Low-Level I/O Optimization in Database Systems
A Case For Multi-Page Requests

Bernhard Seeger

Low-Level I/O Optimization in Database Systems
A Case For Multi-Page Requests

Bernhard Seeger

Abstract

Magnetic disks are the most important storage medium today and are expected to play that role for at least the next ten years. Database systems primarily use magnetic disk drives as their directly accessible persistent storage device. The performance of magnetic disk drives has been considerably improved over the last 25 years. However, the improvement rates are far behind those achieved for processors. This is the most important reason that the I/O to magnetic disks has been more and more the bottleneck of many database applications, in particular of non-standard applications.

This thesis starts with a survey on common low-level techniques to improve the I/O performance of computer systems. Thereafter, the current magnetic disk technology is discussed in great details. The rest of the thesis is then dedicated to studying the impact of multi-page requests on query performance.

A multi-page request retrieves several pages from disk without interfering of other requests. The elapsed time of a multi-page request is determined by the order in which the pages are read from disk. Under the assumption of different disk models, algorithms for implementing multi-page requests are discussed and their performance is analyzed analytically and experimentally. Moreover, the expected cost of a multi-page request is expressed by functions that can be easily computed. For our most accurate disk model, the cost function depends on several parameters describing the geometry of the disk, the number of required pages and the degree of clustering. The cost function, which is shown to be in good agreement with results obtained from experiments with a real disk, demonstrates that significant performance improvements can be achieved by using multi-page requests when the required pages are read according to a well-computed schedule.

In addition to the response time of an individual query, we also examine the impact of multi-page requests on the throughput. Results of an experimental performance comparison demonstrate that the throughput can be improved by several factors when multi-page requests are used in comparison to reading the required pages one at a time.

In order to demonstrate the impact on the performance of important data structures, index structures are modified in such a manner that they exploit multi-page requests for evaluating data-intensive queries. 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 range queries can be reduced by several factors if B^+-trees use multi-page requests.

Acknowledgements

I would like to thank my supervisor, Prof. Dr. Hans-Peter Kriegel, for his great guidance on my way from a master student at the University of Würzburg to a research and teaching assistant at the University of Munich.

My warmest thanks to my wife. Her constant support, encouragement and love made my life brighter and my work easier.

I would like to give my special thanks to Prof. Dr. Per-Åke Larson, the external reviewer of my thesis. During my stay at the University of Waterloo he gave me a great support in my work and eventually, he was the one who pointed out the problem on multi-page requests. I am thankful to Prof. Dr. Seegmüller, the second reviewer of the thesis, for his careful reading of the thesis.

Other people who deserve special mention are my friends Dr. Thomas Brinkhoff and Dr. Ralf Schneider. Our fruitful discussions did not only improve the thesis. Furthermore, I would like to thank my present colleagues of the database group at the University of Munich and to my former colleagues of the University of Würzburg, the University of Karlsruhe, the University of Bremen and the University of Waterloo. John Scourias, a student from the University of Waterloo, deserves special thanks for reading a first version of the thesis and improving the readability.

Low-Level I/O Optimization in Database Systems A Case For Multi-Page Requests

Bernhard Seeger

A previous version was successfully presented as a thesis to
the Fachbereich Mathematik of Ludwigs-Maximilian-Universität
München
in fulfillment for the degree of
Dr. rer. nat. habil.
in
Computer Science

München, July 1994

Contents

Chapter 1

Introduction

The short history of computer technology has been characterized by rapid innovation. In the last three decades, dramatic performance improvements have been achieved for both main memory and processors. However, the development of secondary and tertiary storage, the third major hardware component of computing technology, has not kept pace with the gains made in the other components.

Magnetic disks have been the dominant secondary storage device for almost thirty years. The primary role of magnetic disks in computer systems is to provide a non-volatile storage medium on which programs can keep data permanently even when they are not running. In comparison to main memory, an important advantage of magnetic disks is their reasonable cost: 1 MB of disk space is about 10 to 30 times cheaper than 1 MB of main memory [HP90]. Therefore, disks are also "misused" in many computer systems as inexpensive virtual main memory when real main memory is too small to keep all data resident.

The principle of magnetic disks is based on both magnetic recording and precision mechanics [Hoa85]. Because mechanical movements are required for reading and writing data on disk, the time for accessing data on a magnetic disk is fairly high. Today's disk can access data in 10 to 15 ms on the average. By comparison, an ordinary workstation can perform an access to data in main memory (which does not require any mechanical movements) in a mere 80 ns. Hence, the access times of magnetic disk and main memory differ by a factor of about 150,000. According to Amdahl's law[1], only a small fraction of performance improvements

[1] Amdahl's law states that the performance improvement to be gained from using some faster mode of

in processor speed could be passed to system software which relies on secondary storage. In particular, database systems suffer more and more under the performance gap between processing data in main memory and secondary storage.

A database system (DBS) considers a disk as a collection of buckets, called pages in the following, in which data records are stored. When a request for a data record is issued, the corresponding page is read from or written to the magnetic disk. The time for accessing a page on magnetic disk consists of two basic components: positioning time and transfer time. Positioning time is the time to move the read/write head of the disk to the desired position. Transfer time refers to the time required for transferring a page into main memory. For the last twenty years, transfer time has been improving roughly as fast as processor speed, about 50% annually, whereas the positioning time has improved at the modest rate of about 7% annually. The reason for the vast improvements in transfer time is that the rate of progress in magnetic density has continued undiminished for the last thirty years. Consequently, the I/O time of reading or writing a page for today's magnetic disks is clearly dominated by the positioning time. Moreover, it is expected that current improvement rates in positioning and transfer time will continue at least until the end of the decade [Hoa85, Woo90]. Because many applications of DBSs require several pages from different positions of magnetic disks during a short time period, positioning time is much more crucial to efficiency in a DBS than transfer time.

In order to maintain a balance of performance between main memory and secondary storage, several approaches and algorithms have emerged. Recent examples includes disk arrays [Ouc78, PGK88, BG88], non-volatile storage [CKKS89], disk scheduling [Den67, Fra69, CKR72, SCO90], prefetching [Smi76, CKV93] and multi-page requests [Wei89]. The first two approaches are based on a novel hardware architecture, whereas the remaining ones rely on the design of efficient algorithms. The common goal of the algorithmic approaches is to reduce positioning time on magnetic disks. So far, the design, implementation and analysis of algorithms for multi-page requests has attracted little attention although it provides the fundamental ideas for disk scheduling and prefetching.

In this thesis, the problem of designing and analyzing efficient algorithms for multi-page

---

program execution is limited by the fraction of the time the faster mode can be used [HP90]

requests, also called set-oriented I/O [Wei89] and bulk I/O [BP88], is addressed. A multi-page request is an I/O request that accesses several pages on disk where the order does not count. As a rule, the pages are located close to each other on disk, e.g. on a cylinder, but contiguity is not required. A multi-page request is assumed to be performed without interfering with other requests. The I/O time of a query that requires access to a large number of pages can be substantially reduced when multi-page requests are used instead of processing one page at a time. Moreover, the throughput of the I/O system can also be improved when the required pages of a multi-page request are almost contiguous on disk. In particular, multi-page requests reduce the movement of the disk arm, which is the most expensive portion of access time on magnetic disks. Obviously, more buffer space is required for performing a multi-page request than for the strategy of one page at a time. Although prototype implementations have demonstrated the advantages of multi-page requests, particularly for accessing large objects, the general technique can rarely be found in current DBSs. In contrast to [Wei89], our work is primarily dedicated to selection queries, although the technique can generally be used whenever multiple pages are read from secondary storage. In the following, we will give several examples in which the application of multi-page requests almost always proves advantageous.

Reading a large number of pages occurs, for example, when a secondary index is used for the evaluation of a selection query. A selection query searchs for all objects in a given set which satisfy a given predicate. The simplest way of performing this operation is to scan the index and for each qualifying entry in the index retrieve the required page from the file. This has the drawback that the same page may be accessed more than once. An improvement on this scheme is the following approach: create a list of the pages to be retrieved, eliminate duplicates from this list, and then retrieve the required pages one at a time. If two or more required pages happen to be located close to each other, for example, on the same disk track, total retrieval time may be reduced if all of them are read with a single multi-page read request rather than issuing multiple requests, each reading a single page. It is interesting to note that multi-page requests are already used to a limited extent in commercial systems [BP88] to improve this type of operation.

When a cluster index can be exploited for performing selection queries, e.g. range queries,

multi-page requests can be used in a way similar to that introduced for secondary indices. Let us assume that a cluster index is implemented as a B^+-tree and that a range query should be performed on the B^+-tree. A range query is specified by two search keys L and U with $L < U$. All records are searched whose keys are in the interval $[L, U]$. A B^+-tree is a balanced multi-way tree that consists of internal pages and leaf pages. An internal page consists of entries which refer to lower levels of the tree. Data records are stored in the leaf pages which are linked together in key order. Range queries in B^+-trees are performed as follows: First, search key L is used for traversing the B^+-tree from the root to the corresponding leaf page. The leaves are then accessed sequentially following the pointer to the next leaf that contains records with keys greater than L. Sequential processing of leaf pages requires that one page is read from disk at a time. Hence, multi-page read requests cannot be exploited for improving the performance of the classical range query algorithm for B^+-trees. However, multi-page requests could be used if the range query algorithm were modified as follows. First, we compute a list of all addresses of leaf pages which are required for answering the range query. Second, several leaf pages are read using a multi-page request when pages are located close to each other on disk. Although the new range query algorithm may require a few page accesses more than the classical one, the performance of range queries can be improved in general.

In a different setting, multi-page requests are used for purging modified pages from a write-cache to magnetic disk. A write-cache is a non-volatile disk cache that is protected against power failures. From a user's point of view, a write request is completed when the modified page is in the write-cache. Thus, write-caches can effectively reduce the response time of I/O requests. However, this requires that the buffer is not completely filled up with modified pages. In order to provide free space in the buffer, some of the modified pages have to be written back (i.e. purged) to disk. Almost all approaches suggest purging the pages which have been least recently modified in a single multi-page request. Recently, similar ideas have been proposed to overcome the problem of expensive write operations in disk arrays.

Most disk systems now feature some form of multi-page request. A frequent operation in a DBS is to transfer a file completely from disk into main memory. Because most files (relations) are well clustered on disk, this results in reading pages contiguously stored on disk. After satisfying a few page requests, the disk anticipates the sequential access pattern of the

query and starts transferring several contiguous pages from disk into a buffer although the query still requires and processes one page at a time. This technique is commonly referred to as prefetching [Smi76].

Multi-page requests have already been demonstrated to be beneficial for reading (large) structured objects from magnetic disk into main memory [Wei89, KGM91]. The basic problem is that objects consist of a fairly large number of references to other objects or object fragments. These subobjects may be jointly referenced by a fairly large number of objects. Without introducing undesirable redundancy, objects cannot be well clustered on magnetic disk. Then, a pointer stored in a page, say P, on secondary storage, refers to a page which is in general not identical to P. In the worst case, every pointer of an object refers to a different page, where the corresponding subobjects can be found. Moreover, the same situation can now occur for each of the subobjects. In order to provide fast access to an object, some DBSs, e.g. DASDBS [PSS+87], separate the structural information from the actual data. The structural information can be kept in a page or, if the need arises, in a tree-based directory. When an object is required from magnetic disk, the structural information is read first. In a second step, the data is read using one or more multi-page requests.

Common to all the examples presented above is the following problem: given a list of pages retrieved from (written to) magnetic disk, what is the fastest way of retrieving (writing) them? Furthermore, we are interested in the corresponding cost for the optimal read schedule. In particular, the query optimizer of a DBS can make use of such cost formulas for determining efficient execution plans for query processing. The efficiency of multi-page requests depends on the distance between the corresponding pages. For a very large number of pages, it is not always better to read all pages in a single multi-page request. In particular, this would substantially hurt the response time of other requests. Instead, only pages which are close to each other (e.g. on a common track) are transferred in a single multi-page request.

Examples of retrieving multiple items at once can be found everywhere in daily life. Consider the manner in which people purchase items in a supermarket. In general, the time required to go to the store is greater than the time spent in the store. The strategy of going to the store for every item separately is very inefficient with respect to the overall time someone has to spend per week for purchasing items. In contrast to that, people usually behave as

follows: First, they come up with a list of all items required in the near future. Once they are in the supermarket, they use the shortest way in the store for obtaining all items on the list. Occasionly, we are not pleased about long queues in front of the cashier and about other people who are buying a lot of items and have lined up in front of us. But usually, when we can avoid hot spots, queues are rather short.

The rest of the thesis is structured as follows. In the next chapter, we present an overview of query processing in DBSs. Special consideration is given to low-level methods for improving the I/O performance of the DBSs. In chapter 3, we give a detailed discussion on magnetic disk drive technology. In chapter 4, we present algorithms for implementing multi-page requests and their analysis under the assumption of the so-called linear disk model. We come up with a cost function that varies in the number of required pages and in the size of the buffer. In chapter 5, the same problem is studied under the assumption of the idealized disk model. This model takes into account the geometry of a disk as they are cylinders, tracks and pages. For a scan-based algorithm, the cost of multi-page requests is analyzed. One of the deficiencies of the idealized disk model is that head switch time is not taken into account. In chapter 6, the problem is studied for a disk model that also considers head switch time and other important aspects of today's disks. The main contribution of the thesis can be found in section 6.3. A simple cost function is derived for multi-page requests under a disk model that considers almost all properties of today's magnetic disks. In chapter 7, the results obtained from the cost function of section 6.3 are shown to be close to the results obtained from experiments with a real disk. Moreover, a disk simulator demonstrates the accuracy of the cost function for various parameter settings. In chapter 8, we present a new variant on B^+-trees, called CB^+-tree, which take advantage of multi-page requests for performing range queries. In addition, the CB^+-tree offers global clustering, i.e. pages which contain answers for the same range query are stored close to each other on disk. Although the cost of building up the CB^+-tree is only slightly higher than for the B^+-tree, range queries are performed much more efficiently. In chapter 9, we summarize the work and discuss areas of future research.

Chapter 2

I/O Optimization Techniques

In this chapter we review methods for improving I/O performance. First, we recall that it is of vital importance to expend more effort on I/O optimization. For that, we have to take a closer look at the storage hierarchy of computer systems. The classical view of computer storage consists of main memory and secondary storage, e.g. magnetic disks. Main memory is comprised of DRAMs (dynamic random access memory), which allow access to data in about 80 ns, i.e about 150,000 times faster than accessing data on magnetic disk. This performance gap has been the subject of much research in the areas of computer architecture, operating systems and database systems. In the computer architecture field, fast disk caches have been introduced for avoiding repeated access to frequently referenced data. Today, there exist non-volatile disk caches which are protected against power failure by battery backup. Research related to operating systems has dealt with methods for scheduling I/O requests waiting in front of the disk and with policies for cache management, such as replacement policies. Research in the database area has been focused on designing efficient index structures and clustering policies.

In the following, we first briefly discuss how queries are processed in a DBS. In particular, the need for query optimization is stressed as one the most important issues in today's DBSs. Moreover, the optimization techniques presented in the following sections will be shown to have an important impact on efficiency. Thereafter, we discuss several low-level optimization techniques which belong more to the functionality of an operating system, but are also of great importance to a DBS as well. In section 2, we briefly review disk scheduling policies.

Several approaches for clustering are discussed in section 3. In section 4, a survey is given on techniques for buffer organization. In addition to volatile buffers, our discussion also includes how non-volatile buffers can be exploited, as well. In section 5, a discussion on disk arrays follows. In section 6, the basic idea of multi-page requests is discussed. Section 7 concludes the chapter.

2.1 Query Processing in Database Systems

In the first section, a brief survey is given on query processing in DBSs. Most of the approaches discussed in this section have been developed for relational DBSs. However, they are also almost always applicable to the query processing facility of any DBS and any data model. More details on query processing in DBSs can be found in various books, see [OV91] for example, and articles [SAC+79, JK84, Gra93].

A DBS is a highly concurrent system. It allows that several independent users with different requirements on the system perform queries on the same database at one time. Therefore, the DBS pursues two performance goals with respect to efficient query processing. First, the *response time* of an individual query should be minimized. The response time is defined as the elapsed time from the initiation to the completion of the query. Second, the *throughput* of the system should be maximized. The throughput is only well defined for a given workload (i.e. mix of queries or transactions) on the system. Throughput is measured as the number queries (or transactions) that can be performed by the DBS in one second. In many cases, a DBS optimized with respect to reducing response times also offers high throughput and vice versa. For some cases, however, these performance goals are in conflict with each other.

There are many reasons why a user should not be involved in the process of optimizing queries. First of all, a user might be able to optimize a query with respect to response time, but it is almost impossible to consider throughput as well. Second, as the designers of one of the first relational systems stressed [ABC+76], one of the most important advantages of relational database systems in comparison to other ones is that they take away the heavy burden of manual design and coding from the user. Consequently, relational systems offer a programming language that allows non-experts to express queries in an easy fashion. For

Figure 2.1: Architecture of a DBMS

today's DBSs, SQL [AC75] is the de-facto standard for a query language. SQL is a non-procedural language that was originally developed for System R [ABC+76], one of the first research prototypes of a relational DBS. A query expressed in SQL only describes the response set without mentioning or specifying implementation details. The DBS has to transform an SQL query into an efficient execution sequence, also called the *query execution plan*.

Efficiency of a query can be expressed with respect to the total cost or the response time. The *total cost* is defined as the cost of all components. Cost is generally measured in terms of time units. A cost formula for the total cost can be specified as follows [ML86]:

$$C_{CPU} * (\#instructions) + C_{I/O} * (\#I/Os)$$

where C_{CPU} and $C_{I/O}$ are the cost for a CPU instruction and disk I/O, respectively. Cost formulas for the response time are rarely used for query optimization. Notable exceptions are the query optimizers of distributed and parallel database system [OV91].

In order to discuss the transformation process of queries in more detail, let us first introduce a simplified architecture of a relational DBS illustrated in Figure 2.1. We follow

an approach that has been presented in [OV91]. The architecture is basically adopted from the five-level architecture of Härder [Här87]. The architecture consists of four levels: the programming level (SQL), the logical level, the physical level, and the level of the operating system (functionality). The *logical level* refers to the specific data model of the DBS. In general, an algebra, called *logical algebra*, is associated with that level. For a relational DBS, for example, the logical algebra consists of data structures such as relations, (logical) records and views, etc. The operators include the ones of the relational algebra such as selection, projection and join operators. The *physical level* consists of a collection of data structures and algorithms for implementing the functionality of the logical level. Typical data structures at the physical level are access methods and (physical) records. This level is completely independent from the logical level and it can therefore be used for implementing any type of DBS. Below the physical level, the operating system might be used to bridge the gap between the physical level and the hardware. In particular, a file system and a buffer are provided on that level. However, file management and buffer organization of an operating system are not very appealing to a DBS [Sto81, CHMWS87]. Therefore, these functionalities are commonly reimplemented in a DBS.

An SQL query is processed as follows. First, the query is parsed into an internal form that uses the operators and the data structures of the logical algebra. In general, a (logical) operator tree is used for representing the query. The internal nodes of the tree refer to the algebraic operators, whereas the leaves refer to the logical data structures. In addition, macros and views are expanded into the query. The first step of "optimization" referred to as *algebraic optimization* is based on some heuristics. It is performed without any knowledge of the actual cost of the operators. It is basically restricted to eliminating common and useless expressions in the query. In addition, some equivalence rules are applied to the operator tree. One of the best known rules is that a selection operator should be executed prior to a join and projection operator. As a consequence, selection operators frequently read the data from a permanent file, whereas the other operators can generally read their input from main memory or from a temporary file. Thus, in order to improve I/O performance, special attention has to be given to selection operators. In the next step, the tree is transformed into a *physical operator tree* that only contains the data structures and the operators of the physical level. Many physical operator trees are correct transformations of the same logical operator tree.

The goal of query optimization is to select the most efficient physical operator tree, or what might be more important, to avoid non-efficient ones. This optimization process can be very complex and might require substantial search and cost estimation.

Since it is much too expensive to compute the exact cost of a physical operator tree, query optimization relies on cost estimations. In System R [SAC+79], the I/O cost is simply measured by the number of pages fetched from secondary storage, and CPU-cost is expressed in the number of answers returned from a query. In addition, a weighting factor is used to adjust I/O and CPU cost. The ratio of I/O cost to CPU cost has been constantly increasing for the last twenty years so that many queries are I/O bound in today's systems or they are expected to be I/O bound in the near future. This holds particularly for selection queries. In order to compute the cost of a query, the number of page fetches and the size of the response set of a query must be estimated. For that, the DBS maintains statistics about the physical data sets such as the cardinality of the sets, the number of pages in the set, the availability of indices, etc. Under the assumption of uniformly distributed records, the cost for each operator of the physical algebra is estimated so that the (expected) cost depends solely on the cardinality of the input. The cost of a query, which is represented as a physical operator tree, can then be computed bottom up from the leaves to the root of the tree.

There are several drawbacks related to that approach of query optimization. First, the assumption of uniformly distributed records is rarely fulfilled in practice. This assumption is pessimistic, i.e. the cost of a query will generally be overestimated [Chr84]. Second, for complex queries (i.e. the operator tree is rather tall) errors are propagated from the leaves to the root in the operator tree [IC91]. Third, the I/O model is very simplistic. It is assumed that the cost for a page access is constant for any page that is fetched from secondary storage. The cost for a page access is generally assumed to be identical to the average disk access time [SAC+79]. This rough estimation leads to an overestimation of the costs. Marckert and Lohman [ML86] noticed that deficiency in their experimental validation of System R*. Consequently, the cost for a disk access was decreased to the value observed as the average in their experiments. However, this value might be different in other experiments.

There is great potential for improving query performance at all levels of the DBS architecture. However, the physical algebra and the level of the operating system (functionality)

are related the most to efficient query processing. During the last decade, most of the research focused on improving the physical level. For example, access methods, algorithms for join processing and sorting are discussed in great detail in the database literature. A survey of these methods can be found in [Gra93]. There are only a few (but important) articles, see [CDRS86, BP88] for example, which are concerned with the design of the file system. Today's file systems provide almost the same interface as for the last twenty years. Since then, computer system technology has changed greatly and database systems are used in new application areas such as CAD, geography and environmental science. These applications are posing increasing demands with respect to efficient query processing. A frequently asked question is therefore whether we have forgotten to improve the file system [Wil94]. Of particular interest is a file system that efficiently supports selection queries (operators), since these queries are expected to access data stored in the physical data base (see our discussion above).

Our previous discussion stressed the importance of cost estimation. Without having accurate cost estimations, the query optimizer can hardly find an efficient physical operator tree. Therefore, whenever a new method is introduced into the physical level, the query optimizer also requires its cost estimations. This is even more crucial when the functionality of the operating system (OS) level changes. Then the cost estimations in the physical level have to be re-examined for those operators that exploit the new functionality in the OS level. Otherwise, the query optimizer would come up with a wrong decision. In the remainder of this chapter, we review several low-level techniques which might be candidates for improving query processing of a DBS.

