Multi-Disk B-trees*

Bernhard Seeger† and Per-Åke Larson

Dept. of Computer Science, University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Abstract

In this paper, we consider how to exploit multiple disks to improve the performance of B-tree structured files. Attention is paid both to the response time of individual operations and to the throughput of the system in a multi-user environment. We begin with a survey of three different approaches to designing multi-disk B-trees: distributing records among disks, using large multi-disk pages, and distributing pages among disks. For each approach, several alternatives are discussed and their main advantages and disadvantages are identified. We then propose a new scheme, based on page distribution, that is intended to provide a better local balancing of the request load than previous schemes. Preliminary performance results confirm that this improves both response time and throughput.

1 Introduction

The increase in processor speed over the last few years has been nothing short of spectacular. The performance of disks has also improved but not at the same rate, resulting in a severe I/O bottleneck for many applications. The discrepancy between processor performance and I/O performance is further compounded in multiprocessor systems. Database systems are I/O intensive and I/O performance is critical to overall performance. Given that the performance of individual disk units is unlikely to improve significantly, we need to consider how to efficiently exploit multiple disk units. In this paper we investigate how to parallelize B-trees, that is, how to distribute the data stored in a B-tree over multiple disks. We assume a transaction-processing type of environment and our main focus is on overall throughput of the system. We consider three different approaches to designing multi-disk B-trees:

- *record distribution:* each disk contains a complete B-tree and records are assigned to the disks by hashing or some other technique
- *large page B-trees:* each page of the B-tree is distributed over the disks
- *page distribution:* the pages of an ordinary B-tree are distributed among the disks by some technique

The standard operations for B-trees are insertion, deletion, modification, exact match query, and range query. The objective is to improve both the response time of individual operations and, more importantly, the throughput of the system. We define the response time of an operation as the time it takes to complete the operation. The throughput is the number of operations per second. (This is well defined only if the mix of operations is specified.) In the sequel, we make the assumption that the system is I/O bound – not an unlikely scenario with recent increases in processor speed. When multiple disks are used, the response time of an operation is affected by several factors: the number of disk accesses required, whether or not the required accesses can be performed in parallel, how evenly the accesses are distributed over the available disks, and the amount of time I/O requests wait before being serviced. For a given mix of operations, throughput depends mainly on the total number of disk accesses required for an operation and how well the access load is balanced over the disks. (Needless to say, response time and throughput are strongly related.) From this it follows that we need to pay attention to the following factors in the design of a multi-disk B-tree.

- resource usage, that is, how many disk accesses (and how many disks) will be required for an operation
- balancing of the access load, both locally for each operation and globally for a set of operation
- degree of parallelism, that is, whenever possible perform the accesses required in parallel

When considering a particular approach, we must consider its effect both on response time and throughput. Focusing solely on improving the response time for an operation may lead to solutions that are disastrous for throughput, and vice versa. Improving the response time often translates into using more resources for an operation but this reduces the throughput. This, in turn, may actually increase the response time by increasing the waiting time.

Most previous work on exploiting multiple disks [CABK 88, Kim 86, LKB 87, SM 86, PGK 88] was done without consideration of specific file structures. Some work has been done on multidimensional access (index) methods. Interesting results have been obtained for interpolation hashing [WB 87], the grid file [HL 90] and methods based on bit interleaving [Du 86, FM 89, KP 88]. However, the most widely used file structure, the B^+-tree [BM 72, Com 79], has been paid less attention. Pramanik and Kim proposed the PNB-tree [PK 90], which is a large page B-tree intended for synchronized disks.

This paper consists of two parts. The first part of the paper is a survey of different approaches to designing a multi-disk B-tree. For each approach, we discuss several possible variants and attempt to identify their main advantages and disadvantages. In the second part of the paper, we then propose a new design for a multi-disk B-tree. The design is based on page distribution and the novel part is an algorithm for assigning pages to disks so as to achieve a better balance of the access load. In the last section of the paper, we report the results of a preliminary performance comparison. The results indicate that our method achieves its objectives and provides high throughput.

2 I/O architecture

The I/O performance depends on the architecture of the I/O system. In this section we briefly outline two basic architectures: synchronized disks and independent disks. For database applications, it is important to distinguish between two measures of the performance of an I/O system: transfer rate and seek rate. Transfer rate is simply the total (sustained) rate at which data can be transferred between main memory and disks, typically measured in megabytes per second. This is the measure of most importance for applications characterized by large requests. Seek rate is the total number of (independent) disk seeks per second that the I/O system is capable of performing. This measure is the more important one for applications dominated by many small requests.

Consider p fully synchronized disks. We assume that the physical position of the I/O heads is exactly the same on each one of the p disks. That is, we assume that the disk or controller hardware synchronizes not only the movement of read/write heads but also the rotation of the disks so that each head is at the same position of the track. Rotational synchronization can be done to within a few degrees [Sie 90]. Synchronized disks of this type are commercially available but tend to be fairly expensive. Every read or write request involves every disk: each disk reads/writes the same sectors from the same cylinder and track. In principle, p synchronized disks behave like a large single disk with storage capacity and transfer rate p times that of the individual disks. However, the positioning time and the seek rate are the same as for the individual disks. Synchronization reduces the average access time significantly. Consider the situation when each page of a file consists of, say, one sector on each disk. If the disks are synchronized, the access time is the same on each of the p disks. If the disks are not synchronized, we have to wait until all disks have completed. The total access time is thus determined by the "slowest" disk. Synchronized disks perform well in an environment with many large requests (because of the high transfer rate) but offer little improvement in an environment dominated by small requests (because the seek rate is not improved).

