{entry} = (region, ptr)$.

2b. Each non-leaf node p contains directory entries $dir_{entry} = (region, ptr)$. Each region covers at least the union of all regions of the entries ptr points to. For each entry $(region, ptr)$ the following relation must be true

$$region \supseteq \bigcup_{dir_{entry} \in ptr} (dir_{entry}.region)$$

3. Each node, except for the root node, has exactly one predecessor.

By $region$ there can be any sub-region of the d-dimensional data space defined. A point can also be modeled as a rectangle with zero area. It is worth mentioning that overlaps between regions of different nodes of the same level are allowed. Furthermore, no fixed size of data entries or fixed block size is assumed here to apply the extension, nor the structure need not to be balanced. However, most tree structures used in DBMSs are balanced structures. It is obvious that for access paths like hashing or bitmaps, this extension cannot be applied. The extension works in the following way: In each entry of a non-leaf node aggregated data is attached about the successors of this entry. The entries are extended to the form $(region, agg, ptr)$. agg is computed by applying an aggregation function f on the data stored in the successors. The extension of the structure is formally defined as follows:

Definition 5.2 An index structure is an **extended index structure** if Definition 5.1 is satisfied and extended by:

2c. Each non-leaf node p stores directory entries $dir_{entry} = (region, agg, ptr)$, where agg stores aggregated data about the successor nodes that is computed according to aggregation function f.
Let s be the node ptr is referring to. Then agg is computed by:

$$agg = \begin{cases} f(\{data_{entry}.attr | data_{entry} \in s\}) & : \quad s \text{ is a leaf node} \\ f(\{dir_{entry}.agg | dir_{entry} \in s\}) & : \quad s \text{ is a non-leaf node} \end{cases}$$

The aggregation functions which are applicable to compute data for materialization in the inner nodes are discussed in the following section.

5.3. Materialization of data

Section 5.2 examines how aggregated data is stored in the tree. This section investigates what aggregation functions are applicable to pre-compute data for materialization. In principal, any statistical operation can be used to compute new data from existing data. However, in hierarchical structures like trees, some functions are calculated by using pre-aggregated values while other functions can only calculated by accessing all raw data. Gray et al. classifies aggregate functions into the three categories: *distributive*, *algebraic*, and *holistic* [Gray et al., 1997].

Definition 5.3 Let X be a multi-set and $X_{i,i\in\{1,\cdots,n\}}$ a partition of the multi-set X (*e. g.* $X = \biguplus_{i=1}^{n} X_i$). An aggregate function $f : X \to \mathbb{R}$ is distributive , if there exists a function g and such that, $f(X) = g(f(X_1), \cdots, f(X_n))$

Examples are `count`, `sum`, `min`, and `max`.

Definition 5.4 An aggregation function is called algebraic, if it can be calculated with a fixed number of distributive functions.

Examples are `average` and `covariance`. These functions are not materialized in the inner nodes, but they can easily be computed by arithmetic operations on stored aggregated values of distributive functions, *e. g.* average=sum/count.

Definition 5.5 An aggregation function is called holistic , if there is no constant bound on the size of the storage needed to describe a sub-aggregate.

Examples are `mostFrequent`, `ranks`, and `median`. The last kind of functions we generally do not support in the multidimensional case.

However, in the one-dimensional case `ranks` or `median` functions are supported by extending the B-tree to TBSAM [Srivastava et al., 1989] if an order relation

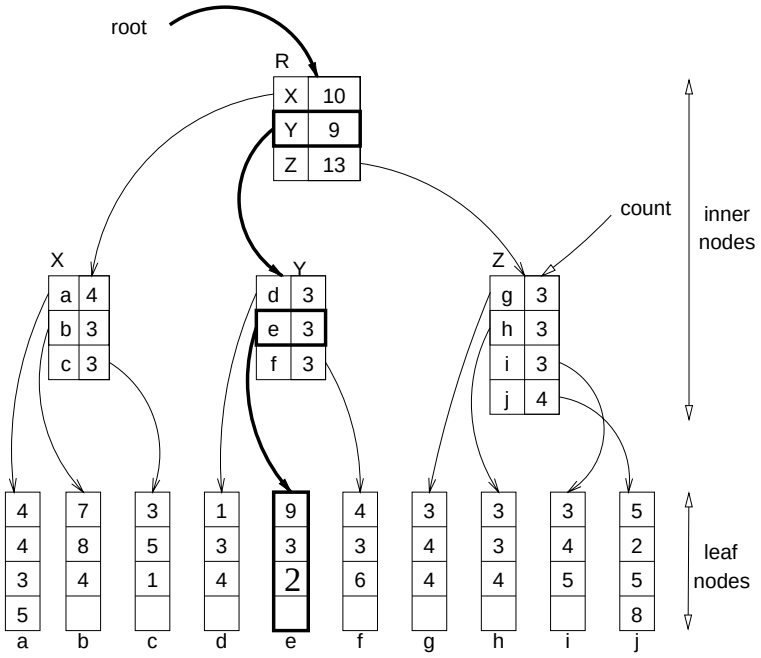

Figure 5.4.: Tree structure TBSAM and access to the 16th element

on the tree is given. We now provide an example to show how distributive aggregates are used to compute holistic functions if such an ordering can be assumed. Suppose that the data in Figure 5.4 is sorted according to a one-dimensional ordering and the task is to lookup the 16th element. If the count function is materialized in the inner nodes, it is possible to find the path to this element by using the count values on the upper levels. Starting from the root R, and looking at the $count$ values, the 16th element has to be in the second successor Y. In node Y we have to find the path to the $16 - 10 = 6th$ element. This has to be in the second entry e of Y. In node e, it is the $6 - 3 = 3rd$ value.

Using the materialized aggregates reduces the traversal of nodes to logarithmic cost for the ranking function. Without these aggregates it would be necessary to scan the data. In the one-dimensional example the computation of a holistic function is supported by aggregated values of a distributive function.

In the remaining part of the thesis we consider multidimensional data. In the case of multidimensional data it is not possible to use aggregated data to compute holistic aggregates.

Definition 5.3, Definition 5.4, and Definition 5.5 classify functions into the three categories distributive functions, algebraic functions, and holistic functions. This classification is meaningful, if and only if, insert operations are considered. However, if delete and update operations are considered, the class of distributive functions must be split into two subclasses:

Definition 5.6 An aggregate function $f : X \rightarrow \mathbb{R}$ is distributive additive if it is distributive and there exists a function h such that for any $Y \subseteq X : f(X \setminus Y) = h(f(X), f(Y))$.

Functions like `count` and `sum` are distributive additive.

Definition 5.7 An aggregation function is called distributive non-additive if the function is distributive but not distributive additive.

Functions like `max` and `min` are distributive non-additive.

The distinction between distributive additive and distributive non-additive functions is important in the case of delete operations. If a tuple is removed, a distributive additive materialized aggregate can be calculated incrementally. In general, distributive non-additive aggregates have to be computed from the scratch each time a tuple is deleted or updated.

In the following approach it is assumed that only distributive (additive and non-additive) aggregates are stored in inner nodes of the index structure. Algebraic aggregates can be calculated from distributive aggregates. Holistic aggregates are not supported. In addition to the mathematical constraints of aggregation of data, the semantics of aggregation are important [Lenz and Shoshani, 1997].

We investigated what kind of functions are applicable to aggregate data for materialization. Next, we examine how the operations on an index structure have to be modified to maintain and use the materialized data.

5.4. Modified operations

The class of tree-based index structures is viewed as an abstract data type on which operations are performed. The operations besides of search are insert, update, and delete. These algorithms are modified in order to maintain the materialized aggregates. Here we consider only distributive aggregates in the inner nodes. The query operation include point queries and range queries. Point queries will not be adapted, but range queries use the additional data to speed up the query processing time. The modification of these operations and their effect on time complexity are discussed in this section.

5.4.1. Insert operation

For each insert operation without a split one path is traversed from the root to the leaf. During the traversal of the inner nodes the aggregated data in the nodes is updated. This works for all distributive (additive and non-additive) aggregates. The number of nodes that are touched corresponds to the height of the tree similar to a structure without aggregated data. The only difference which may yield additional overhead, is that the inner nodes may be modified. If there occurs one or more splits

operation	distributive additive	distributive non-additive
insert	h	h
delete	h	$2h$

Table 5.1.: Number of nodes touched, h=height of tree

during the insertion phase, the aggregated data about the changed / created nodes has to be recalculated, but this does not involve additional update cost in terms of disk accesses.

5.4.2. Delete operation

First we consider the simple case where there is a delete operation without any underflow and merging of siblings. The tree has to be traversed from the root down to the leaf node where the tuple is stored which is deleted. If additive distributive aggregates have to be updated, it can be done during the traversal from the root to the leaf node and the costs are the same as in a standard index structure. If non-additive distributive aggregates are stored, there is an additional traversal from the leaf node to the root necessary to update all inner nodes on the path. If a delete operation yields to an underflow, two siblings have to be merged and all of the aggregated data has to be updated. However, there is no additional overhead in terms of additional access to non-leaf nodes.

5.4.3. Update operation

Update operations are exceptional in typical data warehouse applications. Therefore, we do not investigate them further. Each update can be substituted by one delete and one insert operation.

Table 5.1 summarizes the number of nodes that have to be touched for insert and delete operations when no split or merging of nodes is considered. Split and merge operations of nodes do occur only if many tuples are inserted or deleted. If a much data is changed, it is often more efficient to create a new index from scratch. For this purpose the bottom-up structures are advantageous.

5.4.4. Creating index structures, bottom-up index structures

As mentioned in Chapter 3, bottom-up structures like the packed R-tree [Roussopoulos and Leifker, 1985] or the STR-tree [Leutenegger et al., 1997] are well suited for *read-mostly* environments which are within the scope of this

thesis. If bottom-up structures are used, the leafs of the tree are generated first, and the other levels are built from the bottom to the top. This approach is very efficient for creating a new index structure. There is no additional cost in terms of additional access to secondary memory for adding the aggregated data in the inner nodes during the creation of such a bottom-up structure.

5.4.5. Point query algorithm

We do not change the exact match point query algorithm by using aggregated data. Since a smaller fanout in the inner nodes of the tree might increase the height it may be necessary to traverse more nodes than in a structure without aggregated data. How the structure is affected is discussed in Section 5.6.

5.4.6. Range query algorithm

We modify the range query algorithm in order to use aggregated data and to speed up queries. The main idea is to avoid traversing the next level of the tree if a rectangle of a directory entry is completely contained in the query box of a range query. In this case the pre-computed materialized values stored in the directory entry are used. Based on the assumption that data entries are only points (degenerated rectangles), Figure 5.5 shows the new recursive range query algorithm. The aggregation operation is represented as f, the materialized values are stored in $dir_{entry}.agg$, and the data for materialization is stored in $data_{entry}.attr$.

The function is invoked by $calc\_agg(root, querybox)$. The function checks first if the node is a leaf node or not. In case of a leaf node for each data entry it is checked if the data entry lies in the query box. If the point lies in the query box, the value is used to calculate the aggregate. In case of a non-leaf node the function checks if the corresponding rectangle is completely contained inside the querybox q. If it is completely contained, these data can be used. Otherwise, the function calls itself recursively and proceeds to the next level of the tree.

5.5. Storage cost

The storage of additional aggregates in the index entries reduces the fanout of the non-leaf nodes. This implies that the height of the tree may increase and more inner nodes are needed to hold all necessary entries. The effect of the reduced fanout on extra space for inner nodes is shown in Theorem 5.1. Here we assume that the fanout of the directory entries is fixed:

Theorem 5.1 Let U be a tree-based index structure with n leaf nodes, u the fanout of the inner nodes, and n_u the total number of nodes. Without loss of generality we assume $n \geq u$. Let V be a second tree with n leaf nodes, reduced fanout $v > 1$ with $v < u$, and n_v the total number of leaf nodes. Let $f(u, v) = \frac{n_v}{n_u}$

```
function calc_agg(page p, region q)
initialize result;
if p is leaf node
  for all data_entry ∈ p
    if data_entry ⊆ q
      result := f(result, data_entry.attr);
else
  for all dir_entry ∈ p
    if dir_entry ⊆ query
      result := f(result, dir_entry.agg);
    else
      if (dir_entry ∩ query) ≠ ∅
        result := f(result, calc_agg(dir_entry.ptr, q));
      endif
    endif
endif
return result;
```

Figure 5.5.: Algorithm calc_aggregate for recursive range query processing

be the factor the number of nodes increases when switching from structure U to structure V. This factor has an upper bound $K = \frac{1}{\left(1-\frac{1}{v}\right)\left(1+\frac{1}{u}\right)}$.

Proof:
We assume having a tree-based index structure with n leaf nodes and a fanout of u. Then the total number of nodes (leaf nodes + directory nodes) is calculated by:

$$n_u \approx n + \sum_{i=1}^{\lceil \log_u n \rceil} \frac{n}{u^i} = n \sum_{i=0}^{\lceil \log_u n \rceil} \left(\frac{1}{u}\right)^i = n \frac{1 - \left(\frac{1}{u}\right)^{\lceil \log_u n \rceil + 1}}{1 - \frac{1}{u}}$$

For the tree structure with a fanout of v, the total number of nodes is:

$$n_v \approx n \frac{1 - \left(\frac{1}{v}\right)^{\lceil \log_v n \rceil + 1}}{1 - \frac{1}{v}}$$

If we switch from a tree structure with a fanout of u to a tree structure with a fanout of v, the number of nodes increases at most by factor:

$$\begin{aligned} f(u,v) = \frac{n_v}{n_u} &= \frac{n\left(1-\left(\frac{1}{v}\right)^{\lceil \log_v n\rceil+1}\right)\left(1-\frac{1}{u}\right)}{n\left(1-\frac{1}{v}\right)\left(1-\left(\frac{1}{u}\right)^{\lceil \log_u n\rceil+1}\right)} \\ &\leq \frac{1\left(1-\frac{1}{u}\right)}{\left(1-\frac{1}{v}\right)\left(1-\left(\frac{1}{u}\right)^{(1+1)}\right)} \\ &= \frac{1-\frac{1}{u}}{\left(1-\frac{1}{v}\right)\left(1-\left(\frac{1}{u}\right)^{2}\right)} \\ &= \frac{1}{\left(1-\frac{1}{v}\right)\left(1+\frac{1}{u}\right)} =: K \end{aligned}$$

□

With Theorem 5.1 it is possible to give an upper bound for the additional space that is needed when the fanout of the inner nodes is decreased. Figure 5.6 illustrates this theorem and shows the maximum additional storage cost depending on the decreased fanout for a tree with $1,000,000$ leaf nodes. The fanout of the first tree structure is set $u = 102$ and the fanout of the second structure v is varied between 40 and 102. For a fanout of $v = 102$ of the second structure is the same structure as the first structure and no additional space is needed. For a reduced fanout of $v = 51$ only half of the directory entries fit on each directory page. Figure 5.6 shows that about one percent additional space is occupied. The additional space overhead is rather small in comparison with the whole space when the fanout is reduced.

For the above calculations it is assumed that all nodes have the same fanout. This is true for bottom-up structures, but not for index structures in general. Therefore, experimental evaluation is done with an R^*-tree. The R^*-tree with aggregated data will be denoted by R^*_a-tree [Jürgens and Lenz, 1998]. For different number of tuples an R^*-tree and an R^*_a-tree are compared and the additional storage costs is measured. In the experiments we measured on average $u = 12.8$ and $v = 9.9$ for a blocksize of $b = 512$ Bytes. We compute $K \approx 1.032$. Table 5.2 shows results from our experiments. In none of the experiments the space is increased by more than 3.05 %. These results confirm that the calculation of the upper bound K is quite close to the experimental measured value for the additional space.

5.6. Height of tree

Due to the decreased fanout the number of inner nodes might increase, but also the height of the tree might increase. For each processing of one level of a tree additional time is needed. Therefore, the height h of a tree is an important parameter on efficiency and is investigated. Figure 5.7 shows the height of the tree for different number of tuples t between 10^2 and 10^7 on a logarithmic scale. The fanout of the tree

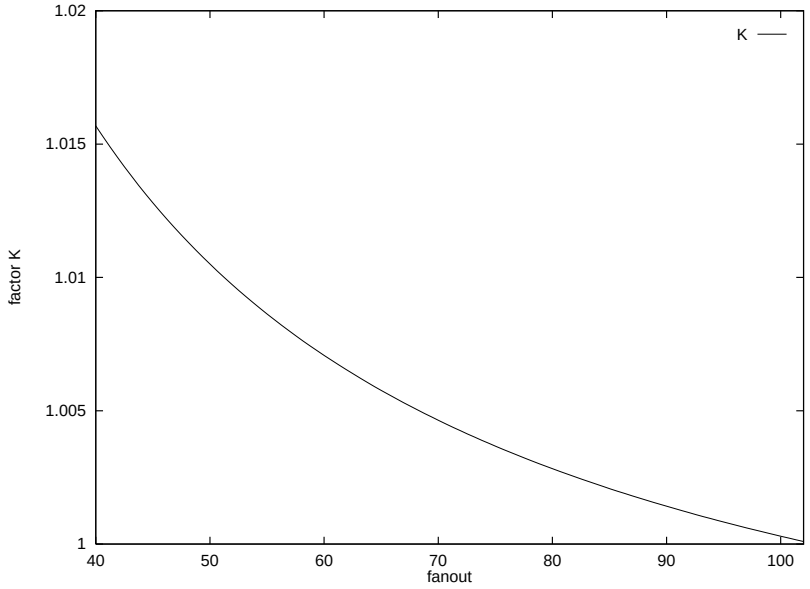

Figure 5.6.: Upper bound for additional nodes depending on reduced fanout v on x-axis ($n = 10^6$, $u = 102$)

Table 5.2.: Factor for additional space

t	additional space $f(u, v)$
6 K	1.0296
60 K	1.0209
600 K	1.0305
1.200K	1.0273
1.800K	1.0260
6.000 K	1.0260

($u = 12.8$, $v = 9.9$)

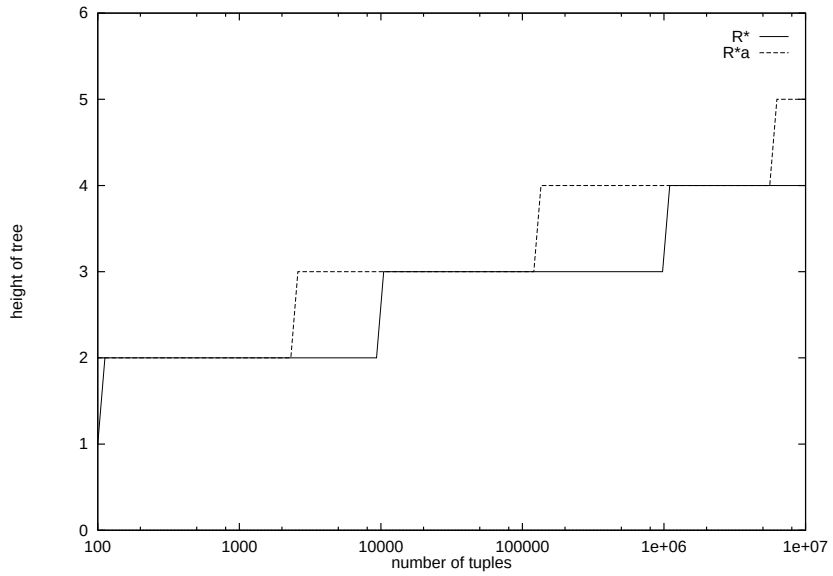

Figure 5.7.: Increased height of tree depending on the number of tuples t ($v = 51$, $u = 102$

with aggregated data is set to $v = 51$ and the fanout of the tree without aggregation is set to $u = 102$. Figure 5.7 shows that even if the fanout of the inner nodes is cut down by half, the height of the tree is only locally increased.

5.7. Overlaps of regions

Some multidimensional index structures (*e. g.* the R^*-tree) allow overlaps between different rectangles on the same level. Structures like the R^+-tree or kdb-tree avoid overlaps but no minimal fill grade can be guaranteed. To be general, the extension of aggregated data in the inner nodes is applicable to index structures that allow overlaps between rectangles of different nodes on the same level. This assumption does not restrict our approach with materialized aggregated data. A rectangle or point of a lower level can lie in more than one rectangle of an upper level. Each rectangle or point has exactly one predecessor that refers to it. In the node of this predecessor, materialized aggregated data is stored. In Figure 5.8 a small tree with two leaf nodes is shown. On the left it can be seen that point 5 lies in the rectangles of leaf nodes A and B. At the right side the tree representation shows that point 5 belongs to rectangle B. Its value is used only in the $count$ and sum of the index entry for node B. A point might lie in two regions, but it is stored only in one leaf node. Therefore, the value of each point is used only in the computations of one aggregate on each level of the tree. The same is true for the upper levels of a tree. A region of a non leaf node can lie in more than one region of entries on higher levels of the tree, but it always belongs to exactly one node.

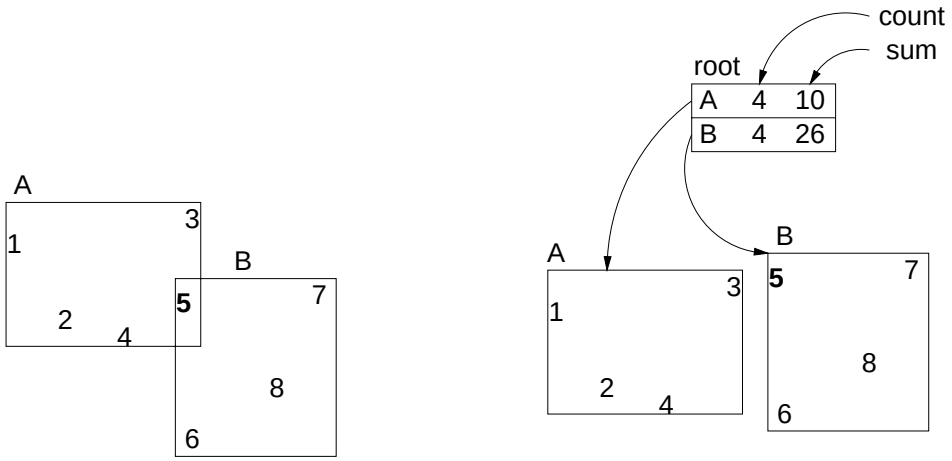

Figure 5.8.: Overlaps between region A and region B

5.8. Experiments

In oder to check to what degree the proposed extension yields performance improvements, we perform a number of experiments. In this section we describe the experiments in detail. First we define the cost model, the physical index structures and the implementation. Then we specify the generation of test data and the query profile. Finally, we present the results of the experiments.

5.8.1. Cost model

Without loss of generality we assume in this chapter that every *leaf* node is mapped to exactly one block on hard disk. If a *leaf* node is mapped to m blocks, the cost is multiplied by the constant factor m. The processing of a *leaf* node corresponds to a one-to-one access to the hard disk (or one to m). The time needed to read the block from disk and to transfer it into main memory is the critical factor. The time for traversing inner nodes and doing computations is negligible. This is due to the fact that CPUs are getting much faster and the capacity of main memories is increasing while random disk accesses remain slow.

The number of inner nodes is much smaller than the number of leaf nodes. In our experiments the number of inner nodes was usually less than 2 % of all nodes. Therefore, we assume that the inner nodes are held in main memory. In contrast is the number of leaf nodes so large that they are stored on the disk. As we mentioned in Chapter 3, disk accesses are by a factor 10^5 slower than accesses to main memory. Therefore, we consider in our cost model only the accesses to the leaf nodes.

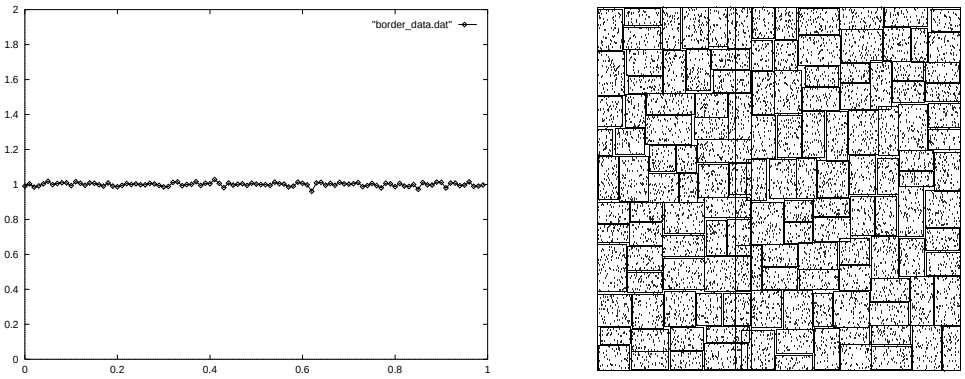

Figure 5.9.: Uniformly distributed data (left: generated density function, right: scatter diagram with rectangles of nodes of R^*-tree)

5.8.2. Physical index structure

Any physical index structure structure that fulfills Definition 5.2 can be used. This chapter presents experiments where the R^*-tree is applied. The R^*-tree is very robust and a widely used reference structure to check the performance of new structures. It will be compared with the R_a^*-tree [Jürgens and Lenz, 1998].

5.8.3. Implementation

To evaluate how the extension works and how much additional space is needed for physical index structures an R^*-tree implementation based on code by Beckmann et al. is used and modified. The algorithms are coded in C and run on a SUN Ultra Sparc with 128 MB RAM.

5.8.4. Generation of test data

The experiments use different data sets. All data sets are two-dimensional and generated and are in the range of $[0, 1)^2$. Uniformly distributed data, skewed distributed data, and normally distributed data is used. For each distribution 10 sets of data with 1,000,000 tuples in each set are generated. We insert the ten different sets of data into ten R^*-trees and in ten R_a^*-trees. The blocksize is set to $b = 2KB$. The size of one directory entry is $s_{dir} = 20B$. Therefore, the resulting fanout of the tree without aggregated data is $u = \lfloor \frac{2048B}{20B} \rfloor = 102$. The aggregated data `count` and `sum` are materialized in the inner nodes. Therefore, the size of the directory entries increases to $s'_{dir} = 28B$. The resulting fanout for the structure with aggregated data is $v = \lfloor \frac{2048B}{28B} \rfloor = 73$.

The left side of Figure 5.9 shows the empirical density function of one-dimensional uniformly distributed data generated with the C functions `srand48/drand48`. The right side of Figure 5.9, shows 10,000 uniformly dis-

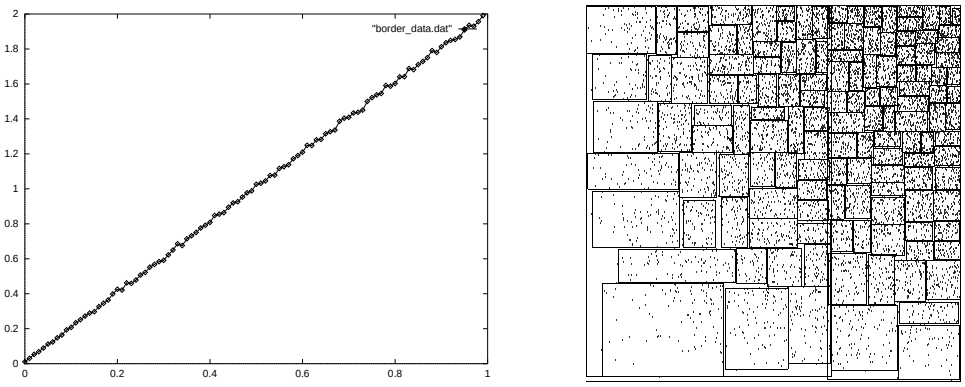

Figure 5.10.: Skewed data (left: generated density function, right: scatter diagram with rectangles of nodes of R^*-tree)

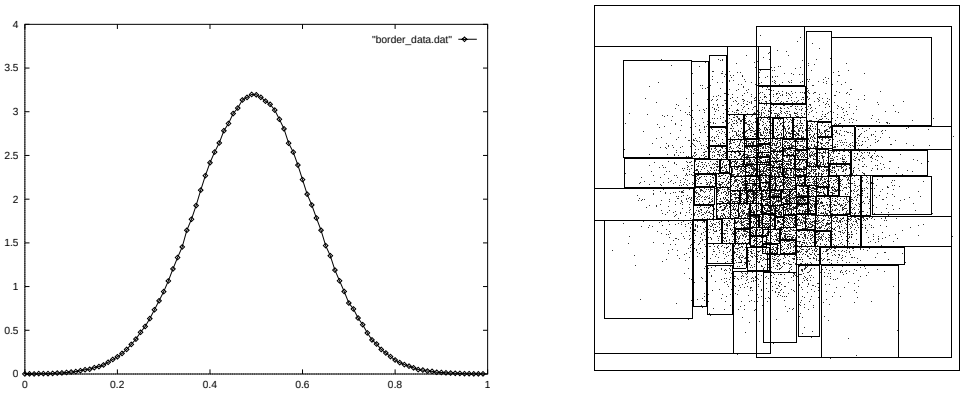

Figure 5.11.: Normally distributed data (left: generated density function, right: scatter diagram with rectangles of nodes of R^*-tree)

tributed points inserted in an R^*-tree. The ten different data sets are generated with different seeds. The rectangles represent the data rectangles of the leaf nodes. The right side of Figure 5.9 shows that the rectangles have approximately the same size.

Figure 5.10 shows skewed data which is computed from uniformly distributed data by applying the function $f(x) = \sqrt{x}$, $x \in \mathbb{R}$. This holds, because $\sqrt{x}$ is the inverse function of $F(x) = \int_0^x 2y \; dy$ and $2y$ is the density function shown on the left side of Figure 5.10.

Figure 5.11 shows normally distributed data with $\mu = 0.5$ and $\sigma = \frac{1}{8}$. This data is generated from the uniformly distributed data by applying the Box-Muller method [Box and Muller, 1958].

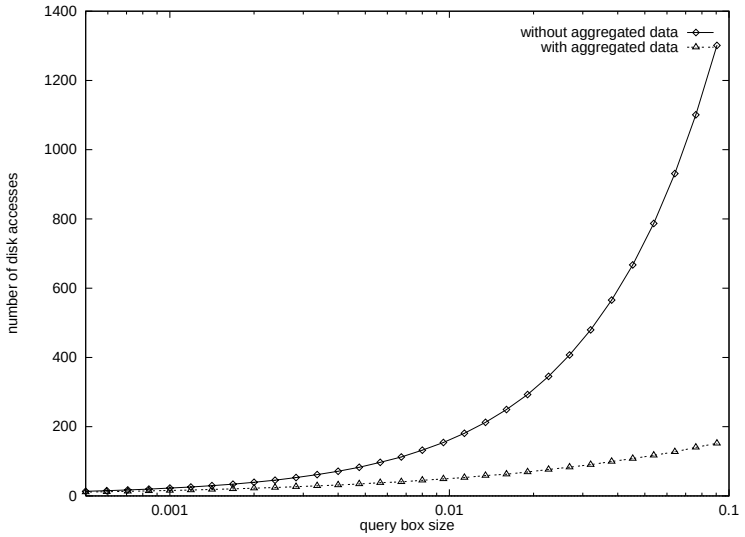

Figure 5.12.: Average number of accesses for uniformly distributed data

5.8.5. Query profile

For each experiment we perform a different query mix. We assume quadratic query boxes of different sizes. The area of the query boxes relative to the data space is $\{0.0005 * (2)^{\frac{i}{4}} | 0 \leq i \leq 30\}$. The distribution of the center of the query boxes is the same as the distribution of the data. *E. g.* uniformly distributed data is queried with query boxes uniformly distributed over the data space, normally distributed data is queried with query boxes normally distributed over data space. For each size of a query box and for each data set, 100 queries are processed and the average number of accesses to leaf nodes per query box is computed.

5.8.6. Results of experiments

Figure 5.12 shows the number of disk accesses for the above described query mix for uniform distribution of data and queries. The x-axis shows the query box size in a logarithmic scale. The y-axis represents the number of disk accesses for the whole query mix. Figure 5.12 shows that for query boxes smaller than 0.001 of the data space the differences between the R^*-tree and the R_a^*-tree are rather small. For query box sizes between 0.001 and 0.01 the benefits of the R_a^*-tree are getting larger. For query boxes larger than 0.01 of the data space or more than one percent there are significant savings when using the R_a^*-tree in comparison with the R_a^*-tree, i. e. up to 85 %.

Figure 5.13 displays the number of accesses for skewed data and Figure 5.14 shows the results of the experiments with normally distributed data. These two Figures show the same trend as the data presented in Figure 5.12.

These experiments allow the following conclusion: The savings with the exten-

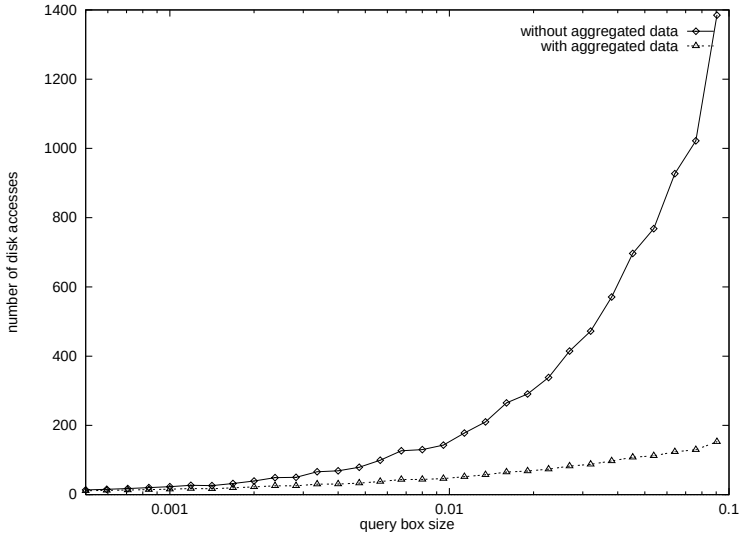

Figure 5.13.: Average number of accesses for skewed data

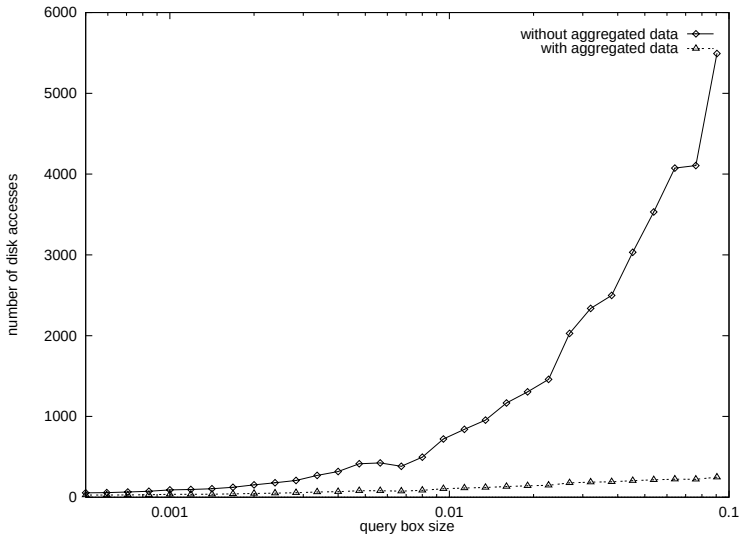

Figure 5.14.: Average number of accesses for normally distributed data

sion of the structure depend on the area of the query box. The greater the query box, the greater are the savings. The number of accesses of the tree without aggregated data is proportional to the area of the query box. The number of access of the tree without aggregated data is proportional to the perimeter of the query box. Therefore, the savings increase for larger quadratic query boxes.

The savings also depend on the number of indexed tuples [Jürgens and Lenz, 1998]. The greater the number of tuples, the greater are the savings with the new structure.

5.9. Summary

Chapter 5 described an extension where aggregated data is stored in the inner nodes of an index structure. The aggregated data is stored in addition to the references to the successor nodes. An example shows how this extension works and how it improves the processing of range queries on aggregated data. It was defined what kind of index structures can be applied with this extension. Further investigations showed for what kind of aggregation functions can be used to pre-compute data and store it in the inner nodes. Operations on the index structure were modified to maintain the redundant data. We changed the range query algorithm and showed how much additional space-overhead is created. Experiments measured the accesses to secondary memory with a standard R^*-tree and with the new extended structure for different sizes of the query boxes. The results showed the benefits of the aggregated data in the index structures. For uniformly, skewed, and normally distributed data significant savings in disk accesses were obtained.