2.2 Disk Scheduling

As early as twenty years ago, the basic policies for scheduling I/O requests were discussed in several pioneering papers [Den67, Fra69, TP72, CKR72]. A *scheduling policy* decides which of the I/O requests from the queue should receive service next. The primary goals of a scheduling policy are not restricted merely to maximizing throughput, but also to minimizing the average response time of a request, and to reducing the variance of response time. In order to meet these requirements, policies optimize schedules with respect to the arrival time

and the disk position of the requests.

The simplest scheduling policy is *first-come-first-serve*. This policy does not take advantage of positional relationships between I/O requests in the present queue. More sophisticated policies are *shortest-seek-time-first (SSTF)* and *SCAN*. For SSTF, the I/O request is serviced which has the minimum seek cost with respect to the current position of the disk arm. This policy improves the overall throughput (I/O rate), but it can lead to discrimination of individual requests. For the SCAN policy, the disk arm operates like an elevator as it moves up or down in one direction. It only changes direction at the innermost and outermost cylinder of the disk. Thus, every cylinder is reached during a scan of the cylinder. Consequently, SCAN provides lower response time variance than SSTF, but a slightly increased response time for the requests. Several variants of the SCAN and SSTF algorithms have been proposed (see [Dei90] for a survey). Note that these methods were proposed almost thirty years ago. At that time, access time was clearly dominated by seek time. This might be one of the reasons that policies try to improve seek time, but not rotational delay. Moreover, rotational scheduling is often considered to be unnecessary for disks [Smi81], since it is rare to have more than one I/O request outstanding for the same cylinder. Scheduling with respect to rotational optimization has also been considered, but the approaches are primarily designed for fixed-head disks such as drums [SF73, Ful74]. More recently, a disk scheduling policy was examined in [SCO90] and in [JW91] that selects the request with minimum positioning time.

In [CJL89, AG92], scheduling of I/O requests was considered with respect to a completely different objective. The basic idea is that priority is assigned to the I/O request such that scheduling does not select requests according to seek time, but according to priority. This situation occurs in real-time databases where transactions have to fulfill deadlines.

Scheduling of I/O requests have almost always been discussed under the assumption that there is a significant number of requests in the queue. For example, in [TP72] a queue length in the range of 100 to 140 has been considered. In general, this assumption is not justified. In [GD87], results have been reported from experiments on a real system. These results have shown that the utilization of a disk is quite low. The number of requests waiting in the queue while another I/O request was being serviced was on the average about 0.2. King [Kin90] stated that systems with average queue length longer than 1 are rarely found in practice.

Instead, he pointed out that successive accesses to disks are usually done by one process and that this process sends I/O requests to the disk one at a time. His conclusion was that "time would be better spent looking at what to do when there is no queue". More recently, an extensive experimental comparison of various scheduling policies has been presented in [WGP94] where the influence of disk caches, in particular, is taken into account.

2.3 Clustering

The most important factor to efficient query processing is that the records of the response set of a query are stored close to each other on disk. The term *clustering* refers to that vague idea. In a DBS, clustering can be achieved at several levels of abstraction. The different approaches can be differentiated according to the underlying objects considered for clustering such as records, pages and files (or relations).

Many applications for database systems, e.g. organization of spatial objects, require efficient processing of range queries and other proximity queries. The efficiency of proximity queries is achieved by using access methods which assign objects to a page according to their spatial location. A page required for answering a proximity query, then, contains in general several answers fulfilling the same query. Consequently, the number of page requests is substantially lower when objects are clustered in comparison to the number of requests when objects are stored at a random position in the file. For spatial database systems in particular, such access methods are indispensable to efficient query processing. Moreover, a database system can aim at storing certain pages of a relation contiguously (or on the same cylinder), if these pages are expected to contain common answers of queries. For example, a spatial access method would improve performance of proximity queries, when pages with spatially close objects are kept contiguously on disk [HSW88]. Similar ideas were examined for a special B^+-tree, called VSAM [KL74], almost twenty years ago.

In contrast to a DBS, operating systems are not aware of the special semantics of a file. Therefore, structural information is only used for clustering pages. Operating systems cluster pages of a file according to their logical position in the file. Pages logically adjacent in the file are stored physically close to each other on disk. Of particular interest is the question of where a new page of a file should be stored on disk. For example, in the "new" file system of

UNIX [MJLF84], new pages of a file are preferably placed at rotationally optimal positions in the cylinder where space for the file has been allocated last. If this is not possible, a cylinder is chosen in the same cylinder group. However, the space in a cylinder group still might not be sufficient for large files. A cylinder is then randomly selected from those cylinders that offer a large fraction of free pages. Although this strategy provides clustering of pages, files may become severely fragmented when records are frequently added and deleted. In that case, a global reorganization of the disk can establish contiguity of all the files again. However, such a reorganization is expensive and it should be avoided whenever possible.

Eventually, DBSs and operating systems support clustering of relations and files, respectively. A physical disk is usually partitioned into a few contiguous parts, called *partitions* or *extents*. Two relations (files) are clustered together in an extent, if queries often require access to both of them. For example, a join operator requires access to several relations which should be preferably kept together in a common extent.

The problem of clustering files becomes more complex if the I/O system is assumed to be located on a large disk array [CABK88, WSZ91] or distributed over several sites [DF82].

2.4 Buffer and Cache Organization

In order to reduce the performance gap between fast main memory and slow secondary storage, buffers and caches have been proposed to keep a large fraction of secondary storage main-memory resident. Buffers are getting cost-effective because the cost of DRAMs is decreasing by a factor of 30% annually [HP90]. Cost reduction is even higher for main memory than for magnetic disk. Moreover, it is remarkable that the data density on a DRAM has been improved by a factor of 1000 over the last twenty years. This is a factor of ten higher than the improvements achieved for density on magnetic disks. However, main memories which are large enough to keep all data resident in main memory are still too expensive for most applications. The reason is that along with the development of main memory the amount of data is also dramatically increasing such that more and more secondary storage is required in computer systems [Gel89].

In general, the primary goal of a buffer is to reduce the number of accesses to disks.

The buffer area is subdivided into *(buffer) frames*, and each frame can contain a page from secondary storage. A *buffer fault* occurs when access to the disk is needed because a requested page is not in the buffer. If an empty frame exists, the requested page is then read into one of the available buffer frames. Otherwise, when all frames are occupied, one of them has to be made available using a *replacement policy*. This policy is only allowed to consider the replacement of those pages which are not currently used by an another process. When the replacement policy selects a frame with a so-called "dirty" page, i.e. it has been modified but not written back to disk, the page first has to be written back to disk before another page can use its frame.

The most popular replacement policy is LRU (least recently used) [EH84], which replaces the page that has not been referenced for the longest time. The efficiency of LRU is based on the fact that a page has a high probability of being accessed again within a short period of time. For some of the relational queries, the reference pattern can be predicted rather well such that other replacement policies are more efficient than LRU [SS86, CD85, TG84]. For example, the LRU policy is not appropriate for sequential scans of a relation where a page can be immediately replaced after it is processed. In [JCL90], a more general approach has been presented that makes use of priorities given to the pages in the buffer. If a page has to be replaced, the one with the lowest priority that has been not accessed in the recent past is chosen. Another interesting generalization of LRU, called *LRU-k*, was proposed in [OOW93]. The basic idea is to take into account not only the last reference to a page, but the last k references, $k > 1$. This leads to a better prediction of the future reference pattern in comparison with the original LRU policy.

As important as replacement policies are the strategies for writing back modified pages from the buffer to disk. There are two different approaches on how to deal with write requests. The *force policy*, also called the write-through policy, a write request results in the corresponding pages to be written to disk immediately. The *no-force policy*, also called the write-back policy, copies the modified page into a buffer frame and defers the I/O operation until later. For the no-force policy, the response time of a write request does not include anymore the time required for writing the page to disk. However, the process which has issued the write request does not have the guarantee anymore that the modified page is in

safe memory. Thus, transaction processing under the no-force policy is more complicated than under the force policy [GR93].

So far, we have assumed that in case of a system failure, e.g. if the power supply is shut off, the contents of a buffer are lost. This might not be true anymore for some of the current and most of the future computer systems. Storage built up from DRAMs can be made non-volatile using battery backup. In particular, the low power consumption of todays DRAMs has made this development possible. Non-volatile storage (NVS) can be differentiated into three classes (see also [Rah92]):

- *Solid state disks* (SSDs) are the most common approach for NVS. An SSD is not a mechanical device, but simply built up of DRAMs. It is still accessed like an ordinary magnetic disk by using the same interface. The advantage of SSD is that positioning time can be ignored so that access time only refers to the sum of controller and transfer time (in the range of $1-3\ ms$). The architecture is illustrated on the left of Figure 2.2.

- Some disk systems (e.g. IBM 3990 [CKB89]) provide a non-volatile cache in the controller. Such a cache is also called a *write-cache*. indexwrite-cache As discussed for SSDs, positioning time is almost completely avoided. The architecture is presented in the middle of Figure 2.2.

- *Expanded storage* is used in IBM 3090 mainframe computers as a $4\ KB$ page addressable extension of main memory [CKB89]. Expanded storage can only be used through a special interface for reading pages from expanded storage and for writing pages into expanded storage. The interface protects expanded storage against errors, e.g. address violation, which occur frequently in main memory. As illustrated on the right of Figure 2.2, expanded storage can be made non-volatile by simply using battery backup [CKKS89].

The advantage of the last approach is that access to slow controllers and slow buses is completely avoided. Access time of a page in expanded storage is about $75\ \mu s$ for the IBM 3090 [CKB89], i.e. a factor of 20 faster than access to a solid state disk.

Non-volatile buffers can be used to improve response time because a write request succeeds when the page is copied into the frame of the buffer rather than when it is written on magnetic

Figure 2.2: Design of several non-volatile storage architectures

disk. Similarly to volatile buffers, the question arises of when a dirty page should be written back to disk. There are two conflicting goals that are attempted to be fulfilled. On the one hand, the response time of a write request should not include the time for writing back a page to disk. This situation would occur when the buffer is completely filled up with dirty pages. Then, the next request (for a page that is not in the buffer) would have to wait until one of the dirty pages is written back to disk. On the other hand, writing back a dirty page to disk can completely be avoided, if another write request refers to the same page. Obviously, a large number of dirty pages in the buffer increases the probability that a write request hits one of them. For a buffer of reasonable size, the force policy can achieve the first goal, but it does not make use of locality of references. Thus, a policy similar to no-force seems to be more advantageous. Another advantage of no-force is that several write operations can be merged in a batch, as suggested in [CKB89]. The amortized time for performing a write operation can then be substantially reduced.

Another interesting approach for using non-volatile buffers has been presented in [SO90]. The basic idea is to *piggy-back* write requests, which are kept in a separate write-only cache, onto read requests, at little or no cost. For example, since a random read results in a rotational delay of half a track on average, the disk arm passes over a few pages without interacting with them. However, if one of these pages is in the cache waiting to be written to disk, the corresponding write request can be performed during the rotational delay of the read request. Thus, the write request can be satisfied without any additional cost.

2.5 Disk Arrays

Over the last five years, many researchers have focused their attention on designing parallel disk systems, also called *disk arrays*. The basic idea of disk arrays is as follows: in order to improve the I/O rate and the transfer rate, data is distributed over a large number of uniform disks (i.e. the disks should be of the same type). Ideally, several I/O requests can be serviced in parallel and a large I/O request can read (write) data in parallel from (to) several disks. In the best case, a disk array can improve the transfer rate and I/O rate (defined as the number of I/O requests per second) linearly in the number of its disks. Note that the access time of an I/O request is only reduced if it transfers a sufficiently large amount of data. For most

database applications, the transfer rate of a single disk is almost always sufficiently high, but the I/O rate might be by far too low. In order to provide high I/O rates, disk arrays should obey the following suggestions, see [SL91]:

- The arms of disks in an array should move independently of each other, so that the disk array can satisfy several I/O requests at the same time.
- Data should be striped in large units over the disk such that small I/O requests are restricted to a single disk. However, large I/O requests should exploit parallelism.

Synchronization of disk arms and small striping units might, however, be beneficial in other applications, e.g. supercomputing, where huge amounts of data have to be transferred into main memory.

One of the most serious problems of disk arrays is that reliability declines linearly in the number of disks [PGK88]. In order to improve reliability, redundancy of data has been introduced for disk arrays. In [PGK88], the term RAID (redundant and inexpensive disks) has been coined to refer to reliable disk arrays. The authors distinguished between five levels for RAID called RAID-1,...,RAID-5. For database applications, the most interesting RAID approaches are RAID-1 and RAID-5. RAID-1 is more commonly referred to as mirrored disks and shadowed disks [BG88].

2.5.1 Mirrored Disks

The basic idea of mirrored disks is to store a physical copy of a disk on a second disk. Thus, there is a storage overhead of a factor of 2, i.e. double the number of disks are required for storing the data in comparison to storing data without redundancy. If a page is read, one of the two disks is chosen to retrieve the page. In general, a read request is serviced from the disk whose arm is closest to the desired page. When a page is written, the same write request is issued on both of the disks. Both disk arms of the mirrored disks then have to be positioned to the desired track where the page has to be written. Therefore, if the next request after reading a page is a write request, the corresponding cost is higher compared to using a single disk. Otherwise, i.e. if the next request is a read operation, the cost for reading is lower. In [BG88], access time for mirrored disks is analyzed under the unrealistic assumption that

each disk arm of a pair of mirrored disks is randomly positioned (independent of the other disk arm) on the surface of the disk. Actually, the cost formulas presented in [BG88] for the access time of a read request and a write request represents only a lower and an upper bound of the expected cost, respectively. Disk arrays built up from mirrored disks are very reliable. A failure of the disk array occurs if both disks of a pair of mirrored disks will be defective. Following the formula in [PGK88], the *mean time to failure*, MTTF for short, of a disk array of 100 mirrored disks is about 75 years under the pessimistic assumptions that first, the MTTF of a single disk is 3 years and second, the mean time to repair (MTTR) the disk array (i.e the expected time for substituting the defective disk) is 5 hours. In case of a failure of a disk in the array, the load of a pair of mirrored disks is put on the surviving disk. The increased load during the time period required to replace the defective disk by a new disk could be a problem. The replacement of a defective disk also includes copying the data on the surviving disk to the new disk. This further increases the load on the surviving disk. Other approaches based on mirrored disks [CK89, HD90] overcome these problems using other policies on how the copy of the original disk is distributed in the disk array. Reliability of these approaches is still sufficiently high, but not as high as for mirrored disks.

2.5.2 Reliability Using Parity Bits

Besides mirrored disks, RAID-4 [Ouc78, SGM86] and RAID-5 [PGK88] are the most appealing approaches of disk arrays for database applications. For simplicity, let us first restrict our presentation to RAID-4. Assuming an array of N disks, the basic idea of RAID-4 is to distribute the data in units of blocks (or larger) over the first $N-1$ disks. The N-th disk of the array, also called the parity disk, is used for keeping redundant data. Hence, there is only a storage overhead of a factor of $\frac{N}{N-1}$. That is an obvious advantage in comparison to mirrored disks. Let $P[i,j]$ be the page on the i-th position of the j-th disk, page $P[i,N]$ on the parity disk is computed by XORing pages $P[i,1],\ldots,P[i,N-1]$ bitwise. In case of a failure of disk j_0, the page $P[i,j_0]$ can be reconstructed by XORing pages $P[i,1],\ldots,P[i,j_0-1],P[i,j_0+1],\ldots,P[i,N]$.

A read request refers to a single disk while a write operation affects the parity disk and the disk where the (data) page is stored. Writing page $P[i,j]$ results in the following steps:

1. Read the parity page P[i,N]. If the old copy of $P[i,j]$ is not available anymore, read the "old" page $P[i,j]$ from disk into $Pold[i,j]$.
2. Compute $P[i,N]$ as ($Pold[i,j]$ XOR $P[i,j]$) XOR $P[i,N]$.
3. Write pages $P[i,N]$ and $P[i,j]$ to disk.

A write operation requires (at most) four disk accesses, two reads and two writes, where both writes and both reads can be done in parallel. Hence, a write operation in a RAID-4 is more expensive than a write operation for a single disk. Because each write results in two accesses to the parity disk, the parity disk might become the bottleneck for RAID-4. In order to overcome this problem, RAID-5 [PGK88] does not use a parity disk anymore, but distributes the parity blocks evenly over all the disks in the array.

For $N > 2$, RAID-4 and RAID-5 are not as reliable as mirrored disks because a failure of two disks in the array at the same time results in data loss. For a disk array of 100 disks, an MTTF of 3 years per disk, and an MTTR of five hours, the MTTF of the disk array is about one and a half years. In order to obtain higher reliability, the disk array can be partitioned into several small groups, where each of them is independently organized as a RAID-5.

In case of a disk failure, the additional load for RAID-4 or RAID-5 is higher than the one for mirrored disks. The reason is that read requests require access to each of the $N-1$ surviving disks. For the same reason, rebuilding a RAID-4 or RAID-5 disk array is more expensive in comparison to mirrored disks.

2.5.3 Exploiting Write-Caches in Disk Arrays

In the previous subsections, we have presented the basic ideas of the most common disk arrays used for database applications. We have not considered several important details. One of them is the design of disk array controllers. Disk array controllers are a rather complex piece of hardware. It is probably one of the reasons that disk arrays are not as inexpensive as once promised [PGK88]. A major difference to ordinary disk controllers is that a disk array controller contains large caches for buffering pages. Since one of the most serious drawbacks of disk arrays is the high cost for writing data, exploiting write-caches has recently been considered for RAID-5 [MC93] and mirrored disks [PBD93].

First, let us discuss the approach to mirrored disks presented in [PBD93]. The idea for improving the performance of mirrored disks is to use a buffer (if possible non-volatile) efficiently. Consider a pair of mirrored disks labeled A and B. Write operations are not written to disk, but collected in the buffer until a certain number of operations has been accumulated. Then, the write operations (of the dirty pages in the buffer) are applied as a batch to the disks. First, disk A is used for writing the buffer while disk B services read requests for those pages that are not in the buffer. Next, the buffer is written out to disk B and read requests are performed on disk A. Thus, each disk alternates between time periods of write-only and read-only activity. The time period in which no write operations are performed on disks can be used to improve the response time of read requests as originally introduced in [BG88]. Moreover, in such time periods disk arms are never synchronized such that the average seek time indeed refers to the cost formulas given in [BG88].

In [MC93, SGH90] similar ideas have been presented for RAID-5. In comparison to the approach for mirrored disks, these approaches designed for RAID-5 seem to be more complex.

2.6 Multi-Page Requests

In the previous sections, we already presented approaches where several page requests are merged into a multi-page request. A *multi-page request* is performed like a batch so that other requests have to wait in front of the disk while the multi-page request is serviced. Almost all of the approaches presented in the previous sections are only concerned about improving write requests. In the following, our emphasis is put on how to improve the performance of read-only queries by using a buffer so that multiple pages can be transferred in a single read request.

One of the most popular techniques for improving I/O performance of read requests is *prefetching* [Smi76], also called prepaging [Tri79] and anticipatory paging [Dei90] in the context of virtual memory systems. Prefetching transfers one or more pages into a buffer without satisfying an explicit read request for one of these pages. In order to be efficient, prefetching only considers those pages which are likely to be required in the near future. In general, prefetching is used when a page fault occurs. Then, the required page and multiple physically adjacent pages are read into a buffer. Reading physically adjacent pages avoids the

cost of positioning the disk arm. Thus, pages can be read for the expense of a page transfer. A scan of a complete file is an example that shows the benefits of prefetching. When page i causes a page fault, pages $i+1, \ldots, i+k$ should also be brought into the buffer because it is likely that one of the next requests will ask for those pages. For a scan of a file, the efficiency of prefetching depends on how well the file is clustered on disk. If only a fraction of the pages in a file is required, the performance of prefetching declines because many pages transferred into the buffer are not used during their stay in the buffer. However, more sophisticated prefetching techniques may overcome such problems [CKV93, PZ91]. Prefetching is already implemented as an optimization technique in many commercial DBSs such as IBM's DB2 [CLSW84, TG84] and Tandem's NonStop SQL [BP88].

The term prefetching is mostly restricted to request patterns which are not known at present. However, there are many situations where the sequence of pages required for performing a query is entirely or partially known shortly after the beginning of query processing. Instead of reading one page at a time, the required pages can be read as a batch in a single multi-page request. The pages in a batch can be read in arbitrary order, but we are interested in the order that will minimize the response time for retrieving the required pages. Note that scheduling policies also try to find an "optimal" order for processing I/O requests, but optimality is not only restricted to improving the cumulative response time of all requests. The approach of multi-page requests is also termed set-oriented I/O [Wei89] and bulk I/O [BP88]. Multi-page requests have been demonstrated to be beneficial for reading large objects from disk [CHMWS87, Wei89, KGM91]. However, multi-page requests can also be considered for improving data-intensive selection queries. For example, NonStop SQL [BP88] offers the possibility of reading physically contiguous blocks in a single request when a range query is performed. In [SLM93], under the assumption of a simple disk model, the benefits of multi-page requests are shown for selection queries on a sequential file, where the queries are evaluated by using a secondary index.

The idea of multi-page requests is increasingly used for current disk systems. Because of the availability of non-volatile disk caches, write operations can be assumed to be completed when the modified page is in the cache. The physical write operation (to disk) can be deferred to a later point in time. The time for performing a physical write can then be reduced by

combining multiple write requests of adjacent pages into a multi-page request [SO90]. This is particularly beneficial for disk arrays where write requests are more expensive than on ordinary disks [MC93, PBD93, SGH90].

2.7 Conclusion

In this chapter, a general introduction was given to query processing and query optimization in a DBS. Due to the high abstraction level of DBSs, query optimization is of great importance for improving the performance (w.r.t. high throughput and short response times). Performance depends on both CPU-time and I/O-time required for answering the query. In the current state of hardware development, I/O-time is increasingly the dominant cost component. The most common approach for improving I/O performance is to design new data structures (e.g. index structures) and algorithms (e.g. for join processing) and to make these available in the physical level of a DBS. However, our discussion was primarily related to the lowest level of the DBS, where the buffer manager and the file system is implemented.

The lower levels of the DBS also offer great potential for improving I/O performance of a query. However, this potential has only been exploited to a limited extent so far. One of the most appealing approaches is the technique of multi-page requests. A special form of multi-page request, called prefetching, is already implemented in commercial products [CLSW84]. However, prefetching is only applicable in a few situations, whereas multi-page requests can generally be used whenever multiple pages have to be read from magnetic disk. In order to incorporate multi-page requests into a DBS, the algorithms of the physical level have to be modified so that multi-page requests are used instead of reading pages one at a time. Moreover, since multi-page requests greatly affect performance, new cost functions have to be introduced so that the query optimizer can take multi-page requests into account for selecting a good query execution plan. In general, a complex query starts processing with selection operators before more expensive operators (e.g. join and projection) are executed. Thus, I/O performance greatly depends on how efficient selection queries (operators) are performed, whereas other operators read the input data directly from the buffer in main memory or (if main memory is too small) from a temporary file stored contiguously on magnetic disk.