Now consider p independent disks. We assume that the user has control over on which (physical) disk a page is stored. (On some systems, the user's view of the disks does not necessarily match physical reality.) Each disk has its own request queue and we assume that the scheduling for each disk is independent of the scheduling for the other disks. The performance of the I/O system depends on its architecture, in particular, on the number of available data paths. In a typical system, several disks are connected through a common bus (channel) to a disk controller, which has a high-speed connection to main memory. The data transfer rate of the channel is normally close to the transmission rate of a disk. Usually, several seeks (one for each disk) may be in progress simultaneously. When a disk has finished its seek and is ready to send or receive data, the channel may be blocked by another disk. If so, the disk has to wait a complete revolution until it can attempt to send or receive data again. This type of channel contention may reduce the effective transfer rate very significantly. Channel contention can be reduced by providing buffer space on each disk unit and, possibly, also increasing the transfer rate of the channel. The disk can then read data into its buffer and sends it, possibly in smaller units, whenever the channel is free. The success of this technique depends on the amount of buffer space provided. Channel contention can be avoided completely by having a separate channel for each disk. This is a more expensive solution. The effect of channel contention is very difficult to model and is ignored in the rest of the paper.

3 Performance measures

We consider the standard B-tree operations. We refer to insertions, deletions, modifications, and exact match queries as *single record operations* and range queries as *multi record operations*. It is also useful to distinguish between small and large range queries (defined at the

size of the data file	height of the index	main memory	accesses
1.1Mb	1	4Kb	1
300.0Mb	2	4Kb	2
2000.0Mb	3	44Kb	2

Table 1: Number of accesses B-trees require to perform an exact match query depending on the size of the file and main memory

end of the section).

In the sequel we make the simplifying assumption that each page is read or written individually, that is, always requires a disk access. In practice, one would try to exploit set-oriented retrieval whenever possible, that is, if two or more physically adjacent pages are required they would be read with a single disk access. How much is gained by set-oriented access depends on how pages are physically clustered on disk. At this point we do not want to make any assumptions about the effects of clustering.

On a lightly loaded system, there does not seem to be any way to significantly reduce the response time of single record operations by exploiting multiple disks. For single record operations, an improvement in response time can only be achieved by reducing the height of the tree. As mentioned in [Sal 88], in practice the height of a B-tree index is usually not more than two. Under the assumption that the root is kept in main memory, all exact match queries can be handled in two disk accesses and most insertions and deletions in three or four disk accesses.

Let us assume 4K bytes per page, 10 bytes per index entry (this seems realistic for a prefix B-tree) and an average utilization of ln 2 in the data pages of the B-tree. First, consider a B-tree that has 1 index page. Because the height of the tree does not change until the index page is completely filled, this B-tree can hold at most 400 data pages, each data page covering 4Kb. Thus, the size of the corresponding data file[‡] is approximately 1.1Mb. For larger B-trees, we assume that the utilization of index pages is also ln 2 (except the root page). Then we can expect an average of 275 entries per index page. A two-level index will contain, at most, 401 index pages and cover 110,000 data pages, corresponding to a data file of about 300Mb. Most files are much smaller than 300Mb but much larger files do exist. However, for such files, we can typically afford to keep more index pages in main memory. To achieve two-access retrieval for a 2 Gb data file, for example, would require keeping only 11 index pages (44Kb) in main memory. A summary is presented in Table 1.

On a heavily loaded system, the response time for single-record operations may be significantly affected by queuing delays. Each I/O request enters the request queue for the appropriate disk and remains there until it can be serviced. The key to improving response time and throughput in this situation is to balance the request load as evenly as possible over the available disks.

A range query involves scanning some number of data pages. Even on a lightly loaded system, the response time can be improved by attempting to read in pages in parallel. As we will see, this can be achieved in several different ways. On a heavily loaded system, load balancing is still key to improved performance. However, it also becomes important not to increase the request load on the I/O system by reading more data than required to answer the query.

After these general remarks, we now introduce some concrete cost measures. For a given query q, let $\Psi_i(q)$ denote the number of accesses to disk i for retrieving data pages, $1 \leq i \leq p$. We will use $\Psi_{max}(q) = \max_{1 \leq i \leq p} \Psi_i(q)$, the maximum number of accesses to a disk, as an approximate measure of the response time of a query. For the purpose of this paper, accesses to index pages are not important and are therefore largely disregarded in the sequel. The total number of disk accesses to retrieve the required data pages is given by $\Psi_{sum}(q) = \sum_{1 \leq i \leq p} \Psi_i(q)$. For a given query q, the maximum improvement achievable by distributing the request over the disks is then

$$\Psi_{max}(q) = \lceil \frac{\Psi_{sum}(q)}{p} \rceil \qquad (1)$$

Normally, this cannot be achieved. The same measure can be defined for a set of operations Q. Then the following inequality holds:

$$\Psi_{max}(Q) \leq \sum_{q \in Q} \Psi_{max}(q) \qquad (2)$$

An operation does not necessarily require the same number of data pages from each disk. Therefore, a set of queries may result in a more uniform demand pattern for the different disks than in executing the queries one by one. Then performance of a set of operations depends on how uniformly the I/O requests are distributed over the different disks.

If the rate of I/O requests is high, queues will form in front of the disks. Thus, in addition to the pure access costs, we have to take into consideration the queue waiting time. Therefore the expected time (in I/O-units) to perform a query q is given by $max_{1 \leq i \leq p} \Psi_i(q) * (1 + W_i)$, where W_i denotes the average queue waiting time in I/O-units for the i-th disk.

Finally, we define the notion of *large* and *small* range queries more precisely. Let N be the number of data pages in an ordinary B-tree, su the storage utilization, and b the capacity of a data page (in number of records). We define a *small range query* to be a range query whose response set does not exceed b records. An ordinary B-tree can answer such a query with one or two requests for

[‡]Let us emphasize that here and in the following the data file does not refer to the space allocated but to the space actually occupied by the data records.

data pages. We define a *large range query* to be a range query whose response set consists of at least $p*b$ records. An ordinary B-tree would require (approximately) $\frac{p}{su}$ data pages to answer the query.

4 B-trees with Record Distribution

The basic idea underlying this approach is to use multiple B-trees, one for each disk, and distribute the records among these B-trees. Many algorithms for assigning records to disks are possible, giving rise to different variants of the basic approach. Typically, records do not migrate between disks; once a record has been assigned to a disk, it remains on that disk until deleted. This approach is intended for independent, unsynchronized disks. We will briefly outline different methods for assigning records to disks and discuss the effects on the basic B-tree operations.

Key range partitioning:
One method is to define a disjoint partition of the key range into p intervals $I_1, .., I_p$ during initialization of the file. A record is inserted in the B-tree on disk j, if its key value is in I_j. Single record operations are restricted to exactly one tree. A range query may involve several disks, depending on the size of the range. This distribution scheme has two serious drawbacks. Firstly, it is difficult to maintain both an even distribution of the data (same number of records on each disk) and an even request load on the disks. Balancing the request load requires prior knowledge of access patterns, necessitating gathering of statistical information. Furthermore, access pattern may not be stable. Secondly, many range queries (except very large queries) cannot be sped up significantly because the desired records will typically reside on a small number of disks.

Hash partitioning:
We can use a hashing function for assigning records to disks, that is, a hashing function is applied to the key of the record and the resulting value determines what disk the record is assigned to. This scheme does not suffer from the balancing problems of the previous scheme. A properly designed hashing function should result in each disk receiving approximately the same amount of data and also approximately the same request load. Single record operations are restricted to exactly one tree. Every range query, however small, will require at least one page from each disk. This has two drawbacks: a) it increases the response time for small range queries because we have to wait for the slowest disk and b) if small range queries are common, it may significantly increase the overall request load on the I/O system. For large range queries this scheme is very efficient. All disks will be used and each disk is expected to receive approximately the same number of accesses.

Round robin assignment:
Instead of partitioning the records based on key values, we can simply assign them in a round-robin fashion without regard to the key value. The i-th record to be inserted is assigned to disk $(i \bmod p + 1)$. In the absence of deletions, the data load will be perfectly balanced. One would also expect the request load to be approximately balanced. However, this scheme has the serious drawback that every operation involves all disks because we don't know *a priori* to which disk a record has been assigned. This not only increases the response time of an operation but it may also significantly increase the overall request load on the system. This drawback is shared by all schemes where the assignment of a record to disk is not determined by its key value.

The GAMMA data base machine [DGS 90] supports these three partitioning schemes for B-trees or any other index method. Recently, a hybrid range-partitioning scheme has been proposed to support both small and large range queries more efficiently [GD 90]. The basic idea is to divide the key range into many small partitions and to assign multiple partitions to each disk. Partitions are assigned to the disks in round robin fashion. The most serious concern about hybrid range partitioning is that it assumes a static data set as well as a static set of partitions.

The performance of a B-tree can be improved by buffering. For the ordinary B-tree, it is suggested to keep a path in main memory. In case of B-trees based on record distribution, buffer space for at least $p + 1$ pages is needed. All the roots and an actual page must be kept in main memory. This guarantees efficient processing of exact match queries. For range queries, both the root and (at least) one data page for disk should be kept in main memory, for a total of $2p$ pages of buffer space. If double buffering is used for each disk, this increases to $3p$ pages.

Among the schemes of this type, hash partitioning appears most promising, mainly because of its automatic load balancing. However, it is evident that it will not perform well in an environment with a high frequency of small range queries.

5 Large Page B-trees

The basic idea of this approach is to use very large pages, here called super pages, partitioning each page across the p disks. A super page thus consists of p subpages, each one stored on a different disk. Retrieval of a super page requires p reads, one read per disk.

Let us first consider the case when the p disks are unsynchronized. Assume first that records are kept in sorted order within a super page. Then single record operations require that the whole super page be read in. This inevitably leads to poor response time because every disk is involved and we have to wait for the slowest disk to complete its read. The throughput will also suffer because of the high request load on the I/O system. To overcome this problem, we must be able to determine on which subpage a record is stored so that we can read

in only one subpage. This can be accomplished by a method proposed by Litwin and Lomet [LL 86] called bounded disorder. This method uses hashing on the key value to assign records to subpages. (One subpage is used as an overflow area). Most exact match queries can now be answered by reading in a single subpage, involving only a single disk. Occasionally, a second subpage (the overflow area) has to be read in. Similarly, most insertions and deletions will only involve a single subpage. To answer a range query, a sequence of complete super pages must be read in. Even the smallest range query requires a complete super page to be read in. The processing of index pages is expensive in this scheme regardless of the internal organization of index pages. A single record operation requires traversal of the index, which amounts to checking one or two index super pages. All subpages of a super page will have to be read in, requiring one read per disk. Not only is this slow but it also increases the request load on the I/O system. For (large) range queries, the overhead caused by index processing is less of a concern.

Now consider large page B-trees using p synchronized disks. Pramanik and Kim proposed the PNB-tree [PK 90] based on this approach. PNB-trees differ slightly from conventional B-trees in that entries (data and index) of a super page are not necessarily in sorted order. However, within a subpage all entries are kept in sorted order. The major advantage of the PNB-tree stated in [PK 90] is the reduction of the height in comparison to the conventional B-tree. However, as discussed above, the height of an ordinary B-tree is seldom more than two or three. Moreover, range queries were not explicitly investigated in [PK 90].

In both environments, single record operations as well as small range queries require access to all p disks. This increases the request load on the system and reduces the query throughput. For single record operations, we are no better off than having a single disk unless the height is reduced. In fact, we are worse off because more buffer space is required. The main advantage of the PNB-tree is the improved performance for large range queries, which is a direct consequence of the higher data transfer rate.

In summary, the idea of using large pages spread over several disks appears promising only if the bulk of the load consists of large range queries. For single record operations and small range queries the resource usage is unnecessarily high.

6 B-trees with Page Distribution

The methods discussed in this section are based on the idea of using an ordinary B-tree but distributing the pages of the tree among the p disks. This requires that each index entry be expanded to indicate which disk the pointer refers to. This information would hardly ever take more than one byte so the resulting reduction on fanout of the index is minimal. We will only consider unsynchronized disks; combining this approach with synchronized disks does not seem to offer any advantage. Two different variants will be discussed: random assignment and round robin assignment.

The traditional way to perform a range query in a B-tree is to first execute an exact match query with the lower value of the specified range. Then data pages are traversed sequentially, reading in one page at a time. However, sequential processing does not exploit the parallelism offered by multiple disks. Therefore, the algorithm must be slightly modified. We suggest a two-phase algorithm. Using the lowest level of the index, we first extract pointers to all the data pages needed to answer the query. In the second phase, we fetch in parallel the required data pages from the disks. To be able to overlap reading and processing of the pages, two buffers per disk are needed, for a total of $2p$ page buffers.

Random distribution:
When a new page is created by splitting, we randomly assign it to one of the disks. This is applied to both index pages and data pages. Random distribution can easily be implemented by using a suitable random number generator or a hashing function taking, for example, the key of the first record of the page as input. The idea of distributing pages in a random fashion is applicable to many other access methods. In particular, the performance of multidimensional access methods such as the buddy-tree [SK 90] can be improved in this way without any complicated optimization algorithms.

This approach provides good (global) load balancing and none of the basic operations require more disk accesses than for an ordinary B-tree. On average, good response time and throughput can be expected. However, for an individual range query, the response time is determined by the slowest disk, i.e. the one that requires the maximum number of accesses to answer the query. In the worst case, all the required pages may reside on the same disk. There is no guarantee of local load balancing.

Round Robin:
Instead of assigning pages to disks randomly, we can assign them in a systematic manner, keeping the number of pages on each disk balanced. This can be done quite easily by keeping track of the number of pages assigned to each disk. If two or more disks contain the same number of pages, we can break the tie, for example, by assigning the new page to the lowest-numbered disk in the set. In the absence of deletions, this scheme achieves a completely balanced data load, that is, each disk contains $\lceil \frac{n}{p} \rceil$ or $\lfloor \frac{n}{p} \rfloor$ pages. The main drawback is the same as for the B-tree based on random distribution, that is, no guarantee of local load balancing. In the worst case, a range query may be restricted to a single disk.

The first two approaches discussed (record distribution and large pages) have unnecessarily high resource consumption for one or more of the basic operations, which reduces throughput. The methods dis-

cussed in this section seem not to have this drawback. In principle, the B-trees presented in this section can be viewed as employing a *dynamic* hybrid range-partitioning scheme. However, they do not guarantee local load balancing, which may result in poor response time for range queries. To improve the response time, we would like a scheme where the pages of contiguous subsequences of pages are uniformly distributed over the disks. This is the topic of the next section.

7 B-trees with Local Balancing

In this section we propose a new algorithm for assigning pages to disks that performs local load balancing. The basic idea of the algorithm is as follows. Consider the pages to be ordered in the natural sequence, that is, in ascending order on the minimal key value of each page. When a new page is created, we avoid assigning it to a disk that contains a nearby page. Nearby here means within a window of size $p-1$ or $p-2$ pages centered on the new page. If there are more than one candidate disks for the new page, we select one with the lowest data load. The algorithm follows.

Algorithm FindDisk(page);

1. $CS := \{1, .., p\}$; left := page; right := page;
2. FOR i := 1 TO $\lceil \frac{p}{2} - 1 \rceil$ DO
   right := RightPage(right);
   left := LeftPage(left);
   $CS := CS \setminus \{$ Disk(left), Disk(right)$\}$;
   END;
3. RETURN ($\min\{i | load[i] = \min_{j \in CS} load[j]\}$);

END FindDisk;

The page for which we require a disk is used as the input parameter of the algorithm *FindDisk*. In case of a split *FindDisk* is called with the new page as the input parameter. This is done just before the new page is written to the disk. CS denotes the set of disks which can be used for storing the pages. The main part of the algorithm is in the second step. All disks on which a page close to the input page resides are rejected from the candidate set CS. The functions *LeftPage* or *RightPage* are assumed to return the address of the left page or the right page of the given page, respectively. If there is no neighboring page, the address of the given page is assumed to be returned. The function *Disk* returns the disk number of the input page. If the input page is not currently assigned to a disk, *Disk* returns zero or some value greater than p. In step three, ties are resolved. The array *load* is assumed to contain the number of pages assigned to each disk. The disks with the lowest load are first selected and, if there is more than one such disk, the one with the lowest disk number is chosen. The functions *LeftPage* and *RightPage* can be easily implemented by using left and right pointers on the lowest level of the B-tree index, see Figure 1. In the worst case, the algorithm requires an additional page to be read in. Typically, p would be much smaller than the capacity of a index page. Because the calls to *LeftPage* and *RightPage* are repeated $\lceil \frac{p}{2} - 1 \rceil$ times, there cannot be more than one additional read request.

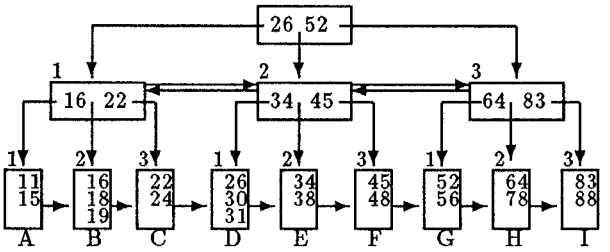

Figure 1: Example B-tree using three disks

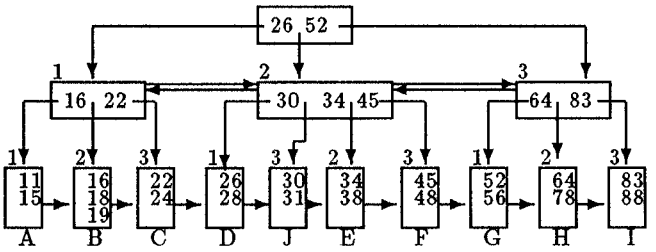

Figure 2: The example B-tree after split of page D

An example B-tree is shown in Figure 1. We assume that the capacity is three for both data and index pages. The number on top of the upper left corner of a page indicates on which disk the page is stored. (This information is stored as part of the pointer referring to the page.) The B-tree in Figure 1 corresponds to the ideal case where the load is both globally and locally balanced. Every disk contains the same number of pages and, for any range query, the number of pages retrieved from each disk differ by at most one.

However, this ideal situation cannot be preserved under further insertions without a global reorganization. Now let us insert a record with key 28, which causes a split of page D. The records $\{26, 28\}$ remain on the old page and the records $\{30, 31\}$ are moved to the new page J. Then the pointers are updated and *FindDisk* is called. After step one of FindDisk we have: $S = \{1, 2, 3\}$, $left$ = J and $right$ = J. The loop will be executed once only, yielding $left$ = D, $right$ = E and $S = \{3\}$. The new page is thus assigned to disk three. Figure 2 shows the resulting B-tree.

Theorem 7.1 *If pages have been assigned to disks using algorithm FindDisk and there have been no deletions, then no two pages of a contiguous subsequence of* $\lceil \frac{p}{2} \rceil$ *or fewer data pages are stored on the same disk.*

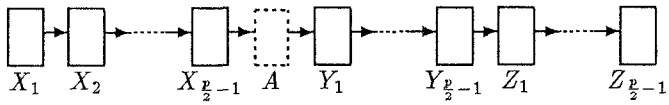

Figure 3: Merge of page A and $X_{p/2-1}$

Proof 7.2 *It is easy to verify that, if the B-tree consists of at most p data pages, the algorithm will assign each page to a different disk. In this case, the theorem is trivially true.*

Now assume that the file consist of more than p data pages and consider the assignment of a new page. The condition in the theorem can be violated only by the new page. However, because of step two in the algorithm, the new page will be assigned to a disk in such a way that none of the $\lceil \frac{p}{2} - 1 \rceil$ *pages to the left and none of the* $\lceil \frac{p}{2} - 1 \rceil$ *pages to the right of the new page are stored on the same disk. All contiguous subsequences of at most* $\lceil \frac{p}{2} \rceil$ *pages in which the new page participates are contained within the range of pages checked in step three. Consequently, the condition in the theorem must still hold after a page split. This proves the theorem.*

Recall that the best response time we can expect for a query answered by a parallel B-tree is $\Psi_{max}(q) = \lceil \frac{\Psi_{sum}(q)}{p} \rceil$. Our parallel B-tree is close to that, even in the worst case, which is expressed in the following corollary.

Corollary 7.3 *Let q be a range query. If pages have been assigned to disks using algorithm FindDisk and there have been no deletions, then the following inequality holds:*

$$\Psi_{max}(q) \leq \lceil \frac{\Psi_{sum}(q)}{\lceil \frac{p}{2} \rceil} \rceil$$

Our B-tree guarantees that, for range queries that retrieve no more than $\lceil \frac{p}{2} \rceil$ data pages, no disk will be accessed more than once. For larger range queries, we can guarantee that no disk will be accessed more than twice the optimal number, that is, the number of accesses required if the pages retrieved were completely evenly distributed. By considering a query that retrieves all pages in the tree, it follows that no disk will contain more than twice the expected number of pages. In other words, if there are N data pages, the maximal number of pages on any disk is $\lceil N/\lceil \frac{p}{2} \rceil \rceil$. These are all worst case results; we can expect average performance to be significantly better.

So far we have assumed that there are no deletions. The question is how to perform deletions so that the condition in theorem 7.1 remains true. A deletion may cause two pages to be merged. If so, one of them will be deleted and we can no longer guarantee that the condition holds. Hence, some of the pages near the deleted one may have to be moved to some other disks.

Figure 3 illustrates the merge of page A with page $X_{p/2-1}$. The neighboring pages to the left and right of A are denoted by X_i and Y_i, respectively. Consider a page Y_i. We can view the page as the owner of a "window" extending to the left and to the right of the page, covering a total of $p-1$ or $p-2$ pages. The deletion of page A has the effect of sliding the window one step to the left, causing one new page, X_i to become visible in the window. We cannot guarantee that all pages in this new window are stored on different disks. Hence, the condition in the theorem may be violated. It is not difficult to see that there is only one way this may occur: if pages X_i and Y_i happen to be stored on the same disk. If this is the case, we should move Y_i, not X_i, to another disk. To find out which disk, we simply use the algorithm FindDisk. In summary, after the merge, we check whether pages X_i and Y_i are stored on the same disk, $1 \leq i \leq \lceil \frac{p}{2} \rceil$. If this is true for some i_0, we use algorithm FindDisk to determine what disk to move page Y_{i_0} to. In the worst case, a merge may require $2 * \lceil \frac{p}{2} - 1 \rceil$ I/O operations. One additional disk access may be needed to determine on which disks the pages Z_i are stored, $1 \leq i \leq \lceil \frac{p}{2} \rceil$, and another write might be necessary to update the pointers in the parent page. However, the required I/O operations can be performed in parallel because the pages Y_i all reside on (and will be moved to) different disks.

So far, we have considered data pages only. As discussed earlier, it is realistic to assume that all index pages, except those at the lowest level of the index, are in main memory. It is obvious that the same algorithm can be used for distributing index pages. However, to avoid interference among the different levels of the tree, a separate *load* array should be used for each level in the algorithm *FindDisk*.

8 Preliminary Performance Comparison

In this section, we report the results of a preliminary performance comparison of different multi-disk B-tree structures. All results are based on simulation experiments. The objective of our first set of experiments was to find out how much is gained by the local load balancing discussed above (compared with random distribution and simple round robin). Thus, in these experiments mainly the response time of range queries is investigated. A second set of experiments focussed on throughput. We compared the performance of three different B-tree structures, each one representing a different approach (record distribution, large pages, page distribution).

8.1 Response Time of Page Distribution B-trees

The first set of experiments were performed in the following way. In each experiment, three different B-trees

	minimum	maximum
local balancing	141	142
round robin	141	142
random assignment	124	159

Table 2: Minimum and maximum number of data pages assigned to a disk ($p = 10, N = 1416$)

were created inserting the same records. The trees differed only in how data pages were assigned to disks: one used local load balancing, one used round robin and one used random assignment. The trees were created by inserting one record at a time. After the trees had been built, the number of pages assigned to each disk was output. Table 2 shows the minimum and maximum number of data pages assigned to a disk for a file containing 1416 data pages. Assuming pages of 4Kb, a B-tree with 1416 data pages corresponds to a data file of about 3.8Mb. As it should, round robin assignment yields a perfectly uniform data load. In this experiment, local load balancing also resulted in a perfectly balanced data load. Obviously, this is not always the case, but in all our experiments the data load was very close to uniform. The uneven data load distribution shown for random assignment is typical for that scheme.

More important than the data load is the response time of queries, in particular range queries. For a range query q the response time is proportional to $\Psi_{max}(q)$, the maximum number of accesses to an individual disk. To illustrate the relation to the best response time achievable by any page distribution scheme, we normalize our results using the following formula:

$$\Psi_{norm}(q) := \frac{\Psi_{max}(q)}{\lceil \Psi_{sum}(q)/p \rceil}$$

$\Psi_{norm}(q) = 1$ is the best we can expect. Thus $\Psi_{norm}(q)$ shows the speed up gained by parallelism. $\Psi_{norm}(q) = 2$ means that this organization needs double the time of the optimal organization. Expressed in a different way: under optimal conditions, this operation can be performed in the same amount of time using half the number of disks.

In order to have control over the number of pages retrieved, range queries were approximated as follows. For a given value of m, $m > 0$, and a key K, we traverse the index to find the page where K is stored. We then retrieve exactly $m - 1$ subsequent pages (with higher key values). We varied the query size, $m = 1, .., 100$, and number of disk, $p = 10, 15, 20, 40$. We considered B-trees with 707, 1416, 2143, 2827 and 7117 data pages, which is equivalent to a data file size in the range 1.9Mb to 19.5Mb. All experiments gave results similar to those plotted in Figure 4 where the file consisted of 2827 data pages and 20 disks were assumed. As expected, the response time of our local load balanced B-tree is optimal for small range queries. For large range queries,

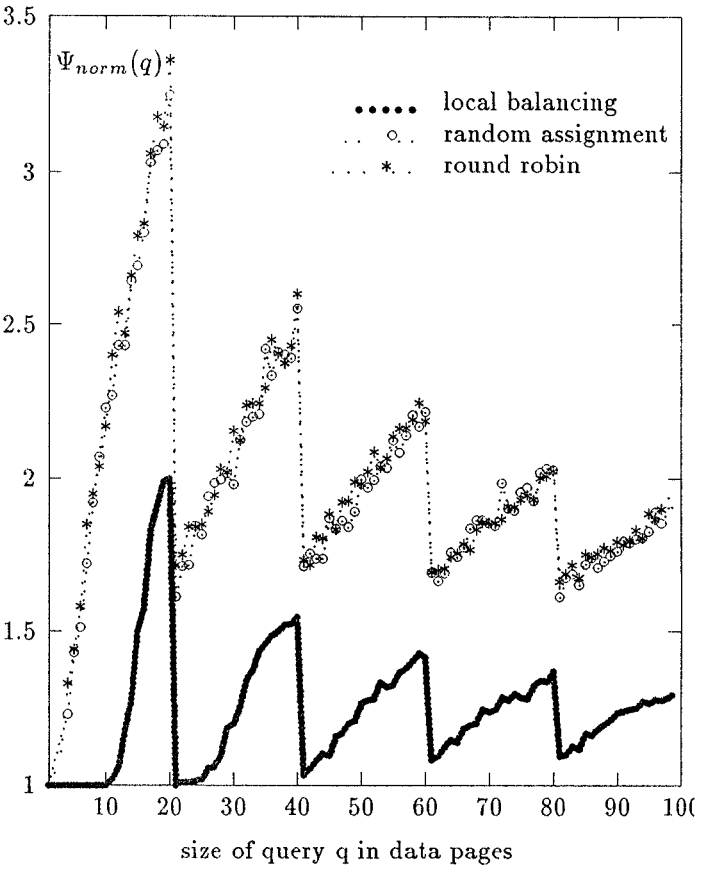

Figure 4: Normalized response time as a function of query size (p = 20, N = 2827)

the response time never more than twice the minimum time. The two other B-trees exhibit similar cyclic performance, but the response time is consistently higher. In summary, local load balancing appears to significantly reduce the response time for range queries.

8.2 Throughput

The next set of experiments were intended to compare the throughput of different B-tree organizations. The most promising schemes from the three classes were chosen: the synchronized large-page B-tree (SLB-tree), the B-tree with records distributed by hashing (HBB-tree), and the B-tree with page distribution and local load balancing (LOB-tree).

Table 3 attempts to summarize the characteristics of the three different types of B-trees. The first three rows indicate the number of disk accesses for different query types. The abbreviations *emq* and *rq* stand for exact match query and range query, respectively. The terms *large* and *small* range query have the meaning defined in section 2. For each type of query q, two performance measures are provided: $\Psi_{sum}(q)$ and $\Psi_{max}(q)$. $\Psi_{sum}(q)$

		LOB-tree	SLB-tree	HBB-tree
emq	Ψ_{sum}	1	p	1
	Ψ_{max}	1	1	1
small rq	Ψ_{sum}	$1-2$	$p-2p$	$p-2p$
	Ψ_{max}	$1-2$	$1-2$	$1-2$
large rq	Ψ_{sum}	$\frac{N}{2p*su}+2$	$\frac{N}{2p*su}+2p$	$\frac{N}{2p*su}+2p$
	Ψ_{max}	$\frac{F_1*N}{2p^2*su}$	$\frac{N}{2p^2*su}+2$	$\frac{F_2*N}{2p^2*su}+2$
avq. qtime		+	−	+

Table 3: Summary comparison of different parallel B-trees

is the total number of I/O requests. This measure indicates the load on the I/O system. A high request load reduces the query throughput. $\Psi_{max}(q)$ indicates the maximum number of accesses per disk. This measure is related to response time. On a lightly loaded system, the response time of a query is largely determined by the maximum number of accesses to any single disk.

For exact match queries, the three schemes behave similarly with the notable exception of the high resource consumption of the SLB-tree. For small range queries, the most notable difference is the much lower resource consumption of the LOB-tree compared with the other two schemes. The LOB-tree requires only one or two pages to be brought in while the other two schemes require one or two pages from each disk. For large queries, the total number of disk accesses is about the same for the three schemes (unless p is very large). The maximum number of accesses per disk is affected by how evenly the accesses are distributed over the disks. This effect is expressed by the constants F_1 and F_2 in the corresponding formulas. In case of the LOB-tree, we have shown that 2 is an upper limit for F_1. The average value of F_1 is expected to be in the range of 1.1 and 1.2. The value of F_2 is more difficult to estimate; all we can say with certainty is that it is greater than one.

On a highly loaded system, request queues will form in front of each disk. (In the case of synchronized disks, there is a single queue.) The queue waiting time may be a significant component in the response time of a query. One can expect the queue waiting time to be significantly higher for the SLB-tree because of the higher resource consumption of all operations. The LOB-tree and HBB-tree should have similar queue waiting times.

Table 3 attempts to summarize the factors affecting throughput for the three different schemes. To provide some more detailed results, we performed a series of experiments. We implemented the different types of B-trees but had to simulate the I/O system (because of the hardware requirements). The experiments were run

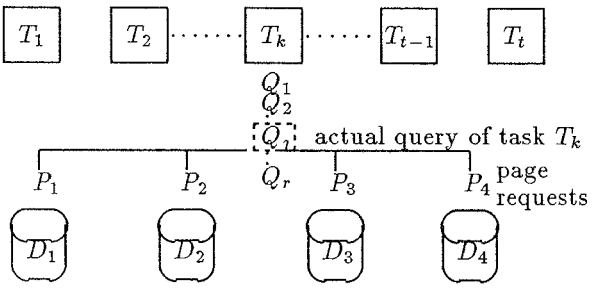

Figure 5: The model for query execution

on a Sequent Symmetry and organized as follows.

- The time for a disk request was calculated from a formula given in [PH 90] corresponding to IBM 3380D disks. This formula takes into account the fact that, in practice, the positioning time is typically much less the average seek time for the disk (due to locality of references).
- We simulated a multi-user environment, as illustrated in Figure 5. Each user was represented by a task. Each task had a certain number of queries to process and processed them sequentially. That is, at any given time, each task was processing exactly one query. A query requires a set of pages. All the addresses of the required data pages were first computed. Then the task sent requests in parallel to the different disks, at most one request at a time for each disk. Thus the total number of outstanding requests for a disk was restricted by the number of tasks.

We performed several experiments but due to space limitations we present only one set of results. The results are for a B-tree consisting of 3177 data pages, i.e. assuming 4Kb pages, the size of the file is about 8.6Mb, and a system with 10 disks. There were 20 query processing tasks running in parallel. This certainly represents a high request load on the system. We varied the number as well as the size of the queries in different experiments. The results are presented in Table 4.

The first column indicates the type of B-tree, the second column shows the query size (the fraction of records retrieved), and the third column lists the number of queries per task. The fourth column shows the total I/O time per disk (in seconds), that is, the total amount of time each disk was busy performing I/O. All operations were clearly I/O bound. The fifth column shows the total elapsed time (in seconds). The last column shows the average time (in seconds) that an I/O request waited in a disk queue.

The performance of the HBB-tree and the SLB-tree is very similar for all the queries considered in our experiments. As expected, the LOB-tree has the best performance for small queries. For queries retrieving 0.25%

	size	#q	IO-time	runtime	qtime
LOB-tree	10.00%	10	111.927	115.703	0.291
SLB-tree	10.00%	10	120.470	125.655	0.365
HBB-tree	10.00%	10	120.086	124.583	0.238
LOB-tree	1.00%	50	61.354	63.945	0.163
SLB-tree	1.00%	50	80.691	82.889	0.369
HBB-tree	1.00%	50	81.495	83.646	0.131
LOB-tree	0.25%	100	34.228	35.680	0.119
SLB-tree	0.25%	100	68.589	70.320	0.368
HBB-tree	0.25%	100	71.237	73.006	0.142
LOB-tree	0.01%	200	10.693	13 488	0.036
SLB-tree	0.01%	200	78.181	80.416	0.361
HBB-tree	0.01%	200	82.678	85.899	0.138

Table 4: Performance of three different multi-disk B-trees for range queries

of the records, the LOB-tree is about twice as fast as the other organizations. For 0.01% queries, it is about eight times faster. For large queries (10% or higher), all three methods seem to achieve approximately the same throughput.

9 Conclusion

In this paper, we outlined several types of multi-disk B-trees based on three different approaches. For each type, we considered how the performance of the standard B-tree operations (insert, delete, modify, exact match query, and range query) are affected. We then presented a new multi-disk B-tree, called the LOB-tree. The LOB-tree is based on distributing the pages of an ordinary B-tree over multiple disks. Pages are assigned to disks so as to locally balance the load on the available disks. The LOB-tree guarantees that any, sufficiently large, range query will be answered at least $\frac{p}{2}$ times faster than for an ordinary (single-disk) B-tree. Preliminary experimental results indicate that, on average, the performance gain is close to optimal. The LOB-tree attempts to reduce the total number of disk accesses required for an operation so as to achieve high throughput in a multi-user environment. An experimental performance comparison with the hash-based B-tree and the large page B-tree using synchronized disks, showed the LOB-tree had the highest throughput for range queries. Results for other operations are not yet available but there is no reason to believe that the LOB-tree would have lower throughput than other schemes for these operations.

References

[BM 72] R. Bayer, E. M. McCreight: 'Organization and maintenance of large ordered indices', Acta Informatica 1, 3, 173 -189, 1972

[Com 79] D. Comer: 'The ubiquitous B-tree', Computing Surveys, Vol. 11, No. 2, 121-137, 1979

[CABK 88] G. Copeland, W. Alexander, E. Boughter, T. Keller: 'Data placement in Bubba', Proc. ACM SIGMOD Conf. on Management of Data, 99-108, 1988

[DGS 90] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao and R. Rasmussen: 'The GAMMA Database Machine Project', Trans. on Knowledge and Data Engineering, Vol. 2, No. 1, 44-62, 1990

[Du 86] H. C. Du: 'Disk allocation methods for binary product files', BIT 26, 138-147, 1986

[FM 89] C. Faloutsos and D. Metaxas: 'Declustering using error correcting codes', Proc. Symp. on Principles of Database Systems, 253-258, 1989

[GD 90] S. Ghandeharizadeh and D. .J. Dewitt: 'Hybrid-range partitioning strategy for multiprocessor database machines', Proc. on VLDB, 481-492, 1990

[HL 90] K. A. Hua and C. Lee: 'An adaptive data placement scheme for parallel database computer systems', Proc. on VLDB, 493-506, 1990

[Kim 86] M. Y. Kim:'Synchronized disk interleaving', IEEE Trans. on Computers, 35(11), 978-988, 1986

[KP 88] M. H. Kim and S. Pramanik: 'Optimal file distribution for partial match retrieval', Proc. ACM SIGMOD Conf. on Management of Data, 173-182, 1988

[LKB 87] M. Livny, S. Khoshafian and H. Boral: 'Multi disk management algorithms', Proc. ACM SIGMETRICS, 69-77, 1987

[LL 86] W. Litwin and D. Lomet: 'The bounded disorder access method', Proc. Conf. on Data Engineering, 38-48, 1986

[PGK 88] D. Patterson, G. Gibson and R. Katz: 'A case for redundant array of inexpensive disks (RAID)', Proc. ACM SIGMOD Conf. on Management of Data, 109-116, 1988

[PH 90] D. Patterson, J. Hennessy: 'Computer architecture: a quantitative approach', Morgan Kaufmann, 1990

[PK 90] S. Pramanik and M. H. Kim: 'Parallel Processing of large node B-trees', Trans. on Computers, 39(9), 1208-1212, 1990

[SM 86] K. Salem and H. Garcia-Molina: 'Disk striping', IEEE Conf. on Data Engineering, 336-342, 1986

[Sal 88] B. Salzberg: 'File structures: an analytical approach', Prentice-Hall, 1988

[SK 90] B. Seeger, H. P. Kriegel: 'The Buddy-Tree: an efficient and robust access method for spatial data base systems', Proc. Conf. on VLDB, 590-601, 1990

[Sie 90] H. M. Sierra: 'An introduction to direct access storage devices', Academic Press, 1990

[WB 87] C. T. Wu, W. A. Burkhard: 'Associative searching in multiple storage units', ACM Trans. on Database Systems, 12, 1, 38-64, 1987