Tuning Access Methods to Supporting Data-Intensive Queries

Bernhard Seeger

Ludwig-Maximilians-Universität München

Institut für Informatik

Bericht 9419

November 1994

Tuning Access Methods to Supporting Data-Intensive Queries

Bernhard Seeger *

Abstract

In this paper, we consider how to improve dynamic access methods which are designed to perform data-intensive selection queries in a dynamic setting. A large number of dynamic access methods have been proposed for supporting such queries. Almost all of them are designed with the primarily goal to reduce the number of page accesses, whereas our main attention is paid to decreasing the cost of individual page requests. There is a great potential for such improvements since the average (page) access time on a magnetic disk drive is typically by a factor of 10 higher than the transfer time. In our approach, two techniques are jointly used: clustering and multi-page requests. The technique of clustering takes care of storing pages on one cylinder which are frequently required by the same query in common. In contrast to issuing a request for each qualifying page of a query, one multi-page request is used for retrieving all qualifying pages from a cylinder. Due to these techniques, the access time of a qualifying page can be substantially reduced compared to the average (disk) access time. For the sake of concreteness, our discussion is based on the problem of supporting range queries on B^+-trees. In an experimental performance study, we show that the response time of a range query can be reduced by several factors if the B^+-tree exploits both techniques (clustering and multi-page requests).

1 Introduction

The importance of supporting selection queries with a large response set on a dynamic file has been reflected in several database benchmarks [Gra91]. Such queries

*author's address: Institut für Informatik, Universität München, Leopoldstr. 11B, D-80802 München, Germany, email: bseeger@informtik.uni-muenchen.de, Tel.: 0049-89-2180-3881, FAX: 0049-89-2180-5246

have two important properties: First, answers are found in several pages which have to be retrieved from secondary storage (e.g. magnetic disks) and second, the time spent on the disk determines the response time of a query. In general, query processing cost is expressed in the number of page accesses assuming that the cost for each page access is the same. The range query is the most important example for such a query and therefore, many dynamic access methods, e.g. B^+-trees, are primarily designed to perform range queries efficiently under frequent insertions and deletions of records. Since efficiency of range queries directly corresponds to the number of page accesses, the cost can be substantially reduced by assigning records to pages such that the keys of the records which are stored in a common data page are close to each other.

In this paper, we address the problem of redesigning dynamic access methods for performing range queries efficiently under a different perspective than usually found in the literature. The goal is not only to reduce the number of page accesses, but also to reduce the cost of the individual page accesses. The cost of reading a page from a magnetic disk drive consists of three parts: seek time, rotational delay and transfer time. Typically, seek time and rotational delay is much higher than the transfer time. Moreover, over the last 20 years, the average rotational delay has decreased only slightly, whereas the average seek time has been shortened significantly [HBP+81, JW91]. For the near future, it is expected that the rotational delay of a request will generally be higher than its seek time. The idea is to reduce the average seek time and the average rotational delay required for reading a qualifying page of a range query. To achieve this, we propose that access methods should exploit two techniques in common. First, adjacent pages (which contain records close to each other) should be stored physically close to each other on magnetic disk, preferably on one cylinder. This technique called *clustering* in the following basically results in reducing the seek time. In order to reduce the rotational delay, we suggest that the qualifying pages of a range query are read by issuing *multi-page requests* also known as set-oriented I/O [Wei89] and bulk I/O [Wil94]. In contrast to reading the qualifying pages one at a time, a multi-page request retrieves several pages from disk in an order which minimizes the total processing cost of the request. The improved range query performance of our redesigned access method is not given for free. The cost for insertion and deletion of records and the storage cost slightly increases compared to the cost of the original access method.

Our approach is easy to implement (assuming that the underlying system offers multi-page requests) and is generally applicable to a large number of dynamic access methods including B^+-trees [BM72, Knu73], R-trees [Gut84] and its variants [BKSS90]. For the sake of concreteness, we base the presentation in this paper to B^+-trees; it is implicit how to apply our approach to other methods.

Most related to our approach of clustering is VSAM [KL74], a variant of the B^+-tree with the facility of clustering data pages. However, VSAM allows only a limited form of multi-page requests: the target pages have to be contiguous on magnetic disk. Moreover, VSAM is primarily concerned about seek time, whereas rotational delay is not explicitly considered for cost reduction. Consequently, the order in which pages are read from a cylinder is not optimized with respect to the physical layout of the pages. Another disadvantage of VSAM is that its implementation directly depends on the underlying hardware, whereas our approach exploits an upper software interface which facilitates the redesign of existing access methods. All other approaches to clustering pages of access methods [HSW88, O'N92, BK94] assume a simplified disk model: the disk corresponds to a linearly ordered sequence of pages. Correspondingly to VSAM, multi-page requests are only allowed to read a contiguous sequence of the pages from the "linear" disk.

The rest of the paper is organized as follows. In section two, we identify the problems of performing range queries on ordinary B^+-trees. In section three, we introduce our disk model. In section four, we propose a new design of a modified B^+-tree, called CB^+-tree. Emphasis is put on the dynamic reorganization and on the range query. In section five, we report the results of an experimental performance comparison. The results indicate that our method considerably improves the cost of range queries compared to B^+-trees.

2 The Disk Model

Magnetic disks are the most important storage medium to date. In the context of a database system (DBS), a magnetic disk is considered as a storage device with direct access to pages. In general, the I/O cost of database queries is expressed in terms of page accesses assuming that the access cost is equal for each page. Due to this simplistic model, the only objective of query processing can be to minimize the number of disk accesses, whereas the cost of an individual disk access is of no concern. Recently, Ruemmler and Wilkes [RW94] examined various disk models and showed that the quality of the model has a great impact on the accuracy of cost estimation. This is even true when only one page has to be read from disk. In particular, they provided an excellent description of an accurate magnetic disk model that serves as a guideline for the following model.

In our disk model, we assume that the total cost of retrieving a page from disk consists of three major components: seek time, rotational delay and transfer time. Seek time denotes the time to move the disk arm to the desired cylinder. After arriving at the target cylinder, the arm waits until the disk is rotated into the position such that the desired page appears under the arm. The corresponding time is called rotational delay. Finally, the page is transferred into main memory and the corresponding time is known as transfer time.

The cost model can be improved by taking into account the time required for performing a head switch. A head switch occurs when a page is required from the cylinder where the disk arms are currently positioned. If the target page is on a different track than the one previously read, the channel of the disk has to be switched from one disk arm (head) to another. In general, the newly activated disk arm is not accurately positioned above the desired track and therefore, needs some mechanical fine-tuning. The corresponding time to perform such a head switch is called *head switch time*.

In order to simplify our model as much as possible without losing much of its accuracy, we make the following assumptions: First, we do not consider the overhead of the disk controller (which is generally less than 1 ms) and channel

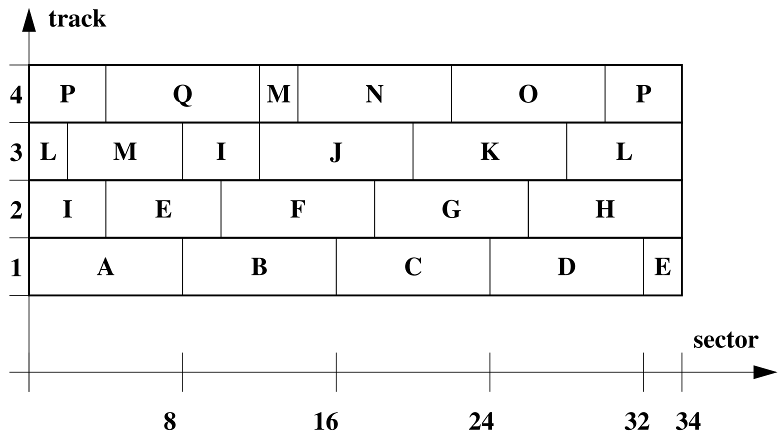

Figure 1: An example for the layout of pages on a cylinder ($TC = 4$, $PC = 17$, $ST = 34$)

contention (when multiple disks share a common disk controller). Second, for data-intensive queries which process a qualifying page only once (e.g. range queries), a buffer would only very little contribute to a performance improvement. Therefore, we assume that there is no buffer on the disk. For other queries, however, such a buffer can have a serious impact on query performance [RW94].

The layout of pages on a disk is assumed to be as follows. A cylinder contains PC pages where PC is assumed to be an integer. A cylinder of the disk consists of TC tracks, each of them containing ST sectors. A sector is the smallest unit on disk that can be referenced. A page is composed of a contiguous sequence of sectors. On each track a contiguous sequence of pages is stored. The first and the last page of that sequence may only be partly on the track. The other part of these pages is stored either on the predecessor track or on the successor track. When such a page is transferred into main memory, a head switch has to be performed. In order to transfer such a page in minimum time (without losing a full disk revolution), the two parts of the page are staggered on the different tracks. The distance from the end of the first part to the beginning of the second part is only determined by the head switch time. The transfer cost of such a page is therefore higher (by the head switch time) than the transfer cost of a page stored entirely on one track.

In Figure 1, an example is given for a typical layout of pages on a cylinder. A page consists of 8 sectors. Some of the pages, for example page E, are stored on two adjacent tracks. Due to the head switch time, the second part of such a page is staggered by four sectors.

In addition to reading single pages from disk, we allow to issue multi-page requests. A *multi-page read request* retrieves several pages from disk which are *not* necessarily contiguous. The disk services a multi-page request from beginning to end without giving service to other requests. Therefore, the response time of a multi-page requests depends mainly on the read schedule, i.e the order in which the required pages are read from disk.

For simplicity, we assume in this paper that a multi-page request is restricted to retrieve pages from only one cylinder. We then suggest that the target pages of a multi-page request are read according to the shortest-latency-time-first (SLTF) policy [Den67]. The SLTF policy determines the target page as the next one in the schedule which requires minimum latency among all the target pages not processed so far. Under the assumption that head switch time can be neglected, the SLTF policy has been shown to produce schedules close to optimal [SF73].

In analogy to an ordinary single-page request, we define the terms seek time, rotational delay and transfer time for a multi-page request. Since target pages of a multi-page request are on a single cylinder, the definition of *seek time* remains the same. *Rotational delay* now refers to the time until the first of the target pages appears under the disk arm. *Transfer time* refers to the time required for reading all target pages assuming that the disk arm is in front of the first page. Since target pages are not contiguous on disk, transfer time of a multi-page request is higher than the sum of the net transfer time of the target pages.

Consider the example that a multi-page request retrieves the pages C, E, G and M. Let us assume that the page layout of the cylinder refers to the one depicted in Figure 1 and that the initial position of the disk arm is on sector 0, i.e. a page whose first sector refers to sector 0 can be read without any delay. Then, the SLTF policy creates the read schedule M, G, E, C. The rotational delay of the multi-page request and the transfer time corresponds to $\frac{2}{34}$ and $1\frac{22}{34}$ disk rotations, respectively.

3 Motivation

In this section we discuss an example for B$^+$-trees to identify the problems of dynamic index structures without the benefits of clustering pages and issuing multi-page requests. For sake of simplicity, it is assumed that a record only consists of an integer key. At most 3 records and 8 entries can be kept in a data page and directory page of the B$^+$-tree, respectively. The following sequence of records is inserted into the B$^+$-tree:

$$146, 75, 3, 95, 189, 165, 106, 229, 239, 14, 208, 90, 8, 222, 122$$

After record 239 is inserted, the B$^+$-tree consists of four data pages, see Figure 2. The pages are labeled E, C, G, M. In our example, the data pages of the B$^+$-tree are stored on a common cylinder of the disk. As depicted on the right hand side of Figure 2, the disk consists of five cylinders. The parameters of the cylinders refer to the ones of the example depicted in Figure 1.

A range query, e.g. find all records which are in the range $[100, 230]$, is performed in the following way: First, an exact match query is performed using the lower key of the specified range. This operation results in reading a data page that serves as initial page for traversing data pages in key-order following the corresponding pointers. During traversal, all records are retrieved which are in the specified range. If a record is found whose key is greater than the upper key of the range (or the highest key is found in the file), all answers of the range query have been retrieved. For example, if $[100, 230]$ is the range, pages C, G, M are read into main memory.

The algorithm for performing a range query on a B$^+$-tree is asymptotically optimal, i.e. $O(q/b)$ data pages are required in the worst case where q denotes the size of the response set and b denotes the capacity of the data pages. However, the example in Figure 2 already shows that the number of page accesses is only a coarse

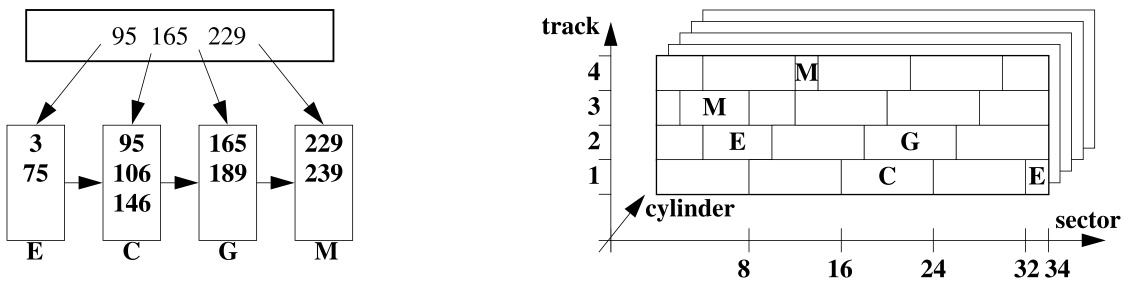

Figure 2: An example of a B$^+$-tree and its data page layout on disk

performance measure. Under the assumption that the disk belongs exclusively to the query, only one seek is required for reading the qualifying pages into main memory. After the seek is performed, let the position of the disk arm be on sector 0. Assuming no processing delays of the pages, the rotational delay and the transfer time for reading the qualifying pages then corresponds to $2\frac{14}{34}$ rotations. If, however, the addresses of the qualifying pages are known in advance, the pages could be read with only one multi-page request. The cost would then correspond to only $1\frac{24}{34}$ rotations. In order to exploit multi-page requests qualifying pages cannot be read sequentially one at a time and therefore, the original range query algorithm of the B$^+$-tree cannot be used anymore.

In the following, we show that pages are generally not clustered in (dynamic) B$^+$-trees. If a data page of a B$^+$-tree has to be split into two, the B$^+$-tree sends a request for a new page to the disk manager. Here, let us assume that the policy of allocating a new page is similar to the one of the fast UNIX file system [MJLF84]: For each file, the disk manager marks the cylinder which satisfied the last request for a new page. If a page is still available from the marked cylinder, the disk manager will give one of the free pages to the B$^+$-tree. Otherwise, the disk manager randomly determines a new cylinder where a large number of free pages can be found. Obviously, this policy is very efficient as long as all pages of a B$^+$-tree can be kept on a cylinder.

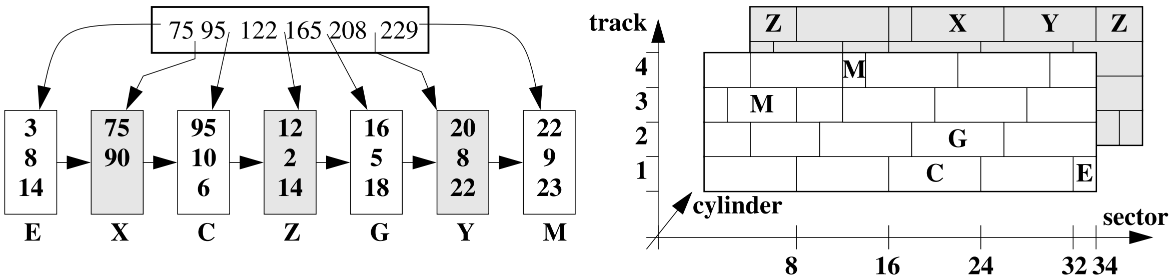

Figure 3: The example after all records are inserted

For large B$^+$-trees, the capacity of a cylinder is much too low for storing all the data pages. In particular, the dynamic behavior of the B$^+$-tree is responsible for worsening the page layout such that for a range query the cost of retrieving a qualifying page increases. The reason for the increasing cost is that when a data page is split into two the new page is mostly allocated on a different cylinder. Under the assumption that an overflow can occur in every page with the same likelihood, adjacent pages are stored on different cylinders with high probability. Figure 3 shows this property for the example B$^+$-tree after all remaining records are inserted. The B$^+$-tree consists of three new pages X, Y, Z which are stored on the fourth track of the fifth cylinder of the disk. We have assumed that no space was available to store these pages on the first cylinder. The 2nd, 3rd and 4th cylinder of the disk do not contain pages of the B$^+$-tree and therefore, are not depicted in Figure 3. Note that adjacent data pages of the example B$^+$-tree are always stored on different cylinders. The range query $[100, 230]$ now requires five data pages. In addition to rotational delay and the transfer time, five seeks have to be performed for reading the qualifying pages. Thus, the average cost for reading a qualifying page is substantially higher for this B$^+$-tree compared to the one of Figure 2.

There are two solutions to reduce the cost of a range query. The first one is to compute all addresses of the required data pages before accessing one. Then, these addresses are sorted with respect to the cylinder of their pages. For each cylinder hit by the query, all pages with qualifying records are read from the cylinder. However, there are two important drawbacks to this approach. First, this method only improves the performance of large and medium-sized range queries. Small range queries still suffer from the fact that nearby data pages are stored on different cylinders. Second, the answers of the range query are not delivered in key-order anymore and therefore, this approach is only appropriate for queries which do not require that the answers are in key-order. Moreover, qualifying pages are not clustered only a few cylinders, but on almost all cylinders where data pages of the B^+-tree are stored. This has an important impact on the response time of small and medium-sized range query. For these types of queries, the potentiality of improving performance is exploited only partly.

The second solution is to cluster data pages which contain adjacent records on the same cylinder. This can easily be achieved for a static file. Moreover, pages can then be stored always contiguously on disk with respect to the key-order [1]. For dynamic files, however, this solution would frequently trigger expensive reorganizations on the file. In order to reduce reorganization cost, a different clustering policy is used: Adjacent pages are still clustered on a common cylinder with high probability, but they will not necessarily be stored contiguously. This results in a loss of clustering which can however be compensated by using multi-page requests for retrieving the qualifying pages of a query. In the following section, we show in full detail how both techniques (clustering and multi-page requests) can be applied to B^+-trees.

4 The CB^+-Tree

In this section, we present a new approach to B^+-trees, called *cluster B^+-trees (CB^+-trees)*, which dynamically maintain clustering of pages on magnetic disks and perform range queries using multi-page requests. The section is structured into four subsections. First, the data structure is briefly discussed. Splitting is then

[1] We assume a key that consists of only one dimension.

explained for directory pages and clusters. Finally, the algorithm for performing range queries is outlined.

4.1 The Data Structure

There is almost no difference in the data structure between a B^+-tree and a CB^+-tree. The CB^+-tree is a balanced tree with directory and data pages whose capacity is limited to b entries and a records, respectively. The basic idea of the CB^+-tree is that a set of adjacent pages which belong to the same level of the tree is dynamically clustered close together on disk. Dynamic in this context means that clustering can be preserved without performing expensive reorganizations on the CB^+-tree. For sake of simplicity, we restrict the approach to data pages in the following. The same approach can also be applied to directory pages.

Similar to clustering adjacent records in a data page, the CB^+-tree clusters adjacent data pages in so-called *containers*. A container is best compared with a small file which can contain at most c data pages. In general, the pages of a container cannot always be stored contiguously on disk, but they may be scattered in a small partition of the disk. In the following, we assume that a container completely belongs to one cylinder of the disk. Thus, one multi-page request can read all pages of a container into main memory.

The directory pages in the level above the data pages are partitioned into so-called *subpages*. A subpage contains all the entries which refer to the same container. There are several subpages in a directory page and therefore, $c \leq b$ has to be fulfilled.

In order to provide clustering under insertions and deletions of records (data pages), containers have to be reorganized in an appropriate fashion. There are two cases which will be discussed in more details in the following subsections:

- After a data page is split into two, the new data page may belong to a full container (with c pages). This is called a *container overflow*. Similar to an ordinary page overflow, a container overflow is eliminated by splitting the full container into two.

- When the original algorithm of the B^+-tree is used for splitting directory pages of the CB^+-tree, a problem might be that a subpage is distributed over several directory pages.

4.2 Splitting of a Container

The approach to splitting a container is first a direct adaptation from the splitting algorithm of the B^+-tree. It has been already proposed for VSAM [KL74]. The basic idea is as follows. First, space for a new container is allocated on a new cylinder and the second half of the data pages in the full container is moved to the new container. Next, the corresponding references in the subpage are updated and the directory page is written back to disk. This is called a *complete reorganization.* Under the assumption that a sufficiently large space can be found on a cylinder, this approach guarantees that subpages contain at least $\lceil c/2 \rceil$ entries. Another advantage is that the cost of moving the data pages from one cylinder to another can be performed in two multi-page requests. Therefore, the cost for moving one of the pages in a complete reorganization is substantially lower than the average cost for an disk access. However, the total cost of an insertion operation which triggers a complete reorganization step is substantially higher than the cost for an insertion operation in an ordinary B^+-tree. For some applications (e.g. databases with real-time constraints), the total cost of an insertion operation might be not acceptable anymore. In the following, therefore, we present a more general approach which allows to control the reorganization cost of an insertion operation if necessary.

The basic idea is to relax the requirement that a container consists of at least $\lceil c/2 \rceil$ data pages. Instead, it is only required that a container with less than $\lceil c/2 \rceil$ pages has a left sibling container such that both have more than c pages. Such a pair of containers is also called a *reorganization pair.* When a container overflow occurs, the split algorithm performs similarly to the one introduced above. First, some space for at least r data pages is allocated on a new cylinder. Here, r is a parameter, $1 < r \leq \lceil c/2 \rceil$, which is set during initialization of the CB^+-tree. It specifies the maximum number of pages involved in such a *reorganization step.*

Then, the r rightmost data pages of the container are moved to the newly allocated cylinder. Typically, r will be much smaller than $c/2$. Therefore, several reorganization steps have to be performed to distribute the pages of a reorganization pair evenly over its containers. Such a reorganization step is triggered when a new page is inserted in one of the containers of the reorganization pair. Then, the r rightmost pages (with respect to the key-order) are moved from the left container to the right container. Accordingly, r references are moved from the left subpage into the right. Thereafter, the corresponding directory page is written back to disk. For each reorganization step, two multi-page requests are only required to read and write the r pages. If r is set to a value considerably smaller than $c/2$, the cost for performing a reorganization step is substantially lower than the cost of a complete reorganization. However, the total cost for performing all the required reorganization steps increases with an decreasing value of r.

4.3 Splitting of a Directory Page

The split of a data page is treated in the CB$^+$-tree as in an ordinary B$^+$-tree, whereas the split of a directory page slightly differs from the one in a B$^+$-tree. An important goal in the design of the CB$^+$-tree is to preserve the property of the B$^+$-tree that insertions are processed on a single path of the tree. This property is essential for many of the well-investigated concurrency control protocols [LY81]. In order to achieve this goal, the CB$^+$-tree stores both subpages of a reorganization pair in a common directory page.

The basic idea of the new split algorithm is to relax the property of the B$^+$-tree that a split has to distribute entries evenly. Instead, the CB$^+$-tree consider only those possibilities for a split which do not cut a subpage or a pair of subpages which belong to a reorganization pair. Among all these possibilities, the split is selected which distribute the entries most evenly among the two directory pages. Consequently, there might be directory pages less than half full. In the worst case, our approach can only guarantee that $\lceil (b-c)/2 \rceil$ entries are stored in a directory page. This might have an impact on the number of the directory pages as well as on the height of the tree.

4.4 Range Queries

The range query is one of the most frequently used selection queries in a DBS. For a dynamic file, range queries are "efficiently" supported by B^+-trees if the cost of the query is only expressed in the number of qualifying pages. The classical algorithm for performing range queries in B^+-trees sequentially (in order of the search key) traverses the data pages where answers to the query can be found. Consequently, the algorithm reads one page at a time and therefore is not able to exploit multi-page requests.

The reading strategy of "one page at a time" has a serious impact on the performance, as the example has shown in section 3. When the data pages are not clustered with respect to the order of the search key, the probability is high that adjacent data pages are stored on different cylinders of the disk. The expected cost for reading the next (in key order) data page will be close to the average access time of the disk. When adjacent pages are clustered on disk (preferably stored on a common cylinder) a seek can generally be avoided and therefore, the expected cost of a disk access can be reduced. Several experiments with IBM's research prototype R* [ML86], for example, have confirmed that clustering of data pages reduces the I/O cost of range queries. In these experiments, VSAM was used as the underlying access method.

Our approach for performing range queries is different from previous ones. Our key observation is as follows: In addition to clustering, the cost of a range query can be substantially reduced when the required data pages are read by issuing multi-page requests. However, it is important for a low response time of a multi-page request that the target pages are read according to an efficient schedule.

The algorithm for performing a range query in a CB^+-tree is given as follows. First, the range query is initiated by an exact match query where the search key corresponds to the lower search key of the specified range. This results in traversing a path in the CB^+-tree. In contrast to the B^+-tree, the traversal process already stops at the lowest index level directly above the leaves of the tree. Then, subpages are sequentially processed until an index entry is found whose (separator) key is greater than the upper search key of the specified range (or the last index entry is found). For each subpage, the (page) references of the qualifying entries are given as input to one multi-page request which retrieves the qualifying data pages of the container.

If the order of the answers is of no concern, the data pages can immediately be processed when they are in main memory. Otherwise, the query has to wait, (at least) until the page which belongs to the leftmost entry in the subpage is in memory. In order to avoid processing delays, we suggest using double buffering in the CB^+-tree. There are two buffer areas, each of them containing c buffer frames, where c denotes the capacity of containers (subpages). First, the (qualifying) data pages of the first container are read into the first buffer. Then, while the pages of the second container are being read into the second buffer, the answers from the data pages of the first buffer are reported in key order. While the data pages of the third container are being read into the first buffer, the records can be processed in the second buffer, and so on. For range queries whose answers are reported in key-order, double buffering compensates for the delays caused by the fact that an efficient read schedule does not necessarily read data pages in key-order anymore.

This policy of query processing is illustrated in Figure 4. For the sake of simplicity, a view on a subtree is given that consists of two directory pages and several data pages. Let us assume that data pages $A, \ldots, L$ are required for answering a range query. The first request reads data pages A, B, and C into the first buffer. The second request reads data pages D, E, and F into the second buffer. The read schedule of that request was (E, F, D) and therefore, the pages are not ordered anymore with respect to their keys. While these pages are being processed, the first buffer receives pages from the third read request. The snapshot in Figure 4 shows that page H is already in the first buffer, whereas pages G and I still have to be retrieved.

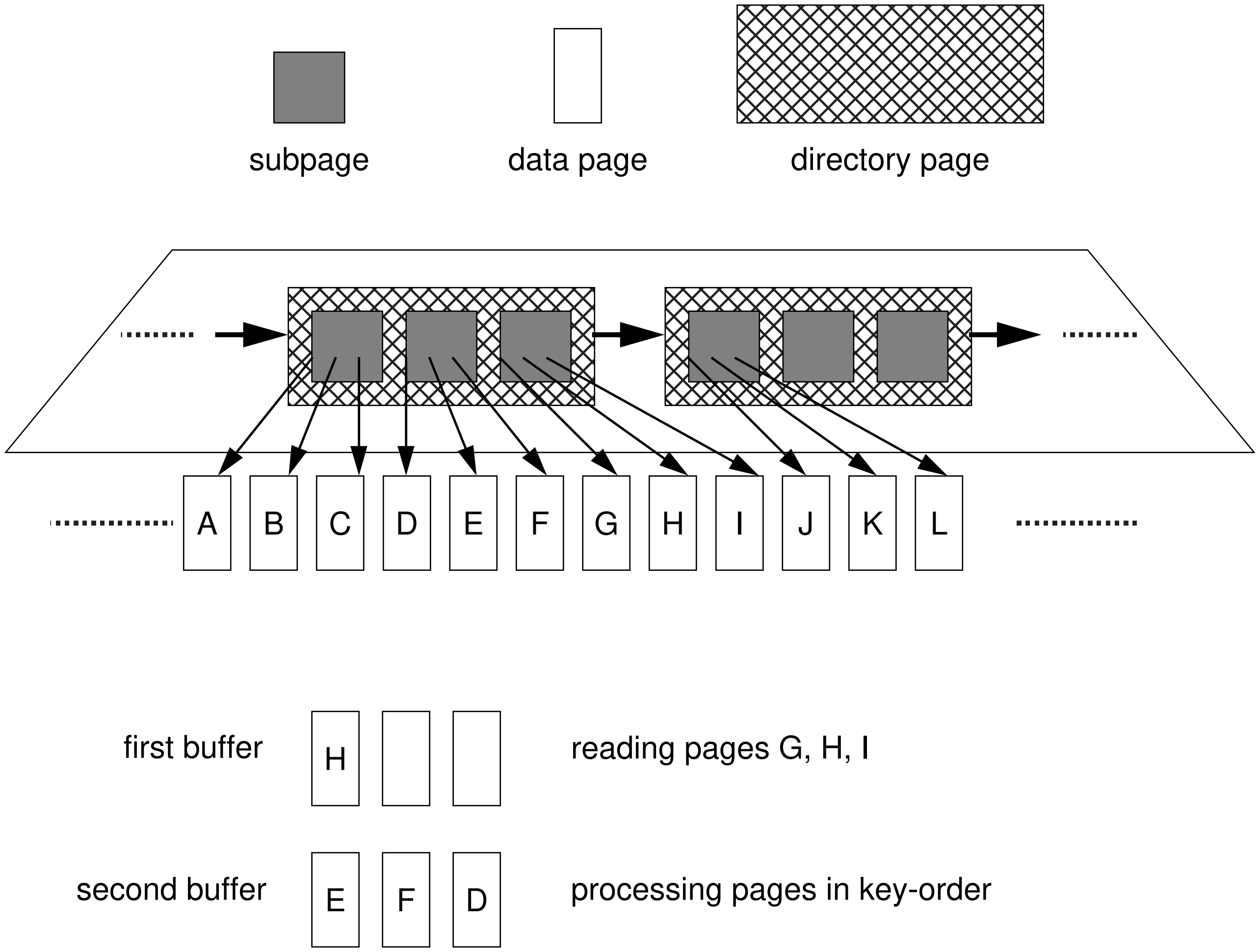

Figure 4: Double buffering for performing range queries in a CB^{+}-tree

	#records	dir. pages	data pages	I/O (dir.)	I/O (data)
CB$^+$-tree	100,000	69	7105	19,452	201,172
B$^+$-tree	100,000	65	7105	19,219	193,521
CB$^+$-tree	200,000	129	14237	81,803	415,093
B$^+$-tree	200,000	129	14237	80,719	399,481

Table 1: Performance for creating some B$^+$-trees and CB$^+$-trees

5 Experimental Performance Comparison

In this section, we report the results of a performance comparison of the B$^+$-trees and CB$^+$-trees. All results (which are related to the I/O cost) were obtained by simulating the disk accesses. In all of our experiments, we used a buffer of 100 pages which followed the least recently used policy. Moreover, the capacity of data pages and directory pages was 20 and 160 in our experiments, respectively. Experiments were also performed with other page capacities, but the results were in agreement with the ones obtained for the above setting.

The objective of our first set of experiments was to find out how much the insertion and storage cost are affected by the techniques of multi-page requests and clustering. The corresponding results are reported in the first subsection. In the second subsection, the response time of range queries is examined for both B$^+$-trees and CB$^+$-trees under the assumption of the disk model which has been introduced in section 2.

5.1 The Cost of Building up

In our first experiment, B$^+$-trees and CB$^+$-trees were created by inserting the same set of records, one record at a time. The records are uniformly distributed. The trees differed only in how data pages were clustered on disk. The B$^+$-tree did not provide any clustering (i.e. data pages are randomly distributed over the disk), whereas the CB$^+$-tree dynamically clustered the data pages with respect to the algorithm given in section 4. The capacity of the containers in the CB$^+$-tree (c) was 32. The number of data pages involved in one reorganization step (r) was assumed to be 8. Let us mention that a pessimistic cost estimation is made for the CB$^+$-tree. The cost of copying r pages from one cylinder to another is assumed to be equivalent to the cost of performing $2r$ ordinary disk accesses (under the assumption that none of the pages is in the buffer). This is considerably higher than the actual cost for performing the two multi-page requests.

Table 1 shows the results obtained from two of our experiments. The second column refers to the number of records in the tree after it was completely built up. The third and fourth column give the number of directory and data pages in the tree, respectively. Finally, the number of disk accesses (read and write) to directory and data pages are reported in the fifth and sixth column, respectively. As it should, the number of data pages is the same for both trees, whereas the number of directory pages is only slightly lower for the original B$^+$-tree (in one of the experiments). Moreover, the number of disc accesses required for building up the structure is about 4% higher for the CB$^+$-tree compared to the B$^+$-tree. The difference in the number of disk accesses to data pages can be explained as follows. About half of the data pages remain on the cylinder where they were originally created. The other half is copied onto a different cylinder. Since a copy operation consists of a read and a write request, the number of data pages provides a good estimate for the number of additional disk accesses required by the CB$^+$-tree.

Overall, insertion cost and storage cost is only slightly higher for the CB$^+$-tree compared to the original B$^+$-tree. Thus, we conclude that the cost overhead of clustering data pages is very low.

5.2 Range Query Performance of B$^+$-trees and CB$^+$-trees

In the next set of experiments, we examined the range query performance of the access methods. We restrict our experiments to a data file with 100,000 uniformly distributed records. For data-intensive queries, the number of qualifying index pages is considerably lower than the number of qualifying data pages. Moreover, many of the index pages are expected to be already in the buffer. For the purpose of this paper, therefore, accesses to index pages are largely disregarded in the sequel and the following results consider only the cost for retrieving qualifying data pages.

In the following, we consider range queries of size α, $0 < \alpha \leq 1$. The size of a range query refers to the ratio of expected number of answers to number of records in the file. In our experiments, the cost for performing a range query of size α is computed by taking an average over 1000 queries. More precisely, we proceed as follows. For each run consisting of 1000 range queries of size α, there is a counter u for the seeks and an array v whose i-th component v_i, $1 \leq i \leq c$, counts the number of multi-page requests of size i. First, these parameters are initialized ($u = 0$ and $v_i = 0$ for $1 \leq i \leq c$), and the buffer is reset. Then, the queries are performed one at a time. For each of the range queries, the value of u is incremented by the number of containers which include at least one of the required data pages. If l, $1 \leq l \leq c$, pages of the container are qualifying pages, the value of v_l is incremented by one. Let q be the total number of qualifying pages of the 1000 range queries; the cost for reading a qualifying page is then computed by

$$\frac{u * seek_{avg} + \sum_{j=1}^{c} v_j * Cost(j)}{q} \qquad (1)$$

Here $Cost(j)$ gives the expected cost of a multi-page request which retrieves j uniformly distributed pages from a cylinder of the disk. In [See94], such a cost function was derived and shown to be in agreement with the results obtained from a real disk.

In the following sets of experiments, the disk parameters are specified as given in the first rows of Table 2. The default values are adopted from the Fujitsu Eagle disk which is frequently used for experiments of this type. The essential difference to the Fujitsu Eagle is only that head switch time is neglected. The other rows of Table 2 refers to the parameters of the CB$^+$-tree. In the sequel, we report the results of several sets of experiments. For each set, the cost is given as a function of a parameter, whereas the values of the other parameters are equal to their their

parameter	symbol	default value
#cylinders	Cyl	840
#tracks	TC	20
#pages per track	PT	8
#sectors per page	SP	8
sector size [Byte]		512
avg. seek time [ms]	$seek_{avg}$	18
avg. rotational delay [ms]		8
page transfer time [ms]		2
head switch time [ms]	hst	0
capacity of data pages	a	20
capacity of directory pages	b	160
capacity of containers	c	32
size of reorganization step	r	$\lceil c/2 \rceil$
query size	α	0.5%
number of records		100,000

Table 2: Parameters of the disk and the CB$^+$-tree

default values.

5.2.1 Varying the query size

In our first set of experiments on range queries, results vary with the size of the range query from 0.025% to 0.5%. A range query of size 0.1% (i.e. 100 answers are expected), for example, requires on the average 8.1 data pages. The results of our experiments have been plotted in Figure 5. In addition to the graph of the total cost, three graphs are given for the cost components of transfer time, rotational delay and seek time.

As expected, the total cost for retrieving a qualifying page of a range query from the CB$^+$-tree decreases with an increasing size of the query. Let us take a closer look to a range query of size 0.5% (500 answers). The cost to read a qualifying page is less than 5 ms. The expected number of qualifying pages is 36.36 and therefore, the total I/O-time of the query is roughly 182 ms. In comparison, an ordinary B$^+$-tree would require the average access time for reading a qualifying page (28 ms) and therefore the I/O-time of the same query would be 1018 ms.

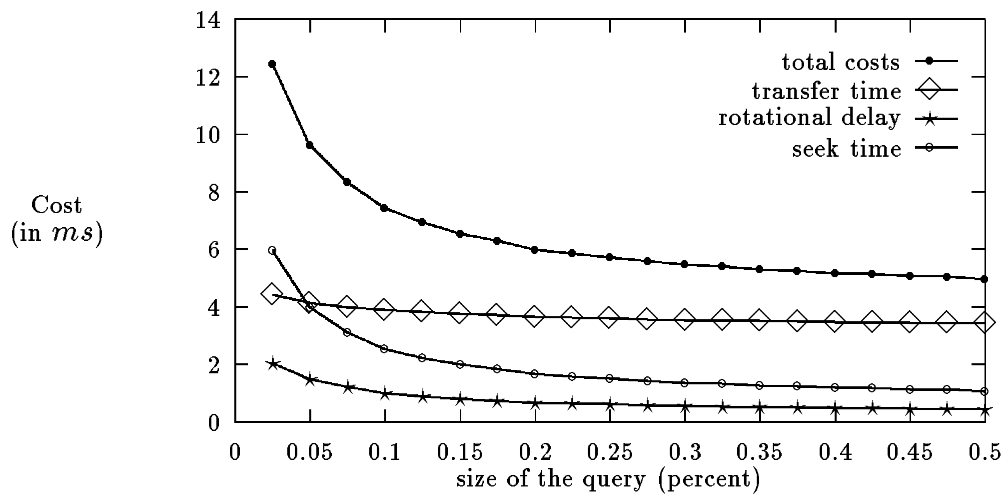

Figure 5: Cost for retrieving a qualifying page of a range query (as a function of the query size)

Another interesting observation is related to the cost components. In contrast to an ordinary (single) page request, the transfer time generally dominates the total cost. For large range queries, it is by a factor of three higher than the seek time. Only for small range queries it can be observed that the seek time is higher than the transfer time.

5.2.2 Varying the capacity of the containers

In our second set of experiments, results vary with c, the capacity of the containers (subpages). A split of a container was performed in a single reorganization step (i.e. $r = c/2$). The size of the range queries is 0.5% (about 500 answers). In these experiments, there is an average number of 35.9 target pages for a range query. In Figure 6, the cost has been depicted for retrieving a qualifying page of a range query. There are again four graphs which show the total cost, transfer time, rotational delay, and seek time.

As expected, the total cost decreases with an increasing value of c. The cost rapidly improves up to $c = 10$. For containers of capacity 10, a qualifying page of the query is read in 8 ms. For larger capacities, the performance still improves, but at a slower rate. The cost for $c = 32$, for example, corresponds to 5 ms which is 60% less than the cost for $c = 10$.

Overall, the results of these experiments impressively demonstrate the benefits of clustering data pages. Even when only two adjacent pages are kept on the same cylinder, the cost for range queries is substantially reduced compared to using no clustering of data pages.

One remaining question is how much does the read schedule of a multi-page request influence the cost of a range query. First of all note that the read schedule does not affect the seek time of a range query and therefore seek time remains the same as plotted in Figure 6. In comparison to our schedules which follow the SLTF policy, we consider random read schedules in the following. Under the assumption that data pages are randomly distributed on a cylinder, the cost for reading the next page in a random schedule corresponds to the sum of average rotational delay and transfer time (10 ms). For $c > 8$, the cost of retrieving a qualifying page of a range query is then by at least a factor of two higher than the cost when the read schedules are computed according to the SLTF policy.

5.2.3 Varying the size of the reorganization steps

The results of the previous experiments are obtained from a CB^+-tree whose containers are split in a single reorganization step (i.e. $r = c/2$). This case is most appealing to range queries, but a long processing time may occur for an insertion of a record. The question is therefore how much does the parameter r influence the performance of range queries. The results on the cost of range queries are reported in Table 3. The second row refers to the size to a range query (in percent) and the first column indicates the number of pages involved in a reorganization step. Consider a range query of size 0.1%. The cost of retrieving a qualifying page is 7.762 ms for $r = 2$, whereas for $r = 16$ the cost is 7.654 ms, that is only an improvement of 1.5%. A similar effect can be observed for other query sizes. Thus, we conclude that query performance is almost independent from the setting

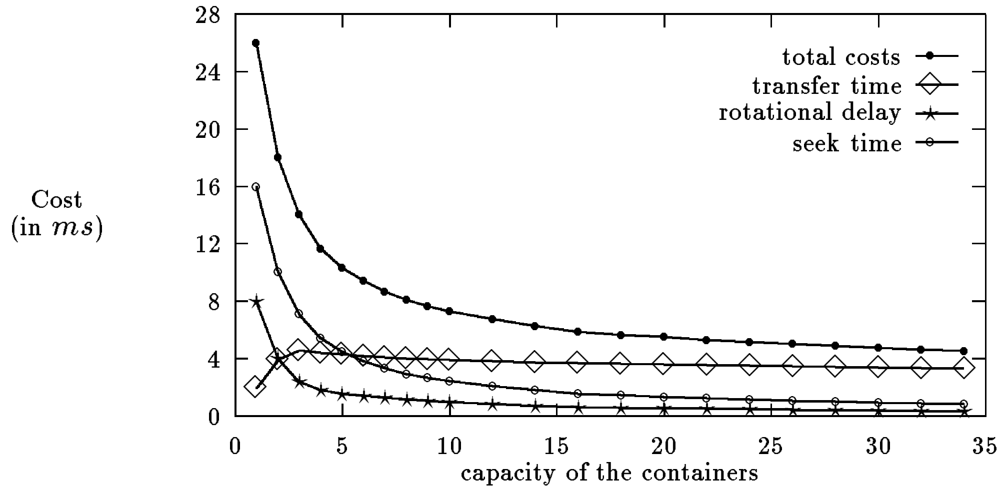

Figure 6: Cost for retrieving a qualifying page of a range query (as a function of the capacity of the containers)

r	size of the range query					
	0.05	0.1	0.2	0.3	0.4	0.5
2	10.198	7.762	6.174	5.544	5.312	5.064
16	10.048	7.654	6.082	5.510	5.180	4.954

Table 3: Cost for retrieving a qualifying page of a range query ($r = 2, 16$)

of r.

5.2.4 Varying the head switch time

Our previous studies were done under the assumptions of an idealized disk model which neglects head switch time. This assumption is however not in agreement with todays disk technology. Therefore, we present in this section a preliminary performance study on range query performance when head switch time contributes to the total cost.

In Figure 7, the results are reported for the cost of reading a qualifying page of a range query of size 0.5% (500 answers). The cost depends on the head switch

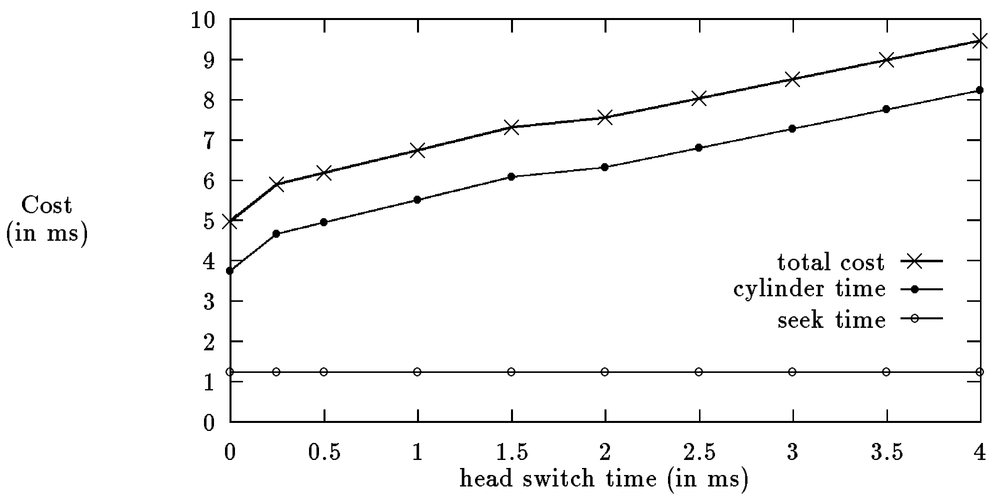

Figure 7: Cost for retrieving a qualifying page of a range query (as a function of the head switch time)

time (*hst*) which varies from 0 to 4 *ms*. Disks with head switch time higher than 4 *ms* are seldom found in practice. There are three graphs in Figure 7. The one shows the total cost, whereas the others refers to the cost components. In this experiment, the sum of seek time and cylinder time yields the total cost. The *cylinder time* is defined as the sum of transfer time and rotational delay.

First of all, let us emphasize that the results confirm the trends observed in our previous experiments when head switch time was neglected. There is a substantial performance improvement of the CB$^+$-tree in comparison to the B$^+$-tree. For a head switch time of 1 *ms*, the total cost of the CB$^+$-tree is about 6.5 *ms* for reading a qualifying page, whereas an ordinary B$^+$-tree requires 28 *ms*. The graph for the total cost shows an almost linear dependency from the head switch time. Only the value obtained for $hst = 0$ behaves differently.

The graphs of the seek time and the cylinder time indicate that the influence of cylinder time on the total cost increases with an increasing value of *hst*. This is because head switch time does not affect seek time in our experiments. In practice, however, there is a correlation between seek time and head switch time.

The seek time contains a component, called the settle time, which is about two to three times higher than the seek time [RW94]. Also if we would have taken this correlation into account our statement would be still the same.

Overall, we conclude that cylinder time dominates query processing cost in most of our experiments. In order to improve query performance, we would not be primarily interested in disks with a low seek time, but in disks with a low cylinder time.

5.2.5 The impact of clustering pages in a cylinder

So far, we assumed that the data pages are randomly distributed over all tracks on a cylinder. This is obviously a pessimistic assumption in the sense that both average-case and worst-case performance is expected to improve when pages are distributed in a more sophisticated way. Note that the local organization of pages on a cylinder does not have any influence on the seek time. Moreover, each qualifying page has to be transferred from a cylinder into main memory which requires at least the transfer time (2 ms). Thus,

$$\frac{u * seek_{avg}}{q} + 2$$

can be used as a lower bound for the expected cost of range queries (formula 1).

In order to improve the local organization of pages in a cylinder, we investigated the following simple strategy: pages are not distributed among all the tracks, but only on a few of them. A parameter t, $1 \leq t \leq TC$, refers to the number of tracks on which data pages are randomly distributed.

We performed experiments with a CB$^+$-tree using the same parameter settings as previously introduced ($a = 20$, $b = 160$, $c = 32$, $r = 16$, 100,000 records). In Figure 8, the cost has been plotted as a function of the query size from 0.025% to 0.5%. Two of the graphs show the results when the strategy is used for $t = 20$ and $t = 4$. Note that the capacity of the containers is 32 and therefore, pages have to be distributed on at least four tracks ($t = 4$). Hence, $t = 4$ gives the best performance for this strategy. A third graph shows the lower bound of the cost.

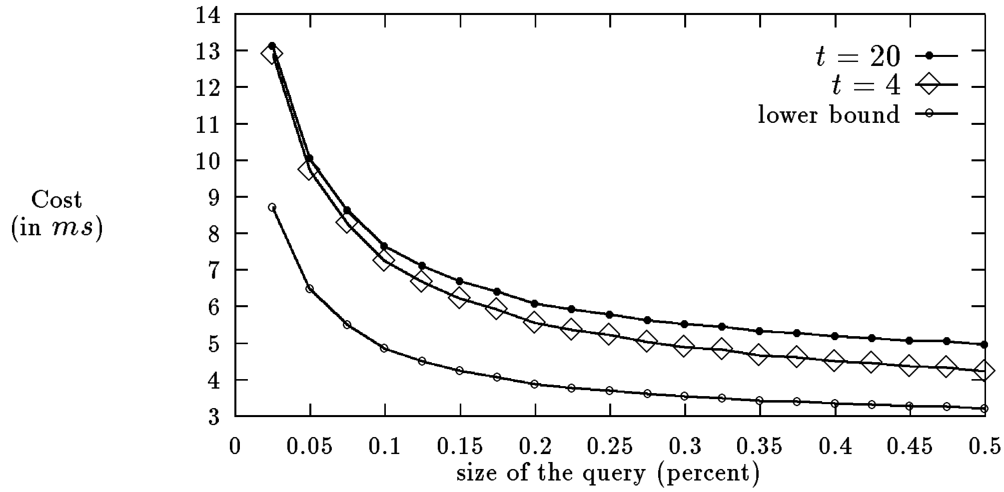

Figure 8: Cost for retrieving a qualifying page of range queries ($t = 4, 20$)

The first two graphs show very similar performance for small range queries, although the corresponding methods substantially differ in the number of tracks where target pages are stored. For larger range queries, the performance is better for $t = 4$, but there is still a considerable distance to the lower bound.

6 Conclusion

In this paper, we considered the problem of dynamic access methods which are primarily designed to support data-intensive queries efficiently in a dynamic setting (frequently occurring insertions and deletions of records). In general, such queries require a large number of pages from a magnetic disk drive and processing cost is dominated by the time spent on the disk. In contrast to most of the previous approaches to access methods, we have been not only concerned about the number of page accesses required for performing a query, but particularly about the cost of the individual page accesses. In order to reduce the cost of individual page accesses, we proposed to use two techniques: clustering and multi-page requests. The technique of clustering takes care of storing pages on one cylinder which are frequently required by queries in common. It is basically responsible for reducing the seek operations on a disk. However, clustering does not result in reducing the rotational delay which has, for a modern disk drive, almost the same impact on the access cost as seek time. In order to reduce the rotational delay of page requests, we proposed to process a query such that a single multi-page request reads all qualifying pages from a cylinder. A multi-page request retrieves these pages with respect to the shortest-latency-time-first (SLTF) policy. Under certain assumptions, this policy produces close to optimal read schedules [SF73].

For sake of concreteness, our discussion was based on the problem of supporting range queries on B^+-trees. We presented a modified data structure, called the CB^+-tree, which exploits both techniques clustering of data pages and multi-page requests. In order to cluster data pages, we introduced the concept of containers which refers to a set of adjacent (with respect to the ordering of the key) data pages. Similar to a page, a container has to be split when it comprises of too many pages. The algorithm for performing range queries on a CB^+-tree is different to the one of the B^+-tree. Instead of reading data pages from magnetic disk one at a time, the qualifying pages in a container are read with one multi-page request. In an experimental performance comparison, we first showed that the insertion cost and the storage cost of the CB^+-tree is only marginal higher than the costs of the original B^+-tree (which does not provide any clustering of pages). Then, the performance of range queries was examined under various parameter settings. The experiments showed that the cost for processing range queries can be by several factors lower for the CB^+-tree compared to the B^+-tree. Such substantial performance improvements can already be observed for a small capacity of the containers (32 pages) and for a rather small number of qualifying pages.

The techniques of clustering and multi-page request are not restricted to B^+-trees, but can also be used for other access methods. First of all, access methods directly derived from the B^+-tree can immediately follow the approach presented in this paper. This class includes for example the z-order based B^+-tree [OM83], a multidimensional access method, and the data structure of the EXODUS storage manager [CDRS86]. The R-tree family [Gut84] represents another important class of access methods which can easily be adapted to exploit these techniques for supporting window queries and other proximity queries efficiently.

Acknowledgement

I would like to thank Thomas Brinkhoff and Ralf Schneider for their comments on an earlier version of this paper.

References

[BK94] T. Brinkhoff and H. P. Kriegel. The impact of global clustering on spatial database systems. In *Proc. of the Conf. on Very Large Databases (VLDB)*, 1994.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD*, pages 322–331, 1990.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[CDRS86] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 91–100, 1986.

[Den67] P. J. Denning. Effects of scheduling on file memory operations. In *Proc. AFIPS*, pages 9–21, 1967.

[Gra91] Jim Gray, editor. *The Benchmark Handbook.* Morgan Kaufmann, 1991.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD*, pages 47–57, 1984.

[HBP+81] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana, and L. G. Taft. A quater century of disk file innovation. *IBM Journal of Research and Development*, 25(5):677–689, 1981.

[HSW88] A. Hutflesz, H.-W. Six, and P. Widmayer. Globally order preserving multidimensional linear hashing. In *Proc. of the IEEE Conference on Data Engineering*, pages 572–579, 1988.

[JW91] D. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Hewlett-Packard, 1991.

[KL74] D. G. Keehn and J. O. Lacy. VSAM data set design parameters. *IBM System Journal*, 13(3):186–212, 1974.

[Knu73] Donald E. Knuth. *The Art of Computing, vol. 3 – Sorting and Searching*. Addison-Wesley, 1973.

[LY81] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[MJLF84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[ML86] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proc. of the ACM SIGMOD*, pages 84–95, 1986.

[OM83] J. Orenstein and T. Merrett. A class of data structures for associative searching. In *Symp. on Principles of Database Systems*, 1983.

[O'N92] P. E. O'Neil. The SB-tree an index-sequential structure for high-performance sequential access. *Acta Informatica*, 29:241–265, 1992.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[See94] B. Seeger. An analysis of schedules for performing multi-page requests. Technical Report 94-16, Institut für Informatik, Universität München, 1994.

[SF73] H. S. Stone and H. Fuller. On the near-optimality of the shortest-latency-time-first drum scheduling discipline. *Communications of the ACM*, 16(6):352–353, 1973.

[Wei89] G. Weikum. Set-oriented disk access to large complex objects. In *Proc. of the IEEE Conference on Data Engineering*, pages 426–433, 1989.

[Wil94] M. Wilkes. Operating systems in a changing world. *Operating Systems Review*, 28(2):9–21, 1994.