Previous cost functions [Wat76, Yao77], that are applicable for selection queries when pa-

ges are read one at a time, are no longer valid when multi-page requests are used. Otherwise, the probability would be very low that the query optimizer will find an optimal execution plan. One of the most serious objection to these cost functions and also to more sophisticated ones [SG76, Kol78] is their independence of the underlying disk architecture. Due to the reduced complexity of the underlying disk models, the resulting cost functions are simple, but unfortunately results are not very accurate. As observed in [ML86], cost functions generally overestimate the actual cost.

In the following chapters, we present more accurate cost functions that can be used for estimating the I/O cost of selection queries which exploit multi-page requests. In particular, these cost functions take into account the disk geometry more than other known cost functions. Before we get into cost functions, however, a detailed discussion is required on the current state of disk technology.

Chapter 3

Magnetic Disk Systems

Despite numerous predictions to the contrary, magnetic disks have dominated secondary storage for the last thirty-five years. In comparison to other storage media, the advantages of magnetic disks are founded on low cost for a reasonably fast and reasonably large non-volatile storage medium.

In the first section, a review is given of the most important properties of present disk technology. Our discussion on magnetic disks gives more details than usually found in database and operating system literature, but it is still restricted to those properties relevant to our approaches. An excellent survey can also be found in [RW93a, RW94].

In the second section of this chapter, several disk models will be presented. In a DBS, disk models serve as a basis for designing efficient query execution plans and for estimating the cost of queries. The efficiency of the execution plan for query processing increases with model accuracy. However, query processing costs are difficult to estimate for a complex disk model.

3.1 Disk Technology

A magnetic *disk drive* is a collection of *platters* rotating on a spindle at a constant speed, about 3600 to 7200 revolutions per minute (*rpm*). Instead of disk drive, the short term *disk* is used in the following. Both surfaces of a platter are coated by a very thin metal film whose thickness is less than 1 *micron*. The platter diameter of today's disks is between 1.8 *inch* and

Figure 3.1: Disk architecture

5.25 *inch*. Each of the surfaces of a platter is divided into concentric circles known as *tracks*. There are typically 500 to 2000 tracks per surface. Each track is in turn divided into *sectors* where data can be stored. A sector is considered to be the smallest unit that can be read from and written to disk. In front of each sector, there is a gap that contains the address and the status of the sector, data for error correction, and some additional information explained later. A movable arm containing a *read/write head* is attached to each of the surfaces. The arms of a disk drive are connected and move synchronously so that the corresponding read/write heads are (almost) on the same track for each of the surfaces. However, only one of the heads can be active, i.e. reading and writing data, at one time. The term *cylinder* is used to refer to all the tracks under the read/write heads at a given time. For today's disks, the arms are implemented similarly to an ordinary arm of a record player. The arms are positioned by a voice coil motor that linearly moves the arms in one draw to the desired track. The read/write heads float on a cushion of air over the surfaces. The distance between platters and heads is only about 0.2 *micron*.

One of the reasons for the success of magnetic disks is the improvement in recording density. For magnetic disks, there are two distinct measures of density. One refers to the density of bits on a track, called the *linear density*, given in bits per millimeter (*bpm*), whereas the other, the *track density*, refers to the number of tracks per millimeter (*tpm*). The *area density* is simply the product of both. The area density of magnetic recording has been increasing constantly (doubling every two and a half years) for thirty years [Hoa85]. This

development allows a constant increase in the capacity of disks although the diameter of the platters is decreasing. The IBM 3390, introduced as the top disk model of IBM in 1989, offers a linear density of 1100 *bpm* and a track density of 98 *tpm*. In comparison, the predecessor disk, the IBM 3380 introduced in 1981, has a linear density of 600 *bpm* and a track density of 32 *tpm*. A more recently introduced disk, the Quantum Pro Drive, offers essentially higher area densities than the IBM 3390, see Table 3.1. For the near future, even higher density improvements have been announced, based on the assumption that *giant magnetoresistive* head technology [Sci93] will replace the current one in a few years.

Disk technology does not depend solely on high record densities, but also on mechanical precision [Hoa85]. This is demonstrated by the following discussion on positioning read/write heads. The positioning accuracy of a head can be controlled using only a dedicated *servo surface*. For such a disk, the heads on each of the surfaces are aligned at the same track. The advantage of this approach is that switching between heads does not cause any mechanical delay and hence almost no time delay. However, in order to achieve a higher track density, most of the present disks additionally prerecord the servo information in the gaps between sectors. Note that track densities are so great that even thermal differences between platters make it impossible to write and read data accurately on all surfaces using only the information from the servo surface. Consequently, the heads are no longer aligned to be exactly on the same track. Whenever a read/write head is activated (e.g. switched from one head to another), the arm first reads the servo information from the surface and then, if the need arises, tunes the head position. This operation is also called a *head switch*. The *head switch time* refers to the time required for switching from one read/write head to another. The head switch time of present disks is in the range of 0.5 *ms* and 3 *ms*. For modern disks, the head switch time of a read request is less than that of a write request. The reason is that a sector can be read from a disk although the arm is not exactly above the track. If some data of the sector is lost it can frequently be rebuilt from the error-correction information. On the other hand, writing a sector requires that the disk arm is precisely positioned on the track.

In order to read from or write to a sector, the arm first has to move to the proper track. This operation is called a *seek* and the time required to move the arm to the desired track is called *seek time*. For long seeks, the disk arm accelerates up to some maximum velocity, then

cruises at constant velocity, then decelerates back to zero, and eventually settles onto the desired track. For short seeks the arm accelerates up to the halfway point, then decelerates and settles. This behavior can be well approximated by a two-part function where one part refers to a linear and the other part to a square function. For example, the seek time of Tandem's XL80 disk is approximated to within 5% by the following formula [GHW90] where M refers to the number of cylinders:

$$seek\_time(dist) = \begin{cases} 2.5 + 0.32\sqrt{dist} & \text{if } dist > M/5 \\ 7 + \frac{dist - M/5}{100} & \text{otherwise} \end{cases} \qquad dist \in \{1, \ldots, M\} \tag{3.1}$$

Cost functions for estimating seek time are very similar to formula 3.1 and can be found in many publications, see for example [HP90]. The *average seek time* is defined as the cumulative time required for all possible seeks divided by the number of all possible seeks. Under the assumption that seek time increases linearly with the number of tracks, the time required for traveling over a third of the tracks refers to the average seek time. Therefore, this time ($seek\_time(M/3)$) is often used as an approximation to the average seek time.

After arriving at the right cylinder, the arm waits until the platter is rotated into the position where the desired sector starts. This time is called the *rotational delay*. The *average rotational delay* is given as half a revolution of the disk. Disks that rotate at 5400 rpm and 3600 rpm have an average rotational delay of 5.6 ms and 8.3 ms, respectively. The sum of seek time and rotational delay is also called *positioning time*.

Finally, the sector is transferred into main memory. The required time for this is known as *transfer time*. The transfer time for a sector can be simply computed as the size of a sector divided by the maximum throughput of a disk. The maximum throughput of today's magnetic disks is in the range of 2 to 5 MB per second. Note that throughput of disks is usually lower than the bandwidth of the bus that connects disk and main memory. In general, an I/O request is not restricted to a sector, but to a number of sectors contiguously stored on disk. For example, a database system transfers data in larger buckets. Such a bucket is called a *page* in the following. For a typical size of a page, e.g. 8 KB, the transfer time is between 1.5 and 4 ms for today's disks. It is interesting to note that the transfer time of a page is not fixed for a disk. The reasons are twofold. First, the record density of a track depends on its distance from the center. This feature is discused below in more detail. Second, some pages cross track boundaries and therefore, their transfer includes a head switch. The transfer time

	Fujitsu M2652	IBM 3390	Quantum ProDrive
year introduced	1993	1989	1992
disk diameter [inch]	5.25	10.8	3.5
area density [bits/mm^2]		107800	209000
capacity [GByte]	2.055	11.3	1.05
minimal seek time [ms]	2	2	3
average seek time [ms]	11	9.5	10.5
avg. rotational delay [ms]	5.56	7.1	6.7
I/O rate [seeks/s]	53.88		52.87
number of platters	12		6
capacity per track (net) [Byte]	45056		23040-49152
capacity per track (gross) [Byte]	52864		23940-51072
transfer rate (async) [MB/s]	3.0	4.2	2.3 - 4.67

Table 3.1: Specifications for some magnetic disks

for those can be double or more the transfer time of an "ordinary" page.

The performance measures of some disks are reported in Table 3.1. The *I/O rate* refers to the "expected" number of page requests that can be satisfied in a second. The I/O rate can be simply computed by using the average access time. The *average access time* of an I/O request is equal to the sum of average seek time, average rotational delay and transfer time. For example, the Fujitsu M2652 has an average access time of 18.56 ms for a page of 8 KB. Thus, the I/O rate is about 53.88 page requests per second. For transaction processing in particular, the I/O rate of a disk is more important than throughput. The minimal seek time refers to the time for moving the disk arm to an adjacent cylinder. The minimal seek time is also frequently quoted as the *track-to-track seek time*. The gross capacity of a track refers to the space occupied by sectors and gaps, whereas the net capacity refers only to the amount of space required by sectors. One of the most important properties of magnetic disks is high life expectation. The *mean time to failure* (MTTF) is the expected life span of a disk given in hours. In general, MTTF is more than three years. Some of the disk manufacturers give a guarantee of five years for their magnetic disk drives.

So far, only the cost that directly occurs on a disk has been taken into consideration. However, disks are connected to main memory through a disk controller and therefore other cost components may occur when an I/O request is satisfied. Obviously, there is some time

overhead at the disk controller. For example, disk controllers have to select channels for transferring both sectors and commands between several I/O requests and multiple disks. In general, however, the time spent on the controller is only a small portion of the time spent on the disk (usually less than 1 *ms* per request).

When a disk controller is connected to several disks, delays can occur when two disks are ready for transferring data at the same time. This situation can obviously be avoided if a disk is the only one connected to the controller while the disk satisfies the corresponding I/O request. As a result, I/O requests on other disks have to wait for the disk controller to become available although the disks are idle. The disk controller itself is also idle most of the time because it is waiting for the I/O request to transfer data. In order to reduce the idle time of a disk controller, a different strategy is pursued. Since the transfer time is only a small portion of access time, the controller disconnects from the disk while the arm moves to the desired position so that other disks can transfer data into main memory. This is called *rotational positioning sensing (RPS)*. If the controller is not available when the disk attempts a reconnection, a so-called *RPS miss* occurs. The disk then has to wait through another rotation before the next attempt can take place. As shown in [HGPG92] and as introduced already for some disks [HP90], a buffer on the disk might be a solution for avoiding RPS misses. The following situation might then occur for a controller connected to more than two disks: one disk can be seeking and others loading their buffer while another is transferring data from its buffer. In addition, a buffer on the disk can reduce the number of disk accesses and thus can have a substantial impact on the average access time [RW94]. Disks with large buffers (1 MB) are rather common today. Another approach that almost eliminates RPS misses is based on the use of alternate pathing [CKB89].

While a disk satisfies an I/O operation, other page requests have to wait in a queue in front of the disk. The time a request spends waiting for the disk to become available is called the *queuing delay*. If the queue contains more than one I/O request, the requests are scheduled according to a certain policy. The problem of efficient scheduling policies has been extensively discussed in the literature, see [Dei90] for a survey. Currently, there is some debate about the actual length of request queues in front of the disk. In [Kin90] it is stated that the average length of the queue is rarely longer than one request, whereas in [RW93b]

Figure 3.2: Layout of sectors (vertical alignment on the left and track skewing on the right)

queue length of 50 and more requests are shown to be rather common in real applications.

Important for reading data from a disk is how addresses are assigned to sectors. The primary goal of address assignment is to organize the sectors in a linear sequence according to their physical position. After having read a given sector its successor sector should then be read with very little or no time delay. An address of a sector is composed of its position on the track, and the addresses of its surface and its cylinder. The position on a track is defined relative to an *track index* kept at the beginning of a track. First, let us discuss the approach where the indices are vertically aligned for each track in a cylinder. Then, when an I/O operation requires the last and the first sector of two consecutive tracks, a head switch has to be performed after transferring the first sector. Since head switch time is so high that the first sector on the next cylinder cannot be read immediately, an additional disk rotation is necessary. This undesirable situation can be avoided, when the index of the next track is staggered by some sectors so that a sequential read over multiple "adjacent" tracks does not result in missing a full rotation. This technique is also called *track skewing* and can be found in almost all disks. In Figure 3.2 both approaches are illustrated for a disk that consists of 4 surfaces, each of them containing 8 sectors. The gray-colored pattern shows the position of the index in the track. In our example, the index of a track is located one sector after the index of the predecessor track. In a similar way as illustrated for tracks, the sectors of adjacent cylinders can be glued together (*cylinder skewing*). Then, sequential reading (writing) of sectors does not cause a delay of a full rotation when the disk arm crosses

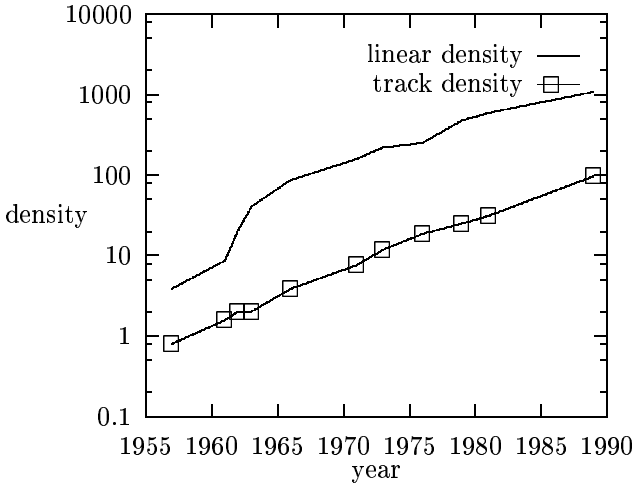

Figure 3.3: The development of linear and track density for IBM disks [HBP+81]

track and cylinder boundaries.

Several properties of disks make address assignment more complex than it seems to be at first glance. So far, it has been assumed that data can be kept in each of the sectors. However, this is generally not true. Even for a new disk, there are some "bad" sectors which cannot keep any data. These bad sectors are immediately replaced by so-called *spare sectors*. A disk contains some spare sectors which are only used for the replacement of bad sectors. In general, spare sectors are stored on every track either before or after the index. Moreover, there are disks with spare cylinders which are used for the replacement of a complete cylinder.

Another property of today's disks is that the number of sectors on a track depends on its perimeter. Most of the disks partition the surface of a platter into a few contiguous zones of cylinders. In each zone, the tracks in the cylinders consist of the same number of sectors, but the number of tracks is lower in those zones which are closer to the center of the disk. This technique is called *zoned bit recording*.

The development of magnetic disk technology has shown almost the same behavior for several years. As already mentioned, the success of magnetic recording, in particular for magnetic disks, is based on the improvements in area density. In Figure 3.3, the historical

development of linear and track density is illustrated for some IBM disks. In 1957, the IBM 350 was the first production movable-head disk offering a linear density of 3.93 *bpm* and a track density of 0.78 *tpm*. In comparison, the IBM 3390, currently one of the most powerful disks of IBM, has a linear density of about 1100 *bpm* and a track density of 98 *tpm* [Woo90]. Linear density has been improved at a slightly higher rate than track density. For the near future, magnetic recording density is expected to improve even faster than during the last decade [Woo90]. In particular, transfer rate will be improved for future disks while positioning time remains almost the same as for current disks. Although rotational speed has been improved by a factor 1.5 to 2 over the last few years, it is unlikely that improvements will continue. A further improvement of rotational speed would cause severe problems related to energy consumption, frictional heat, mechanics and magnetic recording [Dei90]. In Figure 3.4, the historical development of average access time is plotted for some IBM disks. The average access time for a 4 KB page is illustrated with respect to its components such as average seek time, average rotational delay and transfer time. The graph shows that the ratio of transfer time to access time has been substantially reduced so that transfer time has not much influence anymore on access time for today's disks. In contrast to transfer time, rotational delay did not much influence access time twenty years ago, but it can be expected to become the dominant factor of access time in the near future. Moreover, because of locality in disk references, the actual average seek time observed in practice is expected to be only 25% to 33% of the advertised number [CKB89, HP90]. If so, already for current disks, the major portion of access time is not seek time anymore, but rotational delay. Overall, the primary goals for optimizing I/O performance of disks should be to reduce seek time and rotational delay. To preserve locality of references is one optimization technique that might help achieve these goals. Other optimization techniques include the ones presented in Chapter 2.

3.2 Disk Models

In the previous section, the most important properties of current disk technology have been introduced. Obviously, a disk is a rather complex piece of hardware that is difficult to model accurately and in full generality. However, disk models are very important for designing efficient algorithms (e.g. request scheduling in front of the disk) and for estimating the cost

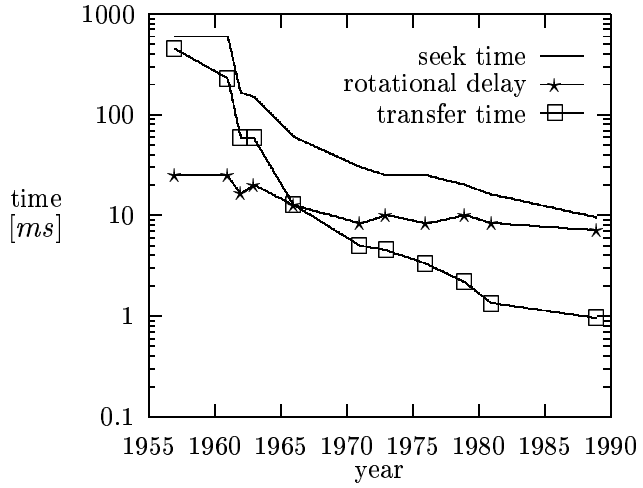

Figure 3.4: The development of access time for IBM disks [HBP+81]

of I/O requests, in particular for multi-page requests.

There are only a few papers, see [RW94] for example, that are concerned about modeling of disks. Most of the studies related to DBSs assume a very simplistic disk model: the access time is assumed to be constant for any page that is read or written. In general, the average access time is used for estimating the cost of a page request. Let A be the average access time. If N pages are required from disk, the cost estimation of the corresponding multi-page request would be $N * A$, which is identical to the cost for reading the pages in N single-page requests in arbitrary order. This approach is implemented in well-known DBSs such as System R and System R* [ML86]. The advantage of this model is its simplicity. For estimating the I/O cost of a query it is sufficient to compute the number of required pages. For example, when a range query is performed on a file using a secondary index, the Waters-Yao formula [Wat76, Yao77] provides an estimation for the number of page requests and hence, for the I/O cost.

In the following, three disk models are presented. In order to make these models as simple as possible without sacrificing too much accuracy, we make the following simplifying assumptions:

1. The time overhead of the controller can be neglected.
2. RPS can always be avoided.
3. Bad sectors do not occur.
4. There is no buffer on the disk.

The first three assumptions are mild abstractions of reality. Assumption 4 does not hold for most modern disks. Moreover, Ruemmler and Wilkes [RW94] have shown that a buffer on the disk has a serious impact on the average access time (since pages are frequently found in the buffer). However, a buffer has only a limited impact on the performance of data-intensive queries which require a page only once. In particular, this can be observed for selection queries, the type of query that is primarily considered throughout the thesis.

The Linear Model

The first step in the direction of a better disk model is to differentiate between transfer time and positioning time. However, each of them is still assumed to be fixed for an arbitrary page on disk. Furthermore, a neighborhood relation of pages is modeled as follows. The pages of a disk are linearly ordered such that every page, except the last, has a successor page. When a page has been read, the successor page could be read without causing any positioning cost.

This linear model is still very simplistic, but it already takes into account that the time for an I/O request depends on the request size and that there is some neighborhood relation between pages on disk. The model is particularly used for analyzing the cost for accessing large objects [Bil92, BK94].

The Idealized Disk Model

In order to come up with a better disk model the next step is to take into account the fact that access time consists of three components: seek time, rotational delay and transfer time. The transfer time is assumed to be constant, whereas the other components depend on the position of the page previously read from disk. Furthermore, a disk is not viewed anymore as a linear sequence of pages. Instead, every page is assumed to have several successor pages

Figure 3.5: A cylinder under the assumption of the idealized disk model

that can be read without any delay (seek or rotational delay). The model makes the following three assumptions:

- Head switch time is zero.
- The indices of the tracks are vertically aligned in a cylinder.
- A track consists of a fixed number of pages.

Now, a cylinder can be viewed as a two-dimensional array $C_{i,j}$ of pages, $0 \leq i < PT$, $0 \leq j < TC$. The parameter PT refers to the number of pages in a track and the parameter TC denotes the number of tracks per cylinder. The model would have represented the architecture of disks almost perfectly ten years ago, but it does not at present. In particular, the assumption of disregarding head switch time is not valid for a modern disk. Therefore, we refer to this model as the *idealized disk model (IDM)*.

An example of the layout of a cylinder ($TC = 5, PT = 4$) is illustrated in Figure 3.5. Consider that page $C_{2,3}$ has been read from the cylinder. The address of the page is 14. Then, a head switch can be performed to any other track such that one of the pages $C_{3,j}$, $1 \leq j \leq 5$ can be read at the expense of a page transfer.

The Head-Switch-Time Disk Model

The head-switch-time model generalizes the idealized disk model with respect to several points. First of all, head switch time is taken into account. Second, a page may cross a track boundary. Finally, track skewing and spare sectors are also considered in the model. Note that a cylinder cannot simply be pictured as a two-dimensional array of pages. In particular, the mapping of pages to sectors is more complicated and will be discussed in greater detail in Chapter 6 and Chapter 7.

Chapter 4

Query Performance under the Linear Model

In this chapter, we address the problem of reading a set of disk pages under the assumptions of the linear model. The linear model is a first approach for a model that is close to the actual disk architecture. In order to improve response time of a query, we assume that the technique of multi-page requests is exploited for query processing.

The chapter is organized as follows. In the first section, we define the problem more precisely and introduce our cost model. In section two, it is shown that finding an optimal read schedule is equivalent to finding the shortest path in a certain graph. In section three, a very simple algorithm is presented for reading a set of pages. Experiments show that its performance is close to optimal. In section four, we analyze the expected cost of the read schedules produced by this algorithm. We begin with two special cases and derive simple closed formulas for the expected cost. These two cases provide upper and lower bounds on the expected cost. The general case is then analyzed and a recurrence relation is derived which can be used to numerically compute the expected cost. Section six extends the model to vector reads (scatter-reads). Section seven summarizes the results and concludes the chapter.

F	target file
N	number of pages in the file
c	capacity of a page (in records)
Q	target set (pages to retrieve)
b	number of target pages
α	$= \frac{b}{N}$
m	maximum gap size
p	number of buffer pages
P	ratio of positioning time to transfer time

Table 4.1: List of symbols

4.1 Problem Statement

Consider a file F whose pages are in a contiguous sequence of pages numbered $1, \ldots, N$. Pages are of fixed size and each page can store a maximum of c records (page capacity). A query selects some subset of the records stored in the file (the response set of the query). A page containing at least one record in the response set is called a *target page* and the set of all target pages is called the *target set*. A page that does not belong to the file and that does not contain any record in the response set (and thus is not in the target set) is called an *empty page*. To compute the result of the query, every target page must be read. We assume that the complete target set is known before actual retrieval of the pages begins. This situation occurs relatively frequently in query processing. Retrieval by means of an index is the most typical example but there are other situations where the target set is known before retrieval begins.

According to the linear model, the time for a read request only consists of positioning time and transfer time. Furthermore, both are assumed to be constants in the linear model. In the following, we take the time required to transfer a page as the cost unit and express the cost in terms of page transfers. Let P denote the ratio of positioning time to transfer time. The cost of a read request transferring f pages is then $P + f$ (page transfers).

Whenever a sequence of pages is read into main memory, a sufficiently large buffer area must be available. We assume that buffer space for at most p pages, $p \geq 1$, is available. The problem then is how to minimize the overall cost of reading the required pages into main

memory. An obvious way of reducing the cost is as follows: whenever there is a contiguous sequence of target pages (at most p pages), read all of them into main memory with a single request. If the transfer time is significantly less than the positioning time, it may be worthwhile reading some empty pages if this reduces the number of read requests. For example, consider a situation where a target page is followed by an empty page which is followed by a target page. Overall cost is (almost always) reduced if all three pages are read with a single request, instead of reading the two target pages using separate requests.

In many operating systems, in particular several variants of UNIX, there are two suitable operations for implementing read requests. An *ordinary read* transfers a contiguous sequence of pages from the disk into a contiguous area of main memory. A *vector read* can be used to transfer the pages into several non-contiguous buffers. The advantage of a vector read is that all empty pages of a read request can be assigned to the same position in the buffer. Thus, at most one page of the buffer is sacrificed for collecting the empty pages of a read request. We first analyze the case of ordinary reads and then the case of vector reads.

Definition 4.1.1 *Let Q be a subset of the set F (the file) which is in turn a subset of $\{1,\ldots,N\}$. and p, $p \geq 1$, an integer (representing the buffer capacity). Let the tuple (s,t) denote a read request reading t pages beginning from page s. Then a sequence $\delta = ((s_1,t_1),\ldots,(s_m,t_m))$ is a* read schedule *for Q, if*

1. $t_i \leq p$ *for every* $i \in \{1,\ldots,m\}$
2. *for every $q \in Q$, there exists a tuple (s_i,t_i) in δ such that $s_i \leq q < s_i + t_i$*

The read schedule is ordered *, if*

3. $s_i < s_{i+1}$ *for every* $i \in \{1,\ldots,m-1\}$

Example: Consider the file and target set illustrated below. The file is assumed to occupy the contiguous area completely. A target page is indicated by 1 and an empty page by 0.

1	0	1	0	0	1	1	0	1	0	0	0	1	1	0	1

Assuming a buffer with 4 pages, ((1,3), (6,2), (9,1), (13,4)) is an example of an ordered read schedule. This schedule is interpreted as follows: the first read request reads pages 1,2 and 3, the second reads 6 and 7, and so on.

Definition 4.1.2 *Let $C(\delta)$ denote the cost of a read schedule δ and Δ the set of all possible read schedules for a given target set Q and buffer size p. The* subset reading problem *is then to find a read schedule δ_{opt} such that*

$$C(\delta_{opt}) = \min_{\delta \in \Delta} C(\delta) \tag{4.1}$$

Up to section 4.5, we assume that the cost of a read schedule $\delta = ((s_1, t_1), \ldots, (s_m, t_m))$ is computed as

$$C(\delta) = \sum_{i=1}^{m} (P + t_i). \tag{4.2}$$

This cost function is admittedly simplistic. A more detailed cost model would have to consider the geometry of the disk, the actual layout of the file on the disk, the seek times and rotational delays incurred, and the time to process the records on a page. Such cost models are studied in the following chapters.

4.2 Optimal Read Schedules

In this section, we show that an optimal read schedule can be found by computing the shortest path in an appropriately constructed graph. The graph is acyclic with positive edge weights and any standard shortest-path algorithm can be used. We first state two lemmas which show that only a restricted class of read schedules need be considered. Note, however, that the lemmas do not necessarily hold under a different cost model.

Lemma 4.2.1 *Any optimal read schedule, $\delta_{opt} = ((s_1, t_1), \ldots, (s_m, t_m))$, has the following two properties:*

1. *$s_j + t_j \leq s_i$ or $s_i + t_i \leq s_j$ for every $i, j \in \{1, \ldots, m\}$, $i \neq j$*

2. *for every $i \in \{1, \ldots, m\}$, $s_i \in Q$ and $s_i + t_i - 1 \in Q$*

In other words, an optimal read schedule must have non-overlapping reads and every read request must begin and end with a target page. Both properties are rather obvious so we will only outline the proof.

Proof: To prove that the first property must be satisfied, assume that a schedule contains two overlapping reads: (s_i, t_i) and (s_j, t_j). If (s_i, t_i) is a subset of (s_j, t_j) (or vice versa), the cost can be reduced by eliminating (s_i, t_i) from the schedule. If the two reads overlap, but neither is a subset of the other, the transfer cost is reduced if the common pages are eliminated from one of the reads. It follows that a read schedule containing overlapping reads cannot be optimal.

If a read request (s_i, t_i) begins with an empty page, we can reduce the transfer cost simply by changing it to $(s_i + 1, t_i - 1)$. The same applies if a read request ends with an empty page. It follows that an optimal schedule must satisfy property two. □

Lemma 4.2.2 *Let δ be a read schedule satisfying the properties of the previous lemma and δ' be the equivalent ordered schedule, that is, containing exactly the same read requests but listed in ascending order on the first component (s_i). Then $C(\delta) = C(\delta')$.*

Proof: The proof follows immediately from the observation that $C(\delta')$ is simply a reordering of the terms in $C(\delta)$. □

An ordered schedule satisfying the two properties of Lemma 4.2.1 will be called a *regular schedule*. The two lemmas guarantee that we need only consider regular schedules. To find an optimal schedule, we create a *schedule graph* from which all regular read schedules can be determined. The schedule graph is created as follows:

1. There is one node for each member (page) of the target set Q. The node corresponding to member (page) i is labeled i. There is one (initial) node labeled 0.

2. Let i denote a node and j the node with the next higher node label. For every node i, there is an edge

   (a) from node i to node j. The weight of the edge is $P + 1$.

   (b) from node i to every node k such that $k - j < p$, $j < k \leq N$. The weight of the edge is $P + (k - j + 1)$.

3. There are no other nodes and edges.

Figure 4.1: Target set and corresponding schedule graph

Let M denote the maximum node label occurring in the graph. Every path from node 0 to node M represents a read schedule and each edge in the path represents a read request. Consider an edge from a node i to a node k. If there are no nodes between i and k, that is, no nodes with labels in the range $i+1$ to $k-1$, then the edge represents the read request $(k, 1)$. Otherwise, the edge represent the read request $(j, (k-j+1))$ where j denotes the node with the next higher label after i. In other words, each edge points to the last page of a read request. The sum of the edge weights of a path is equal to the cost of the read schedule.

An example target set and schedule graph are shown in Figure 4.1. The file consists of 16 pages and there are 8 target pages. The buffer is assumed to have a capacity of 4 pages ($p = 4$). If, for example, a read request ends at page 3, there are three possibilities for the next request. First, we can read page 6 only, requiring the transfer of one page. Second, we can read pages 6 and 7, requiring the transfer of two pages. Third, we can read pages 6,7,8 and 9, requiring the transfer of four pages. Note that this request reads page 8 although it is an empty page. The shortest path for $P = 2$, and thus the optimal schedule, is indicated by bold edges.

Theorem 4.2.3 *The shortest path from node 0 to node M in the schedule graph defines an optimal read schedule.*

Proof: To prove the theorem, we must first prove that (a) every path from node 0 to node M represents a regular schedule and (b) every regular schedule is represented in the graph. Part (a) follows directly from the construction of the graph. Hence, we need only show that every regular read schedule is represented by a path in the graph.

Assume that there exists a regular read schedule which is not represented by any path in the graph. Then the schedule must contain at least one read request for which there is no corresponding edge in the graph. Assume that this read request begins with page j and ends with page k, $k \geq j$. Because the schedule is regular (property 2 of Lemma 4.2.1), pages j and k must be target pages and consequently the graph also contains a node j and a node k. Let i denote the node immediately preceding node j. An edge from node i to node k would represent the read request and we must show that such an edge exists. There are two cases to consider: $j = k$ and $j < k$. For the case $j = k$, the existence of the edge follows from 2(a) in the definition of the graph. For the case $j < k$, we note that $k - j < p$ must be true. Otherwise the schedule would be invalid. From this observation and point 2(b) of the definition of the graph, it follows that there exists an edge from node i to node k. This contradicts the assumption that the read request is not represented in the graph.

The construction of the graph guarantees that there is always an edge between two adjacent nodes. Hence, at least one path from node 0 to node M always exists. It follows that the shortest path from node 0 to node M defines an optimal read schedule. □

Once the graph has been constructed, we can use any shortest-path algorithm to find an optimal read schedule. However, it is questionable whether it is worthwhile in practice to compute an optimal schedule. The model ignores many factors, for example, queuing delays and time to process records. Hence, a theoretically optimal schedule may not in practice be optimal. Furthermore, we cannot estimate the performance of schedules produced by this algorithm (without knowing the exact layout of the target set), something which is needed for query optimization purposes. In the next section, we present a simple algorithm which produces read schedules that are very close to optimal.

4.3 Simplified Algorithm

The basic idea of the algorithm is simple: start reading from the next target page, stop reading either at the last target page that fits into the buffer area or at the last target page before a long stretch of empty pages. This algorithm is based on the observation that it is often cheaper to read a few empty pages than to skip them. A sequence of empty pages is called a *gap*. Let m denote the maximum sequence of empty pages that will be read, that is, when a gap of $m+1$ or more empty pages is encountered, the read request ends with the last target page before the gap.

```
Algorithm ReadSubset(F: File; Q: TargetSet; B: Buffer; p: BfrSize; m: GapSize);
BEGIN
  end := 0;
  REPEAT
    start := NextTargetPage( F, Q, end);
    prev := start;
    LOOP
      next := NextTargetPage( F, Q, prev);
      IF (next > m + 1 + prev) OR (next ≥ p + start) OR (next > N) THEN
        end := prev; EXIT
      END;
      prev := next;
    END;
    ReadIntoBuffer(F, B, start, end);
    Process records in B;
  UNTIL (next > N);
END ReadSubset;
```

The function *NextTargetPage(F,Q,j)* is assumed to compute the position of the first target page after page j. If none exists, it returns a value greater than N. The procedure *ReadIntoBuffer(F, B, start, end)* reads pages $start, start+1, \ldots, end$ from file F into buffer

p	*ReadSubset*	Optimum	Diff.(%)
2	10.079	10.079	0.0000
4	8.883	8.866	0.1948
6	8.206	8.153	0.6527
8	7.818	7.715	1.3372
10	7.585	7.454	1.7552
12	7.415	7.293	1.6660
14	7.299	7.184	1.5967
16	7.209	7.105	1.4669
18	7.144	7.046	1.3847
20	7.090	7.004	1.2371
24	7.019	6.948	1.0190
28	6.971	6.914	0.8242

Table 4.2: Cost (per target page) of read schedules produced by *ReadSubset* compared with cost of optimal schedules. ($P = 10, \alpha = 0.1, N = 100,000$)

B. The first and the last page of a read request are always target pages. The algorithm adds pages to the read request until one of three conditions is satisfied: a gap of $(m + 1)$ or more empty pages is found, the next target page is past the end of the buffer, or the end of the file has been reached.

This simple algorithm does not guarantee optimal read schedules. We have performed extensive simulation experiments which indicate that the schedules produced are close to optimal. The results of one set of experiments are listed in Table 4.2. The results are for a file with 100,000 pages and a target set of 10,000 (randomly chosen) pages. The figures are averages of 20 experiments. The table lists the cost per target page of schedules produced by *ReadSubset*, the cost of optimal schedules per target page and the relative difference. The cost of schedules produced by *ReadSubset* depends on the value of the maximum gap size (m). For each buffer size (p), the value of m was chosen so as to produce the best schedule (minimal cost). As shown in the table, the (best) read schedules produced by algorithm *ReadSubset* were within 2% of the optimum. Similar results were obtained from other experiments.

The behavior of the algorithm depends on the buffer size (p) and maximum gap size (m). Five different cases are discussed below. We illustrate the discussion using the example file and target set shown in Figure 4.3. The value of P is assumed to be 2.

1. $p = 1$:

   Setting $p = 1$ produces schedules reading one target page at a time, that is, the traditional approach. The cost of the schedule produced for the example file is 24 $(8 * 2 + 8)$.

2. $m = 0$, $p = \infty$:

   An unlimited amount of buffer space is assumed. This parameter setting results in schedules where each request reads a contiguous sequence (cluster) of target pages. The algorithm takes advantage of whatever clustering there is in the file but never reads an empty page. The cost of the resulting schedule is 20 $(6 * 2 + 8)$. This case was analyzed in [McF90].

3. $0 < m < \infty$, $p = \infty$:

   This version also assumes an unlimited amount of buffer space. To reduce the number of positioning operations, (short) gaps of empty pages are read instead of skipped. The value of m affects the cost of the read schedules produced. Setting $m = 2$ produces a read schedule with cost 17 $(2 * 2 + 13)$ for our example file.

4. $m = \infty$, $1 < p < \infty$:

   A buffer of limited size is used but there is no restriction on the length of gaps. In other words, a request reads every page up to and including the last target page covered by the buffer. For $p = 7$, we obtain a schedule with cost 20 $(3 * 2 + 14)$ for our example file.

5. $0 < m < \infty$, $1 < p < \infty$:

   This is the most general case: a buffer of limited size is available and the maximum gap size is also limited. For $p = 7$ and $m = 2$, the resulting schedule has a cost of 18 $(3 * 2 + 12)$.

4.4 Analysis

In this section, we analyze the expected cost of read schedules produced by algorithm *Read-Subset*, first for two special cases and then for the general case. The analysis is structured in this way because (a) simple closed formulas can be derived for the two special cases but not

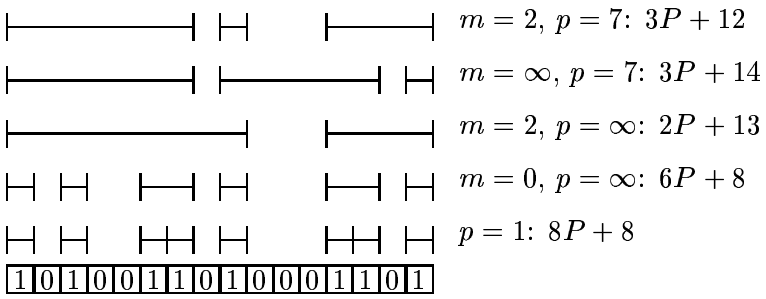

Figure 4.2: Example file and read schedules for different parameter settings

for the general case and (b) the cost formulas for the two special cases are upper and lower bounds for the general case. The analysis is asymptotic, that is, for $N, b \to \infty$ and keeping $\alpha = b/N$ constant. The cost is expressed as the expected cost per target page. Target pages are assumed to be randomly distributed over the file. The probability of a page being a target page is α.

4.4.1 Unlimited Gaps, Limited Buffers

We begin by considering the case $m = \infty$ and $p < \infty$. We first derive the expected number of read requests and then the expected number of pages transferred per request.

The expected number of *target* pages transferred by a read request is $1 + (p-1)\alpha$. The first page is always a target page. Each one of the remaining $p-1$ pages covered by the buffer is a target page with probability α. The number of read requests per target page is then simply

$$\frac{1}{1 + (p-1)\alpha}$$

A read request transfers some number of pages (target pages and empty pages). The number of pages transferred equals p minus the number of empty pages located at the end of the buffer. The probability u_i that there are exactly i empty pages at the end of the buffer is given by

$$u_i = \begin{cases} \alpha(1-\alpha)^i & \text{for } i < p-1 \\ (1-\alpha)^{p-1} & \text{for } i = p-1 \end{cases}$$

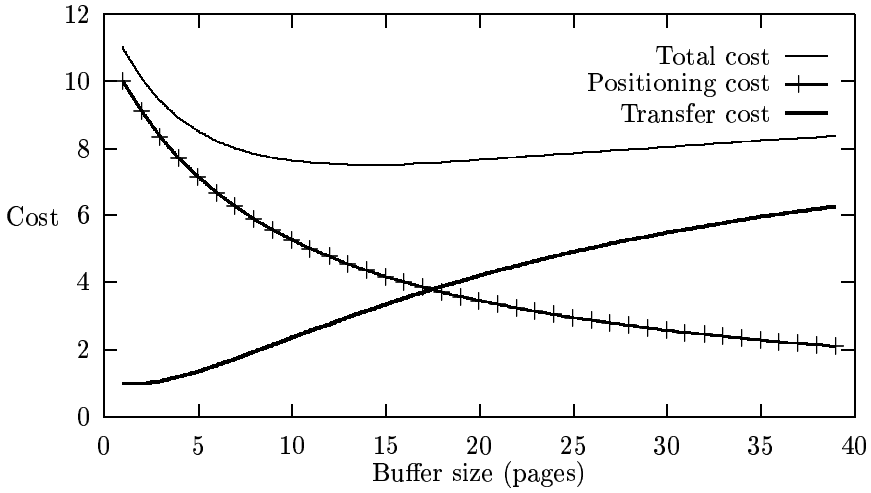

Figure 4.3: $lcost(0.1, p, \infty)$ in page transfers as a function of buffer size ($P = 10$)

The expected number of empty pages at the end of the buffer is then

$$\begin{aligned} E &= \sum_{i=0}^{p-1} i u_i \\ &= (p-1)(1-\alpha)^{p-1} + \sum_{i=1}^{p-2} \alpha(1-\alpha)^i i \\ &= (p-1)(1-\alpha)^{p-1} + \\ &\quad \frac{(1-\alpha)}{\alpha}(1-(1-\alpha)^{p-2} - (p-2)\alpha(1-\alpha)^{p-2}) \\ &= \frac{1-\alpha}{\alpha}(1-(1-\alpha)^{p-1}) \end{aligned}$$

The expected number of pages actually transferred is $p - E$. Combining the expected number of read requests and the number of pages transferred, we obtain the following formula for the expected cost per target page:

$$lcost(\alpha, p, \infty) := \frac{P + p - \frac{1-\alpha}{\alpha}(1-(1-\alpha)^{p-1})}{1 + (p-1)\alpha} \tag{4.3}$$

In Figure 4.3 the expected cost is plotted as a function of the buffer size. The positioning cost and transfer cost are also shown to illustrate the behavior of the two components of the cost function. As expected, the positioning cost decreases (fewer requests) and the transfer

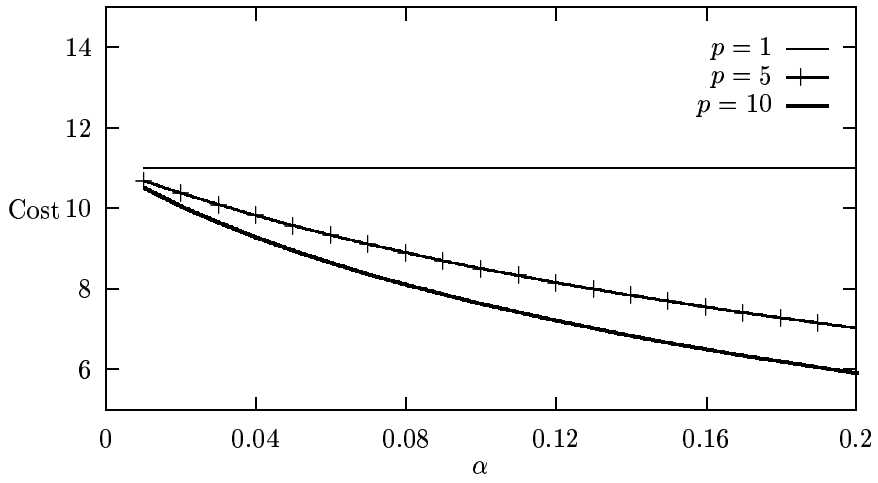

Figure 4.4: $lcost(\alpha, p, \infty)$ in page transfers as a function of α, ($p = 1, 5, 10$, $P = 10$)

cost increases (reading more empty pages) with the buffer size. The cost function has a global minimum. In particular, very large buffers result in a higher overall cost because the number of empty pages read increases.

In Figure 4.4, the expected cost is plotted for three different buffer sizes ($p = 1, 5, 10$). The figure clearly shows the benefit of using a larger buffer. For $\alpha = 0.2$, increasing the buffer size from one page to 10 pages, reduces the expected cost by 50%.

For the case illustrated in Figure 4.3, the expected cost has a minimum. The optimal buffer size cannot be derived analytically but can be computed numerically quite easily. Table 4.3 shows the optimal buffer size as a function of the fraction of target pages. We have also listed, for a few different page capacities, the fraction of records in the response set that corresponds to each fraction of target pages in the file. These results were obtained by using the Waters-Yao formula [Wat76, Yao77]. At first, the optimal buffer size increases slowly with increasing α. However, when α increases to 17%, the lowest cost occurs for $p = \infty$. This simply means that for large α the best policy is to read the whole file with one read request. In practice, this translates to reading the file sequentially using (very) large buffers.

The results are somewhat surprising if we consider the fraction of records in the response

α (in %)	p^*	*Records in the response set (in %)*			
		$c = 5$	$c = 10$	$c = 20$	$c = 40$
1.0	12	0.2010	0.1005	0.0503	0.0251
2.0	12	0.4041	0.2020	0.1010	0.0505
4.0	12	0.8164	0.4082	0.2041	0.1021
6.0	12	1.2375	0.6188	0.3094	0.1547
8.0	13	1.6676	0.8338	0.4169	0.2085
10.0	14	2.1072	1.0536	0.5268	0.2634
12.0	15	2.5567	1.2783	0.6392	0.3196
14.0	18	3.0165	1.5082	0.7541	0.3771
15.0	20	3.2504	1.6252	0.8126	0.4063
16.0	25	3.4871	1.7435	0.8718	0.4359
17.0	∞	3.7266	1.8633	0.9316	0.4658

Table 4.3: Optimal buffer size (p^*) as a function of α, ($P = 10$)

set needed to exceed this critical point. For example, when $c = 20$, our model indicates that the cheapest way to answer a query is to scan the whole file even when the response set contains as little as 1% of the records.

4.4.2 Limited Gaps, Unlimited Buffer

Next we consider the case $p = \infty$ and $0 < m < \infty$. A gap is a contiguous sequence of empty pages delimited on the left and the right by a target page. A *cluster* is a contiguous sequence of pages with the following properties:

- the first and the last page of the sequence are target pages
- the sequence does not contain any single gap longer than m
- the sequence is delimited on the left and the right by gaps strictly longer than m

Because $p = \infty$, each read request will read exactly one cluster. To calculate the expected cost of a read schedule, we calculate the expected number of clusters and their expected length.

Consider an arbitrary page in the file. This page begins a cluster if it is a target page and to the left of it is a gap longer than m. Hence, the probability of a page beginning a cluster is

$$\sum_{j>m} \alpha(1-\alpha)^j \alpha = \alpha(1-\alpha)^{m+1}$$

The expected length of a cluster, including the gap separating it from the next clusters, is then $1/(\alpha(1-\alpha)^{m+1})$. The next step is to compute the expected length of the gap separating two clusters. The probability of the gap being of length $m+1+j$, $j \geq 0$, is $\alpha(1-\alpha)^j$. The expected length of the gap is therefore

$$\sum_{j\geq 0}(m+1+j)\alpha(1-\alpha)^j = m + 1/\alpha$$

The pages which are part of the gap will not be read. The expected length of a cluster, counting only the pages read, is therefore

$$\frac{1}{\alpha(1-\alpha)^{m+1}} - m - \frac{1}{\alpha}$$

The expected cost per page in the file is then

$$\alpha(1-\alpha)^{m+1}(P + \frac{1}{\alpha(1-\alpha)^{m+1}} - m - \frac{1}{\alpha})$$

which can be simplified to

$$\alpha P(1-\alpha)^{m+1} + 1 - (1-\alpha)^{m+1}(1+m\alpha)$$

Finally, by dividing by α, we obtain the expected cost per target page

$$\begin{aligned} lcost(\alpha, \infty, m) &= P(1-\alpha)^{m+1} + \\ &\quad \frac{1}{\alpha}(1-(1-\alpha)^{m+1}(1+m\alpha)) \end{aligned} \tag{4.4}$$

In Figure 4.5, the expected cost has been plotted as a function of m. Positioning and transfer costs are also plotted to show their contribution to the total cost. The cost function has a minimum at about $m = 9$ for the case shown in the figure.

For the purpose of finding the minimum of the function $cost(\alpha, \infty, m)$, (for a given value of α), we can treat m as being defined over the real numbers. The value of m that minimizes the function can then be determined by taking the derivative of the function with respect to

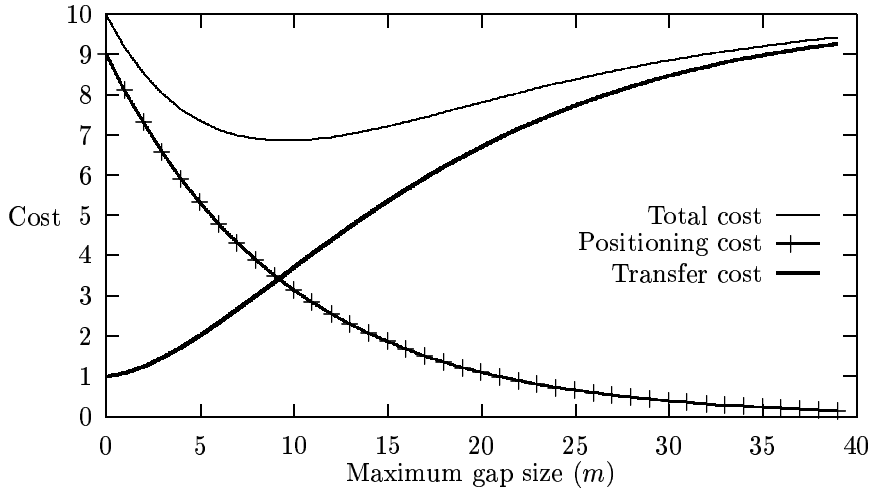

Figure 4.5: $lcost(0.1, \infty, m)$ as a function of m, $(P = 10)$

m. If m^* denotes the real value minimizing the function, the optimal integer value is then either $\lceil m^* \rceil$ or $\lfloor m^* \rfloor$.

$$\begin{aligned} m^* &= P - \frac{1}{\alpha} - \frac{1}{\ln(1-\alpha)} \\ &= P - \frac{1}{2} - \frac{1}{12}\alpha - O(\alpha^2) \end{aligned}$$

Figure 4.6 shows the expected cost as a function of α, for four different values of m. For $P = 10$, the integer minimum for m is either 9 or 10. The graph corresponding to $cost(\alpha, \infty, 20)$ is clearly above the one for $cost(\alpha, \infty, 9)$.

4.4.3 Limited Gaps, Limited Buffer

In this section we analyze the general case of the algorithm, that is, the case where $m < \infty$ and $p < \infty$. Consider a read request filling some number of pages in the buffer. Let $Q(i, j)$, $1 \leq i, j \leq p$, denote the probability that page j in the buffer receives the i-th target page read by this request. Since page j in the buffer can receive at most the j-th target page, it follows $Q(i, j) = 0$ for $i > j$. Furthermore, the first page is always a target page. Hence, for

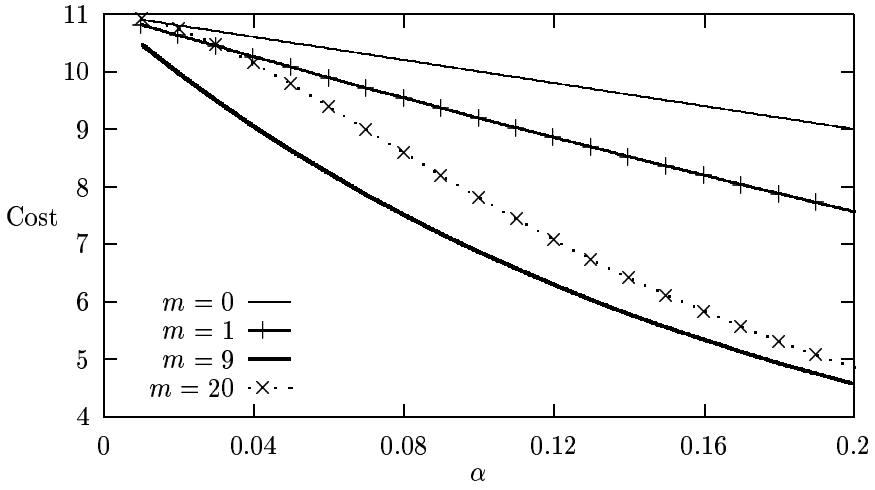

Figure 4.6: $lcost(\alpha, \infty, m)$ as a function of α, $(P = 10)$

$i = 1$ we have

$$Q(1,1) = 1 \quad and \quad Q(1,j) = 0 \;\; for \;\; 2 \leq j \leq p$$

Now consider the case $i > 1$ and $j \leq i$. Assume that the $(i-1)$-th target page is in position $j-k$ $(k \geq 1,\ 1 \leq j-k \leq m+1)$. The conditional probability that the next target page is in position j is then $(1-\alpha)^{k-1}\alpha$. Consequently, the probability that the i-th target page is in position j can be computed by summing over all possible positions for the $(i-1)$-th target page. The $(i-1)$-th target page cannot be to the left of page $j-(m+1)$ and $j-k > 0$ must always hold. It follows that for $j \geq i$ $Q(i,j)$ can be computed by the recurrence relation

$$Q(i,j) = \sum_{k=1}^{\min(j-1,m+1)} Q(i-1, j-k)\alpha(1-\alpha)^{k-1}$$

Let $Q_{stop}(j)$, $1 \leq j \leq p$ denote the probability that exactly j pages are transferred by a read request. There are two cases to consider. If $p-j > m$, a gap of length $m+1$ (or more) follows page j. This occurs with probability $(1-\alpha)^{m+1}$. Otherwise, that is $p-j \leq m$, $p-j$ empty pages follow and the end of the buffer is reached. This occurs with probability

$(1-\alpha)^{p-j}$. Combining the two cases, we obtain

$$Q_{stop}(j) = \begin{cases} (1-\alpha)^{p-j} & \text{if } p-j \leq m \\ (1-\alpha)^{m+1} & \text{if } p-j > m \end{cases}$$

The probability of a page being a target page is independent of its position in the file. For $j > 1$, the probability $Q(i,j)$ is independent of the probability $Q_{stop}(j)$. Therefore, the probability that exactly i target pages are contained in a buffer is given by

$$\sum_{j=1}^{p} Q(i,j)Q_{stop}(j)$$

The expected number of target pages per read request can then be computed as

$$E_{target} = \sum_{i,j\geq 1}^{p} Q(i,j)Q_{stop}(j)i$$

and the expected number of pages transferred as

$$E_{total} = \sum_{i,j\geq 1}^{p} Q(i,j)Q_{stop}(j)j$$

Finally, the expected cost per target page is given by

$$lcost(\alpha,p,m) = \frac{1}{E_{target}}(P + E_{total}) \tag{4.5}$$

The expected cost is plotted in Figure 4.7. The positioning and transfer costs are shown separately. Note that the results are not for a fixed value of m. For each buffer size p, the best value of m was chosen.

The three functions $lcost(\alpha,p,\infty)$, $lcost(\alpha,\infty,m)$ and $lcost(\alpha,p,m)$ are plotted in Figure 4.8. The three curves indicate that the functions $lcost(\alpha,p,\infty)$ and $lcost(\alpha,\infty,m)$ might be special cases of the general function $lcost(\alpha,p,m)$ and that the two simple functions are upper and lower bounds on $lcost(\alpha,p,m)$. The simpler closed formulas are fairly good approximations of the most general case.

4.5 A Cost Model for Vector Reads

So far we have assumed that target pages are read by using the ordinary read command, i.e. a contiguous area of the disk is copied into a contiguous area of the buffer. Thus, the buffer

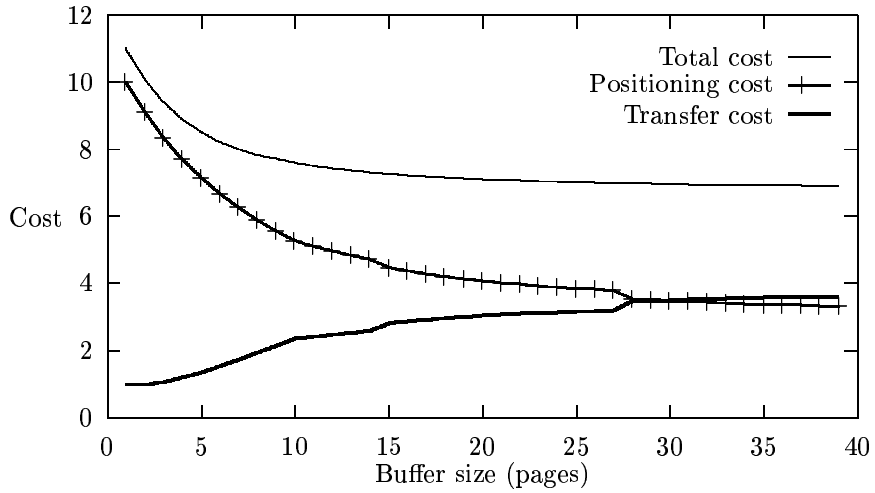

Figure 4.7: $lcost(0.1, p, m^*)$ as a function of p, $(P = 10)$

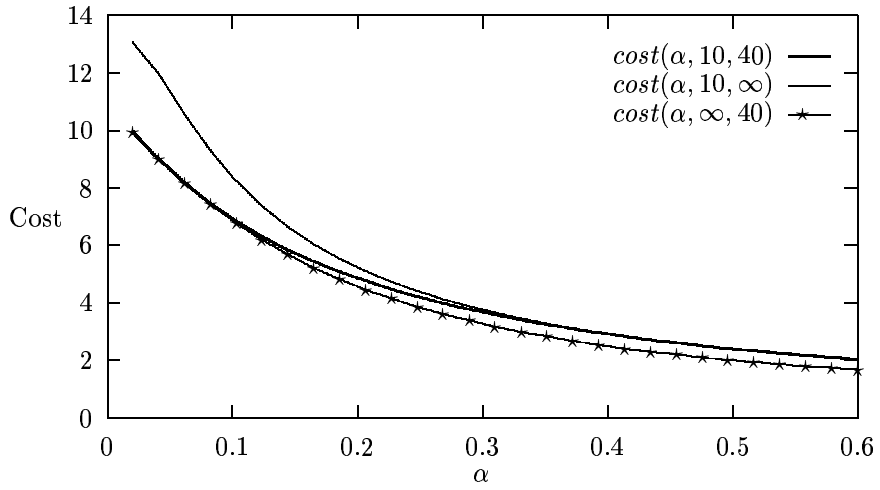

Figure 4.8: Comparison of the cost functions $lcost(\alpha, p, m)$ for $(m, p) = (10, \infty)$, $(10, 40)$, $(\infty, 40)$, $P = 10$

might contain a high fraction of empty pages. Many operating systems (in particular several variants of UNIX) offer another operation, called a vector read, for reading multiple pages in a single request. A vector read transfers a contiguous sequence of pages from secondary storage into a non-contiguous collection of buffer pages. In particular, a single buffer page may receive several pages. This property can be used for assigning all empty pages of a read request to the same buffer page. Then, at most one buffer page is sacrificed for receiving empty pages. In the following, we address the problem of finding optimal read schedules under the assumption that a read request is implemented as a vector read. First, we briefly introduce the modified problem definition and a modified algorithm.

Definition 4.5.1 *Let Q be a subset of the set $F = \{1, \ldots, N\}$ and p, $p \geq 1$, an integer. Let the tuple (s, u, v) denote a read request reading $u + v$ pages beginning from page s where u and v denote the number of empty and target pages, respectively. Then a sequence $\delta = ((s_1, u_1, v_1), \ldots, (s_m, u_m, v_m))$ is a* v-read schedule *for Q, if*

1. $v_i < p$ *for every* $i \in \{1, \ldots, m\}$ *with* $u_i \neq 0$
2. $v_i \leq p$ *for every* $i \in \{1, \ldots, m\}$ *with* $u_i = 0$
3. *for every* $q \in Q$, *there exists a tuple* (s_i, u_i, v_i) *in* δ *such that* $s_i \leq q < s_i + u_i + v_i$

Instead of using cost formula 4.2, we assume that the cost of a v-read schedule is computed as

$$C(\delta) = \sum_{i=1}^{m} (P + u_i + v_i). \tag{4.6}$$

Similarly to our previous cost model presented in section 4, the solution of computing an optimal v-read schedule can be reduced to solving a shortest-path problem in an acyclic graph. The graph can be constructed following almost the same approach as for the graph described in section 4. However, we are primarily interested in a simple approximate algorithm which produces v-read schedules close to the optimum and whose cost can be easily computed. In order to support vector reads, algorithm *ReadSubset* requires only a few modifications. The modified algorithm *VReadSubset* follows.

```
Algorithm VReadSubset(F: File; Q: TargetSet; B: Buffer; p: BfrSize; m: GapSize);
BEGIN
  end := 0;
  REPEAT
    adr[1] := NextTargetPage( F, Q, end);
    ones := 1; zero_flag := 0;
    LOOP
      adr[ones+1] := NextTargetPage( F, Q, adr[ones]);
      IF (adr[ones+1] > m + 1 + adr[ones]) OR (ones ≥ p - zero_flag) OR (adr[ones+1] > N)
      THEN
        end := adr[ones]; EXIT
      END;
      IF adr[ones+1] > adr[ones] + 1 THEN
        zero_flag := 1;
      END;
      ones := ones+1
    END;
    VRead(F, B, adr, ones);
    Process records in B;
  UNTIL (adr[ones+1] > N);
END VReadSubset;
```

The procedure *VRead(F,B,adr,ones)* reads pages with addresses $adr[1]$, $adr[1]+1$, .., $adr[2]-1$, $adr[2]$, $adr[2]+1$, .., $adr[ones]$ from file F into the buffer B. Note that the target pages $adr[1]$, $adr[2]$, .., $adr[ones]$ are assigned to the first *ones* pages in the buffer. All empty pages are read into the p-th buffer page. The algorithm adds pages to the read request until one of three conditions is satisfied: a gap of $(m+1)$ or more empty pages is found, the next target page causes an overflow of the buffer, or the end of the file has been reached. Let us mention that the algorithm *VReadSubset* indeed produces schedules which are close to optimal.

Most interesting is the analysis of the algorithm *VReadSubset* and a comparison of *VReadSubset* with *ReadSubset*. Note that both algorithms produce the same schedule if the buffer is unlimited. In the following, we restrict our attention to the most general case, i.e. we assume

limited gaps and a limited buffer. The following analysis is similar to the one presented in section 4.4.3.

Consider a read request that reads pages labeled $1, 2, 3, \ldots$ into the buffer with p pages. Let $R(i,j)$, $1 \leq i \leq p$, $j \geq 1$ denote the probability that page j is a target page and is assigned to the i-th buffer page. Note that $R(i,j) = 0$ for $i < j$. Since page 1 always has to be a target page, it will always be the first page in the buffer. Hence for $i = 1$ we obtain

$$R(1,1) = 1 \quad and \quad R(1,j) = 0 \;\; for \;\; j \geq 2$$

Furthermore, the p-th page of the buffer will be filled with a target page only if all of the other buffers are filled up with target pages and none of the pages read from the file was an empty page. Therefore, for $i = p$ we have

$$R(p,p) = 1 \quad and \quad R(p,j) = 0 \;\; for \;\; j \geq 1, j \neq p$$

Now consider the case $i \in \{2, \ldots, p-1\}$ and $j \geq i$. Assume that the $(i-1)$-th target page has the label $j-k$ ($k \geq 1$, $1 \leq j-k \leq m+1$). The conditional probability that the next target page has the label j is then $(1-\alpha)^{k-1}\alpha$. Consequently, the probability that the i-th target page has label j can be computed by summing over all possible positions for the $(i-1)$-th target page. The $(i-1)$-th target page cannot be to the left of page $j-(m+1)$ and $j-k > 0$ must always hold. It follows that for $j \geq i$ $R(i,j)$ can be computed by the recurrence relation

$$R(i,j) = \sum_{k=1}^{\min(j-1,m+1)} R(i-1, j-k)\alpha(1-\alpha)^{k-1}$$

Let $R_{stop}(i,j)$, $1 \leq i \leq p, j \geq i$, denote the probability that exactly j pages are transferred by a read request and that i of the j pages are target pages. There are four cases to consider. If $i = p$, the buffer is completely filled and thus no more pages can be read. If $i = p-1$ and $i < j$, $p-1$ target pages are transferred and at least one of the buffer pages is reserved for the empty pages. The buffer is then already full. If $i = j = p-1$, it is possible to read the p-th page into the buffer. However, this page is not read, if the p-th page is an empty page. This occurs with probability $(1-\alpha)$. Otherwise, that is $i < p-1$, a gap of length $m+1$ (or more) follows after the j-th page. This occurs with probability $(1-\alpha)^{m+1}$. Combining the

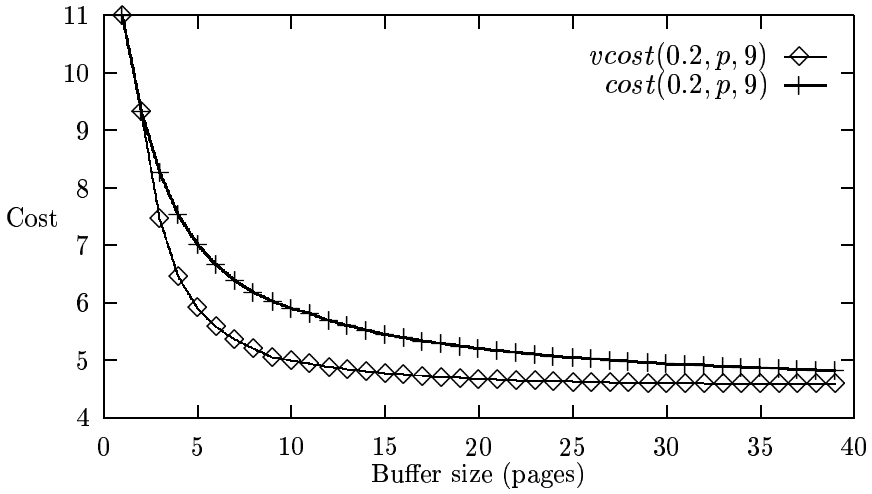

Figure 4.9: $vcost(0.2, p, 9)$ and $lcost(0.2, p, 9)$ as a function of p, $(P = 10)$

four cases, we obtain

$$R_{stop}(i,j) = \begin{cases} 1 & \text{if } i = p \\ 1 & \text{if } i = p-1, i < j \\ (1-\alpha) & \text{if } i = j = p-1 \\ (1-\alpha)^{m+1} & \text{otherwise} \end{cases}$$

Similarly to section 4.4.3, the expected number of target pages per read request can then be computed as

$$V_{target} = \sum_{i=1}^{p} \sum_{j \geq i} R(i,j) R_{stop}(i,j) i$$

and the expected number of pages transferred as

$$V_{total} = \sum_{i=1}^{p} \sum_{j \geq i} R(i,j) R_{stop}(i,j) j$$

Finally, the expected cost per target page is given by

$$vcost(\alpha, p, m) = \frac{1}{V_{target}} (P + V_{total}) \tag{4.7}$$

The expected cost is plotted in Figure 4.9. Additionally, the expected cost $lcost(0.2, p, 9)$ is shown to be higher than $vcost(0.2, p, 9)$, particularly for a small number of buffers.

If pages are always transferred exactly in the order specified in the read request, a separate buffer page for empty pages is not needed. We can let the last target page overwrite the last buffer which was used for collecting empty pages.

4.6 Discussion

In this chapter we investigated how to rapidly retrieve a set of pages from a file stored in a contiguous area on disk. The principal idea is to reduce the response time by making use of multi-page requests that transfer multiple adjacent pages. If it is advantageous to do so, a multi-page request may include some gaps. A gap is a contiguous sequence of empty pages, that is, pages not containing any required records.

The algorithms and their analysis presented in this chapter are based on the assumptions of the linear model. We showed that for the linear model, an optimal read schedule can be found by computing the shortest path in a certain graph. The performance was analyzed for a simplified algorithm which was found to produce close-to-optimal read schedules. The most general cost formula depends on three parameters: the fraction of the target pages in the file, the maximum size of a gap that is allowed to be in a read request, and the size of the buffer. The buffer size is an upper bound for the number of (target) pages that can be transferred in a single multi-page request. The results of our analysis have demonstrated the benefits of using multi-page read requests but under the assumptions of the fairly simplistic linear model. For the linear model, a disk is basically assumed to be a linear sequence of pages. Thus, we have disregarded the partition of disks into cylinders and tracks and the existence of multiple read/write-heads. Furthermore, we simply charged a fixed cost for each positioning operation. This is clearly a blatant simplification. In the next chapters, the problem will be investigated under the assumption of more complex disk models. These models are closer to the real architecture of a disk, but, unfortunately, it is much harder to obtain an analysis.

Chapter 5

A Cost Function for the Idealized Disk Model

In the following chapter, the problem of reading a set of disk pages is investigated under the assumptions of the idealized disk model (*IDM*). This model comes closer to the actual disk architecture than the simple linear model. The IDM takes into account that a disk consists of cylinders, tracks and pages. Moreover, the availability of multiple read/write heads is also considered which allows data to be read from different surfaces in a single revolution of the disk.

The chapter is structured into seven sections. In the first section, a summary is given on the most important assumptions of the IDM. The problem is then precisely stated in the second section. In section three, a simple algorithm is given for reading a set of disk pages. In section four, the expected cost for reading a set of disk pages is analyzed under the assumption that all required pages are located on the same cylinder of the disk. In section five, the analysis is extended to the case when pages are distributed over several cylinders. In the first five sections, the problem is studied assuming that the disk completely belongs to the underlying query. In section six, the issue of several queries being performed at the same time is addressed. In the final section, we summarize our results.

5.1 Assumptions of the Idealized Disk Model

In the IDM, the total cost of reading a page is assumed to be the sum of its three major components: seek time, rotational delay and transfer time. We do not consider the overhead of the disk controller (which is generally less than 1 ms) and channel contention (when multiple disks share a common disk controller). In order to make the analysis tractable, head switch time is also not considered as part of the total cost. Seek time and rotational delay depend on the previous position of the disk arm, whereas the transfer time is assumed to be constant for all pages on the disk. The transfer time of a page is taken as the cost unit and the cost is expressed in terms of page transfers. Consider that page A has to be read from disk. Let B be the page where the read/write head was positioned on when the request for page A received service. Furthermore, let $s(A, B)$ and $r(A, B)$ denote the ratio of seek time and rotational delay to transfer time, respectively. The cost of reading page A is then $s(A, B) + r(A, B) + 1$. Thus, a lower bound for reading a page is 1 (page transfer).

The layout of pages on a disk is assumed to be as follows. First, each track of the disk contains PT pages where PT is an integer. Second, the starting positions of the tracks are aligned in a cylinder. These assumptions are not valid for today's magnetic disks, but they facilitate the analysis that will be presented in the following sections. In particular, a cylinder of the disk can now be represented as a two-dimensional array $C_{i,j}$ of pages where $0 \leq i < TC$ and $0 \leq j < PT$. Here, TC denotes the number of tracks. For a cylinder $(C_{i,j})$, the set $\{C_{i,j} | 0 \leq i < TC\}$ is also called the j-th *column*, $0 \leq j < PT$. It follows that at most one page can be read from a column during a revolution of the disk. Since head switches are assumed to cause no time delay, an arbitrary page from column j can be read without any time delay when the previous page was read from column $(j - 1)$ *mod* PT.

5.2 Problem Statement

Consider a file F as a collection of pages of fixed size distributed over a magnetic disk drive. In order to perform a query on the file, it is assumed that pages $T_1, \ldots, T_N$ of the file must be read from the disk into main memory. These pages are called *target pages* and the set of target pages is called the *target set*. It is assumed that the complete target set is known in

advance before actual retrieval of the pages begins.

In the following, we deal with the problem of reading the target pages into main memory as fast as possible using one or more multi-page requests. A *multi-page request* transfers a set of pages from magnetic disk into main memory without interfering of other requests. In contrast to ordinary read commands, it is not required that the pages are contiguous on magnetic disk. In order to perform a multi-page request, a sufficiently large buffer is required where the target pages can be copied to. For the sake of simplicity, we first assume to have an infinitely large buffer, i.e. the buffer size is not of concern.

Let us first assume that the target set $\{T_1, \ldots, T_N\}$ is read by using one multi-page request. Let Π_N be defined as the set of all permutations on the vector $(1, \ldots, N)$. Then, a sequence $S = (T_{i_1}, \ldots, T_{i_N})$ is called a *read schedule* for the target set $\{T_1, \ldots, T_N\}$, if $(i_1, \ldots, i_N) \in \Pi_N$. A read schedule $(T_{i_1}, \ldots, T_{i_N})$ determines the order in which target pages are read from disk.

The selected schedule has a substantial impact on the cost for the multi-page request. The reason is that the cost for reading a target page $T_{i_{j+1}}$, $1 \leq j < N$, depends on the (disk) position of the target page T_{i_j} previously read. For a given target set $\{T_1, \ldots, T_N\}$, the goal is to find a schedule $(T_{i_1}, \ldots, T_{i_N})$ such that the total elapsed time of the multi-page request is minimized. Let T_0 be the page on disk where the read/write head had been positioned on when the multi-page request received service. The total elapsed time of a multi-page request is then given by

$$s(T_{i_1}, T_0) + \sum_{j=1}^{N-1} s(T_{i_{j+1}}, T_{i_j}) + r(T_{i_1}, T_0) + \sum_{j=1}^{N-1} r(T_{i_{j+1}}, T_{i_j}) + N$$

For a given schedule $S = (T_{i_1}, \ldots, T_{i_N})$, the elapsed time of a multi-page request can be decomposed into three components. Similarly to an ordinary single-page request, we call these components seek time, rotational delay and transfer time of the schedule. The *seek time* of the schedule S is defined as

$$st(T_{i_1}, \ldots, T_{i_N}) = s(T_{i_1}, T_0) + \sum_{j=1}^{N-1} s(T_{i_{j+1}}, T_{i_j})$$

The *rotational delay* is defined as

$$rd(T_{i_1}, \ldots, T_{i_N}) = r(T_{i_1}, T_0) + \sum_{\substack{1 \leq j < N \\ s(T_{i_{j+1}}, T_{i_j}) > 0}} r(T_{i_{j+1}}, T_{i_j})$$

and the *transfer time* is given as

$$tt(T_{i_1}, \ldots, T_{i_N}) = N + \sum_{\substack{1 \leq j < N \\ s(T_{i_{j+1}}, T_{i_j}) = 0}} r(T_{i_{j+1}}, T_{i_j})$$

Let us emphasize that the transfer time of a multi-page request does not only correspond to the actual transfer time N, but it also includes the the rotational delays of the pages which are read from the same cylinder as the predecessor page (i.e. no seek time occurs).

5.3 Algorithms

For a given target set, an efficient algorithm for computing an optimal schedule (i.e. with minimum cost) has not been found so far. A common approach for computing approximate schedules is to read from a cylinder all target pages before going on to the next cylinder. Then there are the following subproblems:

1. Find an efficient algorithm for computing optimal schedules, when target pages are stored on a cylinder.

2. Find an optimal sequence for accessing those cylinders which contain at least one target page.

Under the assumptions of the idealized disk model, an appropriate algorithm can easily be found for the first subproblem. Let $(C_{i,j})$, $0 \leq i < TC$ and $0 \leq j < PT$, be a cylinder of pages where $N \leq PT * TC$ of them are target pages. In the following, PC $(= PT * TC)$ denotes the number of pages on a cylinder. Furthermore, $C_{*,j}$, $0 \leq j < PT$, denotes the set of target pages that can be found in the j-th column. Let us assume that the initial position of the disk arm is at the beginning of the track. The algorithm reads the target pages with respect to the *shortest-latency-time-first* (SLTF) policy [Den67]. One target page is read

Figure 5.1: Target pages in a cylinder

from each of the non-empty sets $C_{*,j}$, $0 \leq j < PT$, in a single disk revolution as long as one target page is unprocessed in $C_{*,j}$. In the last disk revolution required for reading the target pages, the algorithm stops reading after the read/write head has passed over the last column (i.e. farthest away from the beginning of the track) with the maximum number of target pages.

In [SF73] it was shown that the algorithm is optimal, i.e. it reads the target pages in minimum time. In order to differentiate (elapsed) time into transfer time and rotational delay, the algorithm is slightly modified. Note that the response time is not affected when the algorithm does not transfer data until the first set $C_{*,f}$, $0 \leq f < PT$, is reached with the maximum number of target pages. The corresponding time refers to the rotational delay of the schedule when all target pages are on one cylinder. Then, the transferring of pages begins and continues until the last target page is read. This time is the transfer time of the schedule.

An example is illustrated in Figure 5.1. A cylinder consists of four tracks and eight columns. Target pages received the label 1, whereas other pages received the label 0. Overall, 10 pages have to be read from disk. The maximum occurs three times, at the second, fourth and fifth column. The rotational delay then corresponds to 2 page transfers and the transfer time corresponds to 20 page transfers.

The second subproblem is to compute an optimal sequence on how the cylinders are accessed. The author is not aware of an algorithm that computes an optimal sequence in po-

N	number of target pages
$\{T_1, \ldots, T_N\}$	target set
PT	number of pages on a track
TC	number of tracks per cylinder
PC	$= PT * TC$
$C_{i,j}$	denotes the pages of a cylinder, $0 \leq i < TC$ and $0 \leq j < PT$
$(C_{i,j})$	denotes a cylinder
$C_{*,j}$	denotes the set of target pages in the j-th column, $0 \leq j < PT$
M	maximum number of target pages in a column of a given cylinder
Cyl	number of cylinders
C_F	number of cylinders occupied by the file (file cylinders)
C_{act}	number of cylinders with at least one target page
tt_X	expected transfer time
rd_X	expected rotational delay
st_X	expected seek time

Table 5.1: List of symbols

lynomial time. In general, the SCAN policy [Den67], originally proposed for disk scheduling, seems to be a good approximative algorithm. For the SCAN policy, the disk arm follows the physical ordering of the cylinders either from inside to outside or vice versa. Policies other than the simple SCAN policy have been examined in [SCO90].

In general, when a target set is required from magnetic disk, the following combined algorithm is used. First, the cylinders which contain at least one target page are accessed following the SCAN policy. For each of these cylinders, the optimal algorithm is used for reading all target pages from the cylinder.

In order to present an analysis of the schedules produced by the algorithm, the problem is first restricted to the case where all target pages are on the same cylinder. The analysis for the more general problem, when pages are distributed over several cylinders, is presented in section 5.5. A summary of the most important notations used in the remainder of the chapter is given in Table 5.1.

5.4 Analysis of Multi-Page Requests on a Cylinder

In this section, we present an analysis of the optimal algorithm for reading a set of target pages from a cylinder $(C_{i,j})$, $0 \leq i < TC$ and $0 \leq j < PT$. The number of required rotations is determined by the maximum number M of elements in the sets $C_{*,j}$, $0 \leq j < PT$. Without loss of generality, the initial position of the disk arm is assumed to be be in front of column 0, i.e. when the first page is read from column l, $0 \leq l < PT$, the rotational delay corresponds to l page transfers.

Lemma 5.4.1 *Let $\{T_1, \ldots, T_N\}$, $N \leq PC$, be the target set completely stored on a cylinder and let $(T_{i_1}, \ldots, T_{i_N})$ be the corresponding schedule. Furthermore, let $C_{*,0}, \ldots, C_{*,PT-1}$ denote the distribution of the target pages among the columns of the cylinder and let M be defined as $\max_{0 \leq i < PT} |C_{*,i}|$. For the optimal algorithm, the transfer time tt is given (in units of page transfers) as*

$$tt(T_{i_1}, \ldots, T_{i_N}) = 1 + PT * (M - 1) + \max\{j - i \mid |C_{*,j}| = |C_{*,i}| = M, 0 \leq i, j < PT\}$$

and the rotational delay rd is given (in units of page transfers) as

$$rd(T_{i_1}, \ldots, T_{i_N}) = \min\{j \mid \ |C_{*,j}| = M, 0 \leq j < PT\}$$

Proof: The proof follows immediately from the description of the algorithm given in the previous section. □

In the following, cost formulas are derived for the expected transfer time tt_X and the expected rotational delay rd_X. The analysis is made under the basic assumption that target pages are uniformly distributed among the pages of the cylinder. Thus, the formulas only vary in N, the number of target pages. In accordance with Lemma 5.4.1, it is sufficient to compute the expected value of M and the expected value of the position of the last column where the maximum is adopted. The analysis is therefore structured into two parts.

First, the probability $P(N, PT, m)$ is computed that none of the sets $C_{*,j}$, $0 \leq j < PT$, contains more than m of the N target pages which are randomly selected from the PC pages. This probability allows the computation of the expected value of M using the following formula:

$$P(N, PT, 1) + \sum_{i=2}^{\min(N,TC)} ((P(N, PT, i) - P(N, PT, i-1)) * i$$

The formula can be simplified to

$$1 + \sum_{i=1}^{\min(N,TC)} (1 - P(N, PT, i)) \tag{5.1}$$

Second, we present a formula for computing the probability $Q(N, x, M)$ that the maximum M, $1 \leq M \leq TC$, occurs x times, $1 \leq x \leq PT$, assuming that N target pages are selected from the cylinder. Since every column can adopt the maximum with the same probability, the expected position of the last column with M target pages can easily be computed by using elementary probability theory.

In order to compute the probabilities $P(N, PT, m)$ and $Q(N, x, m)$ accurately, two recurrence relations are given in the following subsections. The expected rotational delay and the expected transfer time are then derived from the recurrence relations.

5.4.1 Recurrence Relation for Computing Probability P

Let $(C_{i,j})$ be a cylinder consisting of PT columns and TC tracks. Furthermore, let $F(n, p, m)$ be the number of ways in which n target pages, $1 \leq n \leq PC$, can be selected from p columns, $1 \leq p \leq PT$, of the cylinder such that at most m target pages, $1 \leq m \leq TC$, are taken from each of the p columns. It follows that the probability that none of the p columns contains more than m target pages is given as

$$P(n, p, m) = \frac{F(n, p, m)}{\binom{PC}{n}} \tag{5.2}$$

Suppose that n target pages, $n < p * TC$, have already been randomly selected from the p columns of the cylinder and each of the p columns delivers at most m target pages. Let the next target page be randomly selected from the remaining $p * TC - n$ pages. The term $R(n+1, p, m)$ denotes the conditional probability that the $(n+1)$-st target page is not selected from a column with $TC - m$ pages. Then $R(n + 1, p, m)$ can be expressed as

$$R(n + 1, p, m) = \frac{P(n + 1, p, m)}{P(n, p, m)} \tag{5.3}$$

The $(n+1)$-st target page will produce a column that has delivered more than m target pages if and only if the target page is taken from a column with $TC - m$ remaining pages. The number of possibilities that a certain column contains $T - m$ target pages is given as

$$\binom{TC}{m} * F(n - m, p - 1, m)$$

Furthermore, the probability that this column receives the $(n+1)$-st target page is $\frac{TC-m}{p*TC-n}$. There are p columns and hence, the probability of the $(n+1)$-st target page taken from a column with $TC-m$ remaining pages can be expressed as

$$1 - R(n+1,p,m) = p * \frac{TC-m}{p*TC-n} * \frac{\binom{TC}{m} * F(n-m,p-1,m)}{F(n,p,m)} \tag{5.4}$$

By combining equations 5.2, 5.3 and 5.4, we obtain the following recurrence relation

$$P(n+1,p,m) = P(n,p,m) - \frac{p*(TC-m)}{p*TC-n} * \frac{\binom{TC}{m}\binom{(p-1)TC}{n-m}}{\binom{p*TC}{n}} P(n-m,p-1,m) \tag{5.5}$$

The recurrence relation can be simplified by using the following equality:

$$\frac{TC-m}{p*TC-n} * \frac{\binom{TC}{m}}{\binom{p*TC}{n}} = \frac{\binom{TC}{m+1}}{\binom{p*TC}{n+1}}$$

The recurrence relation is then given as

$$P(n+1,p,m) = P(n,p,m) - p * \frac{\binom{TC}{m+1}\binom{(p-1)TC}{n-m}}{\binom{p*TC}{n+1}} P(n-m,p-1,m) \tag{5.6}$$

In order to compute $P(N,PT,m)$ we need to calculate $P(n,p,m)$ for $n = m, 2, \ldots, p*TC$ and $p = 1, 2, \ldots, PT$. The recursive relation is initialized by using the formula $P(n,p,m) = 1$ for $n \leq m$. Although the above recurrence relation looks complicated at first glance, the computation can be organized so that the evaluation of the next value of $P(n,p,m)$ requires only a few arithmetic operations. In particular, the product of the three binomial coefficients can also be computed in a recursive fashion.

To the best of the author's knowledge the computational problem of computing the probability P has not been solved so far. A similar problem has only been addressed in [Ram87] where the overflow problem of hashing has been examined.

5.4.2 Recurrence Relation for Computing Probability Q

Let $< n, x, m >$ be a three-tuple that denotes the state of the selection process after n target pages, $n < PC$, are read from the cylinder. The parameter m, $m \leq TC$, refers to the maximum number of target pages in a column of the cylinder after reading n pages, and x, $x \leq PT$, denotes the number of columns where m target pages can be found. Let $Q(n,x,m)$ be the probability of obtaining the state $< n, x, m >$ after distributing n target pages starting

from state $< 1, 1, 1 >$. Thus, $Q(1, 1, 1) = 1$ and $Q(1, x, m) = 0$ for $m > 1$ and $x > 1$. Now, let us consider the situation after the $(n + 1)$-st target page is randomly selected from the remaining $PC - n$ pages.

First, the case is discussed for obtaining a transition state $< n + 1, x, m >$ with $x = 1$. There are two possible situations. In the first one the $(n+1)$-st page is selected from a column where the (current) maximum number of target pages can be found, i.e. there are $TC - m$ remaining pages in the column. Hence, the probability of this event is $\frac{TC-m}{PC-n}$. Then, the value of m is incremented by 1 and only a single column adopts the maximum. In the second situation, $x = 1$ is already true before taking the $(n+1)$-st target page from the cylinder. If the $(n+1)$-st target page is selected from a column with more than $TC-(m-1)$ remaining pages, the value of m remains unchanged. Thus, there is still exactly one column with m target pages which are already transferred into main memory. Note that $R(n-m, x-1, m-1)$ refers to the probability that the $(n+1)$-st target page is in a column with more than $TC-(m-1)$ remaining pages. Apart from these situations, there is no way to obtain a transition state with $x = 1$. For $n \geq 1$ and $x = 1$, we obtain the following recurrence relation:

$$\begin{aligned} Q(n+1, 1, m) &= \sum_{j=1}^{PT} j * \frac{TC - m}{PC - n} * Q(n, j, m) + \\ &\quad (1 - \frac{TC - m}{PC - n}) * Q(n, 1, m) * R(n - m, PT - 1, m - 1) \end{aligned} \tag{5.7}$$

Second, let us consider a transition state $< n+1, x, m >$ for $x > 1$. This state can only be realized for the following two cases. In the first case, the original state has been $< n, x, m >$ and the $(n+1)$-st target page is selected from a column with more than $TC-(m-1)$ pages. In the second case, the original state has been $< n, x-1, m >$ and the $(n+1)$-st target page is taken from a cylinder with $TC - (m - 1)$ target pages. The corresponding probability is given as $(1 - R(N + 1 - (x - 1) * m, PT - (x - 1), m - 1))$. Then, the recurrence relation can be computed for $x > 1$ as

$$\begin{aligned} Q(n+1, x, m) &= (1 - x * \frac{TC - m}{PC - n}) * R(n + 1 - x * m, PT - x, m - 1) * Q(n, x, m) + \\ &\quad (1 - (x - 1) * \frac{TC - m}{PC - m}) * \\ &\quad (1 - R(n + 1 - (x - 1) * m, PT - (x - 1), m - 1)) * Q(n, x - 1, m) \end{aligned} \tag{5.8}$$

The probability that for a request of N pages the maximum occurs x times is then given as

$$\sum_{m=1}^{N} Q(N, x, m) \tag{5.9}$$

5.4.3 Expected Rotational Delay

In the previous subsection, the probabilities P and Q are derived as recurrence relations. In this subsection, probability Q is used in computing the expected rotational delay.

If the response set only consists of a single page, the expected rotational delay is half a revolution ($= PT/2$). The same result holds when the maximum number of target pages is adopted only once. In general, the expected rotational delay refers to the expected distance from the beginning of a track to the *first* cylinder with the maximum number of target pages. In order to obtain the expected distance, formula 5.9 is used for computing the probability that the maximum occurs i times, $1 \leq i \leq PT$. The property that the probability for realizing the maximum is the same for each of the columns can then be exploited.

Now, let us suppose that the maximum occurs x times, $1 \leq x \leq PT$. This corresponds to randomly selecting x out of PT columns. The distance to the *first* column can be computed by using elementary probability theory. There are

$$\binom{PT}{x}$$

possibilities to select x columns. Under the assumption that the j-th column, $j \geq x$, is the first one, there are

$$\binom{PT-j}{x-1}$$

possibilities to select the remaining $(x-1)$ columns such that each of them is behind of the j-th column. Hence, the expected distance is given as

$$\sum_{j=1}^{PT-(x-1)} \frac{\binom{PT-j}{x-1}}{\binom{PT}{x}} * j \tag{5.10}$$

and consequently, the expected rotational delay is given as

$$rd_X(N) = \sum_{x=1}^{PT} \{ \sum_{m=1}^{N} Q(N, x, m) \} * \{ \sum_{j=1}^{PT-(x-1)} \frac{\binom{PT-j}{x-1}}{\binom{PT}{x}} * j \} - 1 \tag{5.11}$$

5.4.4 Expected Transfer Time

The transfer of pages from the disk starts when the first column, say $C_{*,f}$, $0 \leq f < PT$, with M target pages is under the disk arm. Then, $M-1$ disk revolutions are required for reading all pages from column $C_{*,f}$. If all other columns contain fewer target pages than column $C_{*,f}$, the transfer is completed. Otherwise, there is a column, say $C_{*,l}$, $f < l < PT$, which refers to the last column with M target pages. Then the transfer of pages continues until the disk arm has passed over the column $C_{*,l}$. For a multi-page request of N pages, the expected cost for the complete revolutions can easily be derived from formula 5.1. We yield

$$PT * \sum_{j=1}^{\min(N,TC)} (1 - P(N, PT, j))$$

Assume that the maximum is realized x times, $1 \leq x \leq PT$. Then, the expected distance to the first column with M target pages is already given by formula 5.10. In a similar way, the expected distance to the last column with M target pages can be computed as

$$\sum_{j=x}^{PT} \frac{\binom{j-1}{x-1}}{\binom{PT}{x}} * j = PT - \sum_{j=1}^{PT-(x-1)} \frac{\binom{PT-j}{x-1}}{\binom{PT}{x}} * j$$

Overall, the following formula is obtained for the expected transfer time:

$$\begin{aligned} tt_X(N,p) &= 1 + PT * \sum_{j=1}^{\min(N,TC)} (1 - P(N, PT, j)) \\ &\quad + \sum_{x=2}^{PT} \{ \sum_{m=1}^{N} Q(N,x,m) \} * (PT - 2 * \sum_{j=1}^{PT-(x-1)} \frac{\binom{PT-j}{x-1}}{\binom{PT}{x}} * j) \end{aligned} \tag{5.12}$$

5.4.5 Discussion

In this subsection, the cost is discussed for reading N target pages from one cylinder. In Figure 5.2, the expected rotational delay, the transfer cost and the sum of both are plotted as a function of the number of target pages. The cylinder consists of 20 tracks, each of them containing 8 pages ($TC = 20, PT = 8$). The cost is expressed as the expected cost per target page.

For $N = 1$ (i.e. there is only one target page), we obtain the well-known result that the expected rotational delay is half a rotation. As expected, the cost per target page decreases

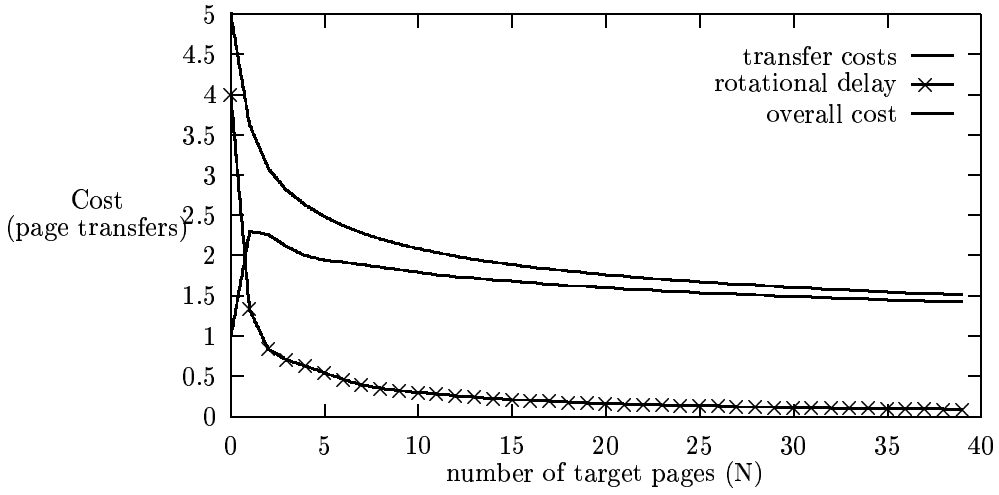

Figure 5.2: Expected rotational delay and expected transfer costs ($TC = 20, PT = 8$)

with an increasing number of target pages. For $N = 10$, the cost per target page is only 2 page transfers. For larger values of N, the cost per target pages approaches the minimum of one page transfer.

When pages are read in random order, the cost per target page would be the same as reading a single target page from the cylinder (5 page transfers in the example). Even if the target pages are already clustered on a single cylinder, the graph demonstrates the importance of reading the target pages in the order given by an optimal schedule.

5.5 A Global Cost Function

In this section, a cost function is presented for reading target pages which are distributed over several cylinders of the disk. Thus, the cost function has to consider seek time as a major component of the total cost.

In a scenario where a query is performed on a file, it is very unlikely that target pages are uniformly distributed over the disk. On the contrary, it is expected that the underlying file system clusters a file on a few cylinders. Accordingly, target pages would be distributed on

a few cylinders. In order to take into account the effect of clustering, the cost function does not only depend on the number of target pages, but also on the cylinders on which the pages of the file are distributed. A cylinder that contains a page of the file is called a *file cylinder* in the following.

In the analysis, we make the simplifying assumptions that the file cylinders are distributed uniformly over the disk and that the target pages are uniformly distributed over the file cylinders. The first assumption is in agreement with the policy on how the fast file system of UNIX [MJLF84] organizes files on the disk when a so-called cylinder group consists of exactly one cylinder. The second assumption can often be observed for queries in a DBMS, e.g. range queries performed on a secondary index. In order to simplify the analysis, we assume that the initial position of the disk arm is at the outermost cylinder. The algorithm for reading the pages visits only those file cylinders where a target page can actually be found in linear order from outside to inside. A file cylinder where a target page is located is called a *target cylinder*.

First, a cost formula is given for the required seek operations. Note that a seek consists of several components. First, the arm is accelerated until it reaches half the seek distance or a constant velocity. In the later case, the arm coasts across some cylinders at its maximum speed. Third, the arm decelerates its speed and stops on top of the desired track. Therefore, the seek time can be well modeled as a square root function for small seeks and as a linear function for large seeks. The *seek time of a multi-page request* is simply the sum of the seek time required to move from one target cylinder to the next target cylinder. The distribution of target cylinders is determined through the number of target pages and the number of file cylinders. The exact distribution is again given as a recurrence relation. The analysis of the seek cost is first based on the exact distribution of target cylinders. Thereafter, an approximate formula is derived for the expected seek cost which avoids the computation of the recurrence relation. This formula produces results which are very close to the ones obtained from the exact formula.

Let Cyl be the number of cylinders on the disk. For a file F, the parameter C_F denotes the number of file cylinders. Consider N target pages randomly distributed over C_F file cylinders. Let $< n, j >$, $n \geq 1, 1 \leq j \leq C_F$, refer to a transition state where j of the C_F

file cylinders are target cylinders. Furthermore, let $S(n, j)$ be the probability of realizing the transition state $< n, j >$. Note that $S(1, 1) = 1$ and $S(1, j) = 0$, $2 \leq j \leq C_F$. For $n > 1$, two situations are distinguished: either the n-th target page is on a cylinder which already contains one of the other $n-1$ target pages, or the n-th target page is on an "empty" cylinder that does not contain any of the other target pages. Then the probability S can be computed for $n > 1$ by using the following recurrence relation:

$$S(n, j) = \frac{(C_F - (j-1)) * PC}{C_F * PC - (n-1)} S(n-1, j-1) + \frac{j * PC - (n-1)}{C_F * PC - (n-1)} S(n-1, j) \qquad (5.13)$$

Note that this recurrence relation is actually a special case of the formulas 5.7 and 5.8.

Let us consider that C_{act} cylinders are target cylinders, i.e. they are hit by at least one of the target pages. Recall that the disk arm is assumed to be on the first cylinder and that target cylinders are uniformly distributed. The first cylinder is a target cylinder with probability C_{act}/Cyl. In general, the probability that the first target cylinder is the k-th cylinder on the disk, $1 \leq k \leq Cyl - C_{act} + 1$, is given as

$$\frac{\binom{Cyl-k}{C_{act}-1}}{\binom{Cyl}{C_{act}}}$$

Assume that the disk requires $seek\_time(i)$ to pass over i cylinders. When the first cylinder is a target cylinder, the expected cost to move from one target cylinder to another is given by $H(Cyl-1, C_{act}-1)$. The function H is computed as

$$H(m, l) = \sum_{k=1}^{m-l+1} \frac{\binom{m-k}{l-1}}{\binom{m-1}{l}} * seek\_time(k)$$

When the first cylinder is not a target cylinder, the expected cost to move from one target cylinder to another is given as $H(Cyl-1, C_{act})$. This event occurs with probability $1 - C_{act}/Cyl$. Because target cylinders are uniformly distributed, the expected distance between two target cylinders is independent of their actual position. Under the assumption of j target cylinders the expected cost for the seek operation is then given as

$$G_{Cyl}(j) = \frac{j}{Cyl} * (j-1) * H(Cyl-1, j-1) + (1 - \frac{j}{Cyl}) * j * H(Cyl-1, j)$$

Finally, the expected seek time st_X can be obtained by combining the formulas in the following way:

$$st_X(N, C_F) = \sum_{j=1}^{C_F} S(N, j) * G_{Cyl}(j) \qquad (5.14)$$

When the target set is distributed over several cylinders, the rotational delay and transfer time of the multi-page request is defined as the sum of the rotational delays and transfer times which occur on the target cylinders, respectively. The expected rotational delay and the expected transfer time can be approximately computed under the assumption that each of the target cylinders receives target pages independently of each other, i.e. the values computed for the one cylinder has no effect on the values of another cylinder. Then, a target cylinder receives j of the N pages with probability

$$\binom{N}{j}(\frac{1}{C_F})^j * (1 - \frac{1}{C_F})^{N-j}$$

The formulas 5.14, 5.12 and 5.11 can be combined for computing the total expected cost of reading N target pages distributed on C_F cylinders. We yield

$$ICost(N, C_F) = st_X(N, C_F) + \sum_{j=1}^{N}\{\binom{N}{j}(\frac{1}{C_F})^j * (1 - \frac{1}{C})^{N-j} * (tt_X(j) + rd_X(j))\} \quad (5.15)$$

5.5.1 An Approximation for the Seek Time

The computation of probability S using the recurrence relation, see formula 5.13, is time consuming when the number of file cylinders and the number of target pages is high. In this subsection, an approximate formula is given for the seek cost which is based on the expected number of target cylinders. The (exact) expected number of target cylinders can be computed by using the Waters-Yao formula [Wat76, Yao77]. The formula was originally proposed for a problem in a different context. In [Yao77] it is shown that the formula of Cardenas [Car75] is a simple and accurate approximation of the expected value if (in our terminology) a cylinder consists of a large number of pages. Since this property is satisfied in our setting, the formula of Cardenas is used in the following. Thus, the expected number of target cylinders is approximately given as

$$x = C_F * (1 - (1 - \frac{1}{C_F})^N)$$

Since x is not an integer, $\lfloor x \rfloor$ and $\lceil x \rceil$ are used for computing a linear combination. The approximate formula is given as

$$st_X(N, C_F) \simeq (x - \lfloor x \rfloor) * \lceil x \rceil * G_{Cyl}(\lceil x \rceil) + (x - \lceil x \rceil) * \lfloor x \rfloor * G_{Cyl}(\lfloor x \rfloor) \quad (5.16)$$

Cyl	840
TC	20
PT	8
page size [KB]	4
average seek time	9
average rotational delay	4

Table 5.2: Specification of the Fujitsu Eagle disk

In [PBD93] the formula $x * seek\_time(Cyl/x)$ has been proposed for approximating the seek cost. This simple approximation also gives accurate results (relative errors of about 2% and less) if the number of target pages is high. However, for a small number of target pages, the relative error of the seek cost is too high. In contrast, formula 5.16 still produces accurate results (with a relative error less than 1%) in such a situation.

5.5.2 Discussion

In order to demonstrate that the cost function 5.15 is indeed accurate, we compared the results of the cost functions with the ones obtained from simulations. The underlying disk parameters correspond to the one of the Fujitsu Eagle disk (see Table 5.2) which is a rather old disk, but is frequently used in various experimental comparisons. Note that the timings of Table 5.2 are given in page transfers. Moreover, the seek time of the Fujitsu Eagle is well approximated by using the following function [SCO90]:

$$seek\_time(x) = \begin{cases} 0 & \text{if } x = 0 \\ 2.3 + 4.35\sqrt{x} & \text{if } x \leq 239 \\ 9 + 0.14(x - 239) & \text{if } x > 239 \end{cases} \tag{5.17}$$

For one of our experiments, the results of the cost functions and the results of the simulations are reported in Table 5.3. In this experiment, the number of target pages is fixed ($N = 40$), whereas the number of file cylinders is varying from one to forty. The cost is decomposed into three components: transfer cost, rotational delay and seek cost. For each of these cost components, we report the results of the cost function, the results of the simulation and additionally, the relative error given in percent. As demonstrated in Table 5.3, the relative errors are less than 1% for each of the cost components.

cylinder	rotational delay			transfer time			seek time		
	model	simu.	rel. error	model	simu.	rel. error	model	simu.	rel. error
1	1.4205	1.4246	0.2869	0.0873	0.0872	0.0831	0.2841	0.2852	0.3707
2	1.6075	1.6077	0.0130	0.1690	0.1677	0.7886	0.4637	0.4633	0.0855
3	1.7218	1.7166	0.3020	0.2467	0.2470	0.1362	0.6130	0.6127	0.0576
4	1.8033	1.8041	0.0465	0.3213	0.3215	0.0700	0.7476	0.7479	0.0363
5	1.8642	1.8652	0.0528	0.3943	0.3923	0.5094	0.8730	0.8740	0.1063
6	1.9124	1.9136	0.0671	0.4646	0.4639	0.1544	0.9917	0.9924	0.0720
7	1.9522	1.9556	0.1705	0.5317	0.5323	0.1012	1.1047	1.1052	0.0449
8	1.9855	1.9917	0.3078	0.5966	0.5985	0.3283	1.2124	1.2128	0.0325
9	2.0127	2.0107	0.1006	0.6604	0.6598	0.0887	1.3151	1.3150	0.0025
10	2.0340	2.0383	0.2150	0.7241	0.7229	0.1666	1.4127	1.4133	0.0415
11	2.0497	2.0513	0.0765	0.7881	0.7885	0.0428	1.5054	1.5050	0.0251
12	2.0605	2.0628	0.1117	0.8523	0.8473	0.5903	1.5932	1.5944	0.0784
13	2.0669	2.0692	0.1138	0.9166	0.9160	0.0653	1.6764	1.6775	0.0635
14	2.0695	2.0703	0.0394	0.9806	0.9797	0.0901	1.7551	1.7575	0.1330
15	2.0689	2.0730	0.1978	1.0441	1.0450	0.0861	1.8296	1.8325	0.1590
16	2.0656	2.0664	0.0393	1.1068	1.1078	0.0911	1.9001	1.9028	0.1429
17	2.0602	2.0627	0.1249	1.1685	1.1696	0.0925	1.9668	1.9709	0.2077
18	2.0529	2.0580	0.2492	1.2289	1.2327	0.3094	2.0299	2.0320	0.1028
19	2.0442	2.0451	0.0461	1.2880	1.2888	0.0626	2.0897	2.0927	0.1402
20	2.0343	2.0329	0.0707	1.3456	1.3480	0.1772	2.1464	2.1463	0.0055
21	2.0235	2.0251	0.0788	1.4017	1.4039	0.1571	2.2002	2.2042	0.1784
22	2.0121	2.0118	0.0143	1.4562	1.4574	0.0832	2.2513	2.2554	0.1803
23	2.0000	1.9995	0.0271	1.5091	1.5098	0.0458	2.2998	2.3011	0.0529
24	1.9876	1.9911	0.1748	1.5604	1.5582	0.1420	2.3460	2.3482	0.0945
25	1.9750	1.9754	0.0231	1.6101	1.6167	0.4071	2.3899	2.3962	0.2612
26	1.9621	1.9634	0.0635	1.6582	1.6606	0.1454	2.4318	2.4366	0.1978
27	1.9492	1.9512	0.1054	1.7048	1.7071	0.1339	2.4717	2.4751	0.1367
28	1.9362	1.9352	0.0497	1.7499	1.7538	0.2268	2.5098	2.5137	0.1555
29	1.9233	1.9241	0.0429	1.7935	1.7932	0.0179	2.5462	2.5484	0.0882
30	1.9104	1.9104	0.0012	1.8357	1.8372	0.0797	2.5810	2.5852	0.1631
31	1.8977	1.8983	0.0335	1.8765	1.8784	0.1007	2.6143	2.6180	0.1430
32	1.8850	1.8822	0.1526	1.9160	1.9215	0.2851	2.6461	2.6529	0.2542
33	1.8726	1.8718	0.0439	1.9543	1.9510	0.1681	2.6767	2.6805	0.1424
34	1.8603	1.8566	0.1974	1.9913	1.9941	0.1438	2.7060	2.7087	0.1020
35	1.8482	1.8475	0.0405	2.0271	2.0311	0.1979	2.7341	2.7393	0.1906
36	1.8363	1.8359	0.0259	2.0618	2.0625	0.0376	2.7611	2.7632	0.0748
37	1.8247	1.8216	0.1666	2.0954	2.0966	0.0583	2.7871	2.7907	0.1313
38	1.8132	1.8084	0.2658	2.1279	2.1320	0.1916	2.8120	2.8193	0.2571
39	1.8020	1.8006	0.0772	2.1594	2.1601	0.0333	2.8361	2.8407	0.1611
40	1.7910	1.7916	0.0326	2.1900	2.1866	0.1555	2.8592	2.8613	0.0730

Table 5.3: A comparison of results obtained from simulations and the cost function ($N = 40$)

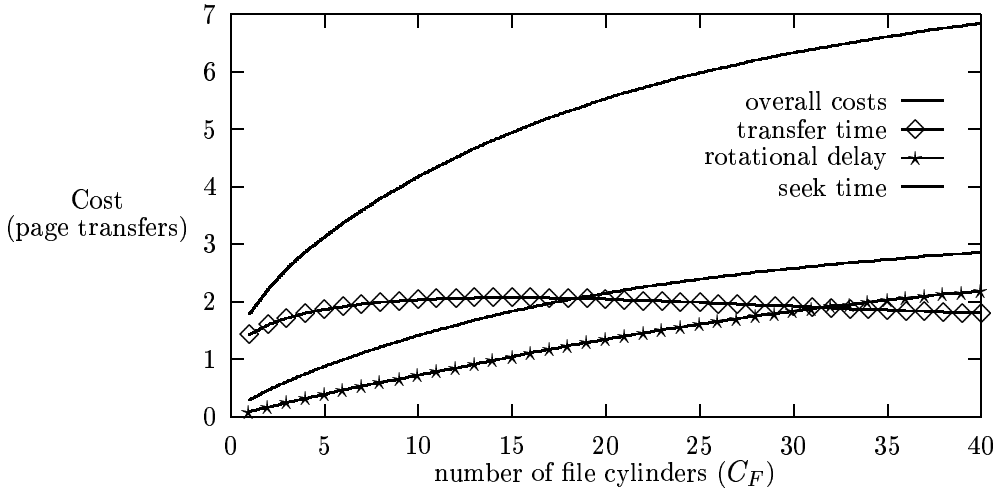

Figure 5.3: Cost function (varying in cylinder, constant number of 40 pages)

In Figure 5.3, the expected cost is plotted for the three cost components as a function of the number of file cylinders. Additionally, a fourth curve shows the total cost per target page given as the sum of the three cost components. The cost is measured in units of page transfers. As expected, the total cost increases with an increasing number of file cylinders. Best performance is obtained when the file is kept only on a single cylinder. Reading a target page in this case only requires less than two page transfers. In general, it is more likely that files will consist of a large number of pages and therefore the number of file cylinders has to be greater than one. For example, let us consider that target pages are distributed on 5 cylinders. Then the approach of reading all target pages in a single multi-page request requires only three page transfers (per target page).

Let us compare these results to the ones obtained when target pages are read using a random schedule, i.e the target pages are read in a random sequence without considering their physical position on disk. Then, when the disk arm is already on the desired cylinder, 5 page transfers will be required on the average for reading a page from the cylinder. Otherwise, the disk arm has to move to the desired cylinder first. The average seek time of the Fujitsu Eagle disk is roughly 9 page transfers. The total time is then 14 page transfers for reading

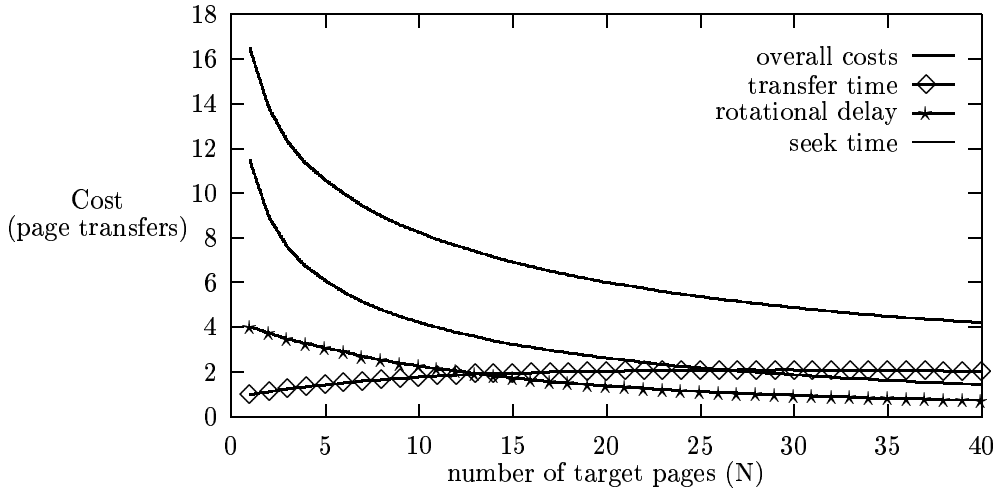

Figure 5.4: Cost function (varying number of pages, constant number of 10 cylinders)

a page. The probability is approximately 1/5 that a target page is read from the same file cylinder as the previous target page. Thus, the expected cost for reading a page is about 12 page transfers. In comparison to reading target pages using the proposed schedule, a random schedule is less efficient by a factor of four. However, the advantage of using efficient schedules decreases with an increasing number of file cylinders. For example, when 40 target pages are randomly distributed over 40 file cylinders, the cost of reading a target page corresponds to 7 page transfers, i.e. it is still half the cost of reading pages randomly.

The total cost depends on its three components as follows. Transfer time is the dominant component for $C_F \leq 18$. For $C_F > 18$, the seek cost is higher than the transfer cost. For a large number of file cylinders, transfer cost can even be lower than the rotational delay.

In Figure 5.4, the cost is illustrated for a fixed number of file cylinders ($C_F = 10$) and a varying number of target pages. As long as the number of target pages is low, the performance gain of reading target pages using the proposed schedules is fairly low in comparison to random schedules. However, the total cost per target page constantly decreases with an increasing number of target pages.

5.6 Multiple Queries

In the previous sections, the performance of multi-page requests is examined under the assumption that the disk belongs exclusively to one query. This is typically true for a low-loaded or a batch-oriented system. In a heavily loaded system where several queries are concurrently processing, it is not acceptable that the disk arm belongs to a single query from beginning to end. Otherwise, a huge request may defer the execution of small requests (which may require only a few pages) such that their response time becomes unacceptably high. However, the primary goal in a heavily loaded system is not to improve the response time of individual queries, but to improve the throughput. The throughput is defined for a given mix of queries and for a given arrival rate of queries by the number of completed queries per second. The reciprocal value of the throughput gives the average response time of a query. The question arises how the policy of multi-page requests influences the throughput of the system? This question is discussed under the assumption that the response time of a query is determined by its I/O time.

Our approach for processing queries concurrently using several multi-page requests is as follows. A query sends multi-page requests to the disk one by one. Each of these multi-page requests contains all target pages of the query which belong to the same cylinder of the disk. Since multi-page requests are processed without interfering of other requests, the disk gives services to a query as long as target pages are still unprocessed on the cylinder where the disk arm is currently positioned. Several multi-page requests of different queries are waiting in front of the disk in a request queue to receive service from the disk. Any of the well-known disk scheduling policies [Dei90] can be used for organizing the request queue. This approach to query processing has basically two advantages. First, it prevents a multi-page request from occupying the disk for a very long time period. Second, the size of the buffer required for a multi-page request is limited by the capacity of a cylinder.

In the following, we report the results of a preliminary performance comparison of multi-page requests and ordinary (single) page requests when multiple queries are processed concurrently. All results are obtained from simulations which are based on our disk model and on the corresponding cost function.

The simulation runs on a Sequent Symmetry, a shared-memory multi-processor system.

Figure 5.5: The implementation of the simulation

The system runs DYNIX, a version of the UNIX 4.3BSD operating system. The simulation is implemented in C using the *μSystem* [BS90], a library which provides simple but effective mechanisms to deal with light-weighted processes termed *threads* in the following. In comparison to UNIX processes, the overhead is very low for creating a thread, absorbing a thread, and switching execution from one thread to another.

The implementation of the simulation is illustrated in Figure 5.5. A thread *CreateQ* creates the queries with respect to a given query profile. In general, several queries are running concurrently as independent threads at a time. Each of the active queries sends page requests to the disk one by one. We assume that the query is blocked while the disk satisfies one of its requests. Thereafter, when the request is completed, the next request of the query is immediately sent to the disk. If all requests of a query are satisfied, the execution of the query is finished. The disk itself is also implemented as a thread which is running on a separate processor. Attached to the disk is a queue of waiting requests. The circular SCAN (C-SCAN) [Dei90] scheduling strategy is used for selecting the next request which receives service from the disk. In contrast to the ordinary SCAN policy, the C-SCAN policy moves unidirectionally across the disk surface toward the inner cylinder. When there are no

more requests for service ahead of the arm, it jumps back to service the request nearest the outermost cylinder and proceeds inward again. In addition to the queuing delay, the cost of a request consists of two components: the seek time and the cylinder time (i.e. the time spent on the cylinder). In our simulation, the disk simulates a Fujitsu Eagle and therefore, the seek time can be computed using the formula 5.17. The cylinder time is obtained as the sum of formula 5.12 and formula 5.11 derived in section 5.4. For a request of one target page, the cylinder time is then roughly 10.3 ms.

A *query profile* is characterized by one or several query types. Each query type is described by the following parameters: the number of target pages (N) and the number of (file) cylinders (C_F). We assume that the target pages are uniformly distributed over the file cylinders and that the file cylinders are uniformly distributed among the cylinders of the disk. For each query type, the probability of selecting a query of this type has to be specified. In addition to the query types, a query profile is characterized by the arrival rate of the queries. In the following, we assume that the arrivals of queries follow an exponential distribution where λ denotes the average number of queries which arrive in one second. The assumption of an exponential distribution is rather common for this kind of simulations and therefore, it is also used in our experiments. Overall, a query profile QPF can be described by the following list of parameters

$$QPF = (\lambda, pb_1, QT_1(N_1, C_1), \ldots, pb_k, QT_k(N_k, C_k))$$

where λ denotes the average number of queries per second and QT_i is the i-th query type which delivers queries with probability pb_i, $1 \leq i \leq k$. The parameters N_i and C_i denote the number of target pages and the number of file cylinders of query type QT_i.

In our first set of experiments, we investigated query profiles

$$QPF_1(\lambda) = (\lambda, 1.0, QT(20, 5))$$

where parameter λ is varying in the range between 0.5 and 10. In Figure 5.6 the average response time (expressed in ms) is depicted. The one graph refers to the policy when target pages are read using ordinary read requests, whereas the other graph depicts the results when multi-page requests are used. For $\lambda = 1$, 20 target pages on the average are required to be read from disk. For $\lambda = 0.5$, the queuing delay does not have much influence on the response

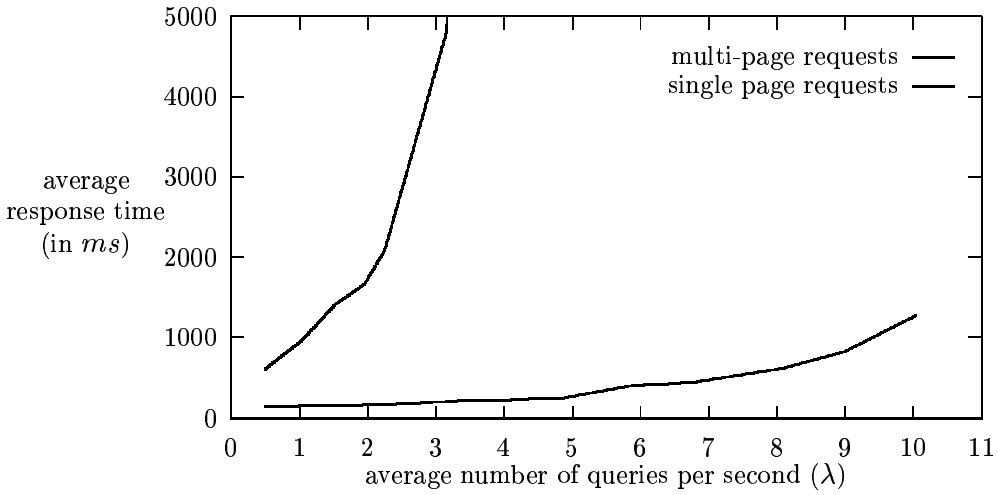

Figure 5.6: Results of query profiles $QPF_1(\lambda)$ varying in λ

time of the queries and therefore, these results correspond to the ones obtained when queries are performed one by one. As depicted in Figure 5.6, when queries are performed using single-page requests the average response time per query dramatically increases for $\lambda > 3$, whereas multi-page requests still offer low response time of the queries. Only for $\lambda > 10$, response times are not acceptable anymore for multi-page requests.

In the second set of experiments, query profiles

$$QPF_2(\lambda) = (\lambda, 0.5, QT(1,1), 0.4, QT(20,5), 0.1, QT(100,10))$$

are considered, $0.5 \leq \lambda \leq 14$, each of them containing three different query types. A query that requires only one page is assumed to occur with probability 50%, whereas the probability of a query that requires 100 target pages is only 10%. The average response time of queries of query profile QPF_2 is depicted in Figure 5.7. The graphs show almost the same characteristics as the ones observed for query profile QPF_1.

One serious problem of using multi-page requests might be that the multi-page requests of data-intensive queries occupy the disk for a very long time period such that the response time of small queries which require only one page (or a few pages) increases substantially.

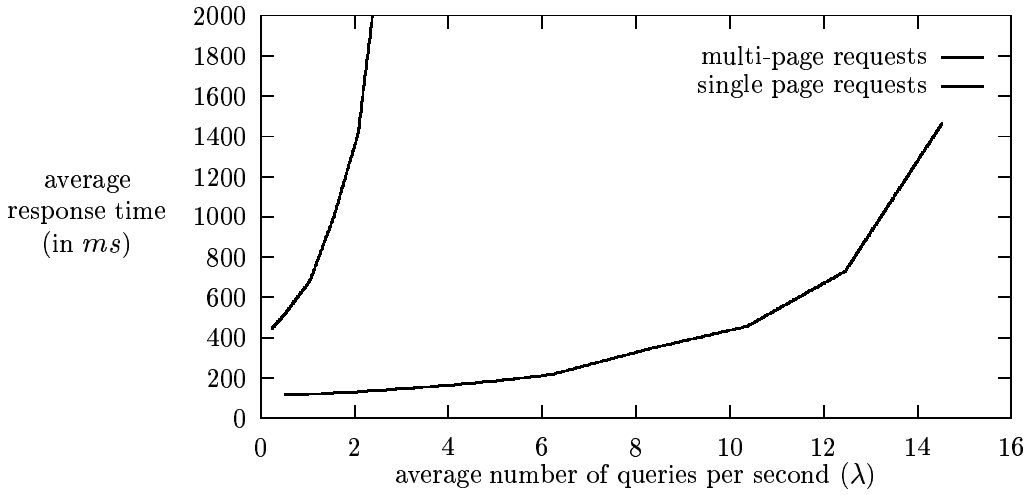

Figure 5.7: Results of query profiles $QPF_2(\lambda)$ varying in λ

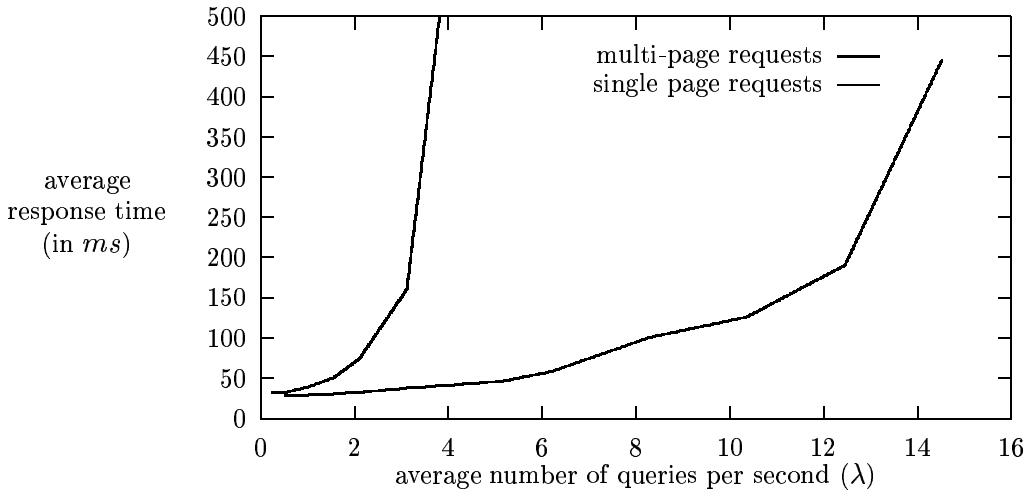

Figure 5.8: The average response time of queries of type $QT(1,1)$

An interesting question is therefore how the average response times of small queries develops when all queries are performed using multi-page requests. In order to answer this question for our query profiles $QPF_2(\lambda)$, the average response time of the queries of type $QT(1,1)$ is plotted in Figure 5.8. The response time of a single query of type $QT(1,1)$ obviously cannot take advantage from multi-page request. Therefore, both graphs show similar results for a low-loaded system (i.e. small λ). On a heavily loaded system, however, the average response time of these queries is considerably lower when multi-page requests are used in comparison to using single page requests. Overall, we conclude that multi-page requests does not only reduce the response time of data-intensive queries on a loaded system, but also of those queries which require only a few pages from disk.

5.7 Summary

In this chapter, we examined schedules for efficiently reading a set of pages from a file scattered on some cylinders of a magnetic disk. First, a simple algorithm is introduced for computing schedules. The main contribution of the chapter is its analysis which is completely based on an analytical model. As a result, we derived a cost function for the expected cost of reading a set of disk pages. The cost function varies in the number of required pages and in the number of cylinders where pages of the file are kept. The cost function illustrates the dependency between cost, clustering of data, and scheduling policies. In particular, we found that although the required pages are clustered on a single cylinder, the cost for reading the pages can be reduced substantially when an efficient read schedule is used. Moreover, the cost function also shows the relationship between its three components of seek time, rotational delay and transfer time.

In order to obtain an analysis, several simplifying assumptions have to be made with respect to the distribution of files, the distribution of target pages and the architecture of the disk. A file is assumed to be distributed over some cylinders which are randomly selected from the disk. This assumption is in agreement with the policy of the fast UNIX file system [MJLF84] (when a so-called cylinder group consists of a single cylinder). Moreover, target pages are randomly selected from the pages of the file. This assumption is valid for many queries in a DBMS. In our disk model, a cylinder is modeled as a two-dimensional array of

pages. In particular, head switch time and track skewing is not considered in the model so far. The additional costs caused by head switches and track skewing are not expected to dominate the overall cost of a multi-page request, but they may have some impact on it. This issue of head switch time is discussed in full detail in the following chapter.

Chapter 6

Disk Models that Consider Head Switch Time

One of the most important assumptions of our disk models has been that head switch time can be neglected. This assumption, however, is not in agreement with current disk technology. In contrast with the improvements in track density, the ratio of head switch time to transfer time is unfortunately increasing for today's disks. In particular, high head switch times occur when pages are written. A read request is not influenced so much by the head switch time, since the disk arm need not be positioned with the same accuracy required for write requests.

In the first section of this chapter, we present algorithms for reading a set of disk pages, when head switch time is taken into account. Two of the algorithms produce schedules which are close to optimal. Results of an experimental comparison show that there is almost no difference in the expected cost of schedules produced by all of these algorithms. In the second and third sections, practical approaches are pursued to derive approximate functions for estimating the cost of the schedules. The approach in section 2 makes use of the cost functions originally proposed for the idealized disk model that does not consider head switch time. In contrast to that, a completely new approach is presented in section 3. In particular, the cost functions are very easy to compute. In addition to head switches, the underlying disk model considers track skewing and other properties of a disk. Section 4 concludes the chapter.

6.1 Algorithms

In this section, we make the simplifying assumptions that the starting positions of the tracks are vertically aligned in a cylinder and that a track consists of PT pages, where PT is an integer. Hence, a cylinder can again be considered as a two-dimensional array $C_{i,j}$ of pages, $0 \leq i < TC$ and $0 \leq j < PT$.

Before going into more details, let us first introduce an important notation used throughout the chapter. For two positive integers, say n and m, the expression $n \ \% \ m$ refers to $\lfloor n/m \rfloor$.

The following discussion is at first restricted to the case that head switch time is one page transfer. Then, after a page $C_{i,j}$ is read from disk, any page from the $(j+2)\%PT$-th column can be read with a rotational latency of 1 page transfer. Furthermore, page $C_{i,(j+1)\%PT}$ can read without any rotational latency, but a page $C_{k,(j+1)\%PT}$, $k \in \{0, 1, \ldots, i-1, i+1, \ldots, TC-1\}$, requires a rotational latency of PT page transfers.

A *track cluster of size* k is a contiguous sequence $C_{i,j}, C_{i,j+1}, \ldots, C_{i,j+k-1}$, $0 \leq j < TC$ and $k \leq TC - j$, of target pages on a track such that one of the following conditions is fulfilled:

(1) $j = 0$ and $k = TC$

(2) pages $C_{i,(j-1)\%TC}$ and $C_{i,(j+k)\%TC}$ are not target pages.

When condition (1) is fulfilled, a track cluster is called *complete*. Otherwise, condition 2 is fulfilled and a track cluster is called *partial*. Let us recall that pages that are not target pages are called empty pages.

In the following subsections, three simple algorithms are considered for computing efficient read schedules. The last two of these algorithms produce read schedules whose cost is close to optimum. In the fourth subsection, we present some results from an experimental comparison of these algorithms.

6.1.1 Elevator Algorithm

The *Elevator* algorithm is a very simple algorithm for reading a set of pages from a cylinder. The algorithm follows.

Algorithm **Elevator**
Initial state: the disk arm is in front of column 0, $j_{old} = PT - 1$ and $i_{old} = 0$.
BEGIN
WHILE (there is an unprocessed target page) DO

(1) Determine the next column j with an unprocessed target page, $j = (j_{old} + 1)\%PT, (j_{old} + 2)\%PT, \ldots$.

(2) Compute the smallest i, $i \geq i_{old}$, such that $C_{i,j}$ is an unprocessed target page. If such a page does not exist, compute smallest i, $i \geq 0$, such that $C_{i,j}$ is an unprocessed target page.

(3) Read target pages $C_{i,j}, C_{i,(j+1)\%PT}, \ldots$ until a page, say $C_{i,k}$, $0 \leq k < PT$, is detected that is either an empty page or a target page that has been already processed.

(4) Set $j_{old} = k$ and $i_{old} = i$.

END;
END Elevator;

Without loss of generality, it is assumed in the algorithm *Elevator* that the initial position of the disk arm is in front of column 0. The loop is repeated as long as a target page is unprocessed. In step (1) of the loop the next column is computed which contains an unprocessed target page. Thereafter, in step (2) one of the unprocessed target pages is determined in an elevator-based fashion. In step (3), the pages are read from the corresponding track until an empty page is found or a target page is found that has already been processed. Finally, indices are updated in the last step of the loop.

Let us illustrate the algorithm using the example given in Figure 6.1. In our example, a cylinder is assumed to consist of 8 columns and 4 tracks. The *Elevator* algorithm first reads the target page A. Next, the algorithm examines column 2. There are two possible pages for reading, and the algorithm decides to read page C (because it is on the lowest track above track 0). The remaining target pages are read in the following order: E, G, B, F, D. Overall,

Figure 6.1: An example of 7 target pages A,...,G distributed on a cylinder ($PT = 8$ and $TC = 4$)

the cost of the read schedule is 19 page transfers.

Note that the *Elevator* algorithm is not restricted to disks that fulfill our assumptions. It can also be used for disks with an arbitrary head switch time. Moreover, the algorithm can easily be generalized to disks with track skewing of an arbitrary number of sectors.

6.1.2 Shortest-Latency-Time-First Algorithm

The basic idea of the next approach is to read track clusters one by one. When a track cluster is completely processed, the next track cluster is determined according to the policy *shortest-latency-time-first (SLTF)*. This policy selects the track cluster whose starting position comes under the disk arm first. Due to its simplicity, we abstain from giving the algorithm in full detail. Instead, let us again discuss the example given in Figure 6.1 under the assumption that the disk arm is in front of column 0. After the algorithm reads page A, the next track cluster is the one that only consists of page D. Note that the *Elevator* algorithm performs differently. The remaining target pages are read in the following order: E, B, C, F, G. The cost of the schedule is 13, i.e. it is less than the cost of the schedule computed by the *Elevator* algorithm.

In [SF73], the SLTF policy has been shown to produce close-to-optimal schedules (*SLTF schedules*) when head switch time is of no concern. In the following, we show that SLTF schedules are also close to optimal for a disk that performs a head switch at the expense of one page transfer. The basic idea of the proof is to transform a multi-page request performed

Figure 6.2: The transformed target set of the example

on a disk that considers head switch time into an equivalent multi-page request performed on a disk that neglects head switch time.

For a given target set, let S be a schedule, not necessarily an SLTF schedule. Schedule S can be transformed into an equivalent schedule S_T on a disk without head switch time such that the cost for both schedules is the same. Schedule S_T is created as follows. Let (T_1, T_2) be a pair of adjacent target pages in schedule S. Then these pages are also adjacent in schedule S_T, if T_1 is the predecessor page of T_2 on the same track. Otherwise, a new page P is inserted in schedule S_T between pages T_1 and T_2. More precisely, page P is in the column left of the column where page T_2 is stored. Such a page is also called an *artificial target page*. It follows that for a given target set, the cost of schedule S is always higher than the minimum cost for reading the target pages and the artificial target pages from a disk without head switch time. For our example given in Figure 6.1, the transformed situation is depicted in Figure 6.2. The target pages with label P refer to the artificial ones.

Now, let $S = (CL_1, \ldots, CL_k)$ be an SLTF schedule consisting of k track clusters and let S' be an arbitrary schedule. The transformed schedules of S and S' are called S_T and S'_T, respectively. Then, S_T is given as $(CL_1, P_2, CL_2, \ldots, P_k, CL_k)$, where P_i, $2 \leq i \leq k$ are artificial target pages. It follows that there are artificial target pages P'_i in schedule S'_T such that page P_i and page P'_i are in the same column, $2 \leq i \leq k$. Thus, the cost of the transformed SLTF schedule S_T is not higher than the cost of any other transformed schedule. This observation is used in the proof of the following theorem.

Theorem 6.1.1 *For a given target set, let S be an SLTF schedule on a cylinder with a head switch time of 1 page transfer and let S_T be the transformed schedule. Then, the cost of schedule S_T is not more than PT higher than the cost of an optimal schedule.*

Proof:

For $k > 0$, let $S_T = (CL_1, P_2, CL_2, \ldots, P_k, CL_k)$ be the transformed schedule of S. Without loss of generality, the disk arm is assumed to be positioned in front of column 0 initially. Let M denote the maximum number of target pages (including the artificial ones) that are in a column. Then, in order to read the required pages, the cost of the SLTF schedule would be at least $(M-1) * PT + j_0 + 1$ where j_0 refers to the last column with maximum number of target pages. Moreover, $(M-1) * PT + j_0 + 1$ is also a lower bound for the cost of any other transformed schedule.

Next, we show that the column, say j, which contains page P_k contains M target pages. If this is not true, the disk arm would pass over the column j once without reading any page. For the original schedule S, this would cause that no target page is read from column j or from column $(j+1)\%PT$. Since schedule S is an SLTF schedule, reading track cluster CL_k would have been started prior to the the last rotation. Hence, CL_k is not the last track cluster in the schedule. This contradiction eventually proves the statement that page P_k is in a column with M target pages. The cost of schedules is maximized if column j is identical to column j_0 and track cluster CL_k consists of PT pages. Thus, in the worst case, the cost of the schedule S is at most $M * PT + j_0 + 1$. □

For a given target set, there is still some freedom in selecting SLTF schedules. So far, we have not explained which page of a complete track cluster (i.e. occupying all pages of a track) serves as the page read first. For the transformed schedule S_T, this page determines the column where an artificial target page is inserted. Due to Theorem 6.1.1, our primary goal is to keep the maximum number of target pages as low as possible in the transformed schedule. Thus, for each complete track cluster, we proceed as follows: First, we determine the column, say j, which contains a minimum number of target pages. Ties might be resolved by selecting the column with the smallest index (this is only advisable when the initial position of the disk arm is in front of column 0). Reading the track cluster should then start from column

Figure 6.3: An example of a target set that has an expensive SLTF schedule

$(j+1)\%PT$.

The same comment can be made as for the *Elevator* algorithm. SLTF schedules do not impose any restrictions on a disk with respect to track skewing and head switch time.

6.1.3 Look-Back Algorithm

Although the shortest-latency-time-first algorithm is shown to be close to optimum, the cost of its schedule can be almost double the minimum cost, in the worst case. An example for such an undesirable situation is shown in Figure 6.3. The cost of the SLTF schedule (A,B,...,H) would be 16, whereas the cost of the optimal schedule $A, C, \ldots, H, B$ is only 10. In order to reduce cost, it is obviously necessary to allow that portions of a track cluster can be read. A schedule is then no longer an SLTF schedule. This was our starting point for designing a new algorithm which will be given in this subsection.

The basic idea of our new approach is to consider sequences of 2 adjacent columns in a cylinder. The i-th sequence G_i^2, $0 \leq i < PT$, also called *i-th group of size* 2, is defined as the set of target pages found in column i and column $(i+1)\%PT$. Since our discussion is restricted to groups of size 2, we simply use G_i to refer to the i-th group. For the example illustrated in Figure 6.1, we obtain $G_0 = \{A, B\}$, $G_1 = \{B, C, D\}$, $G_2 = \{C, D\}$, $G_3 = \{E\}$, etc. It is obvious that two target pages from the same group cannot be read in one disk revolution when they are on different tracks. Therefore, the following algorithm tries to reduce the number of occupied tracks in the groups of a cylinder.

Algorithm **Look-Back**

Initial state: the disk arm is in front of column 0, $j_{old} = PT - 1$.

BEGIN

WHILE (there is an unprocessed target page) DO

1. Determine the next group G_j, $j = (j_{old} + 1)\%PT, (j_{old} + 2)\%PT, \ldots$ which contains at least one unprocessed target page.
2. (a) If there is a track t such that $C_{t,j}$ is an unprocessed target page and $C_{t,(j-1)\%PT}$ is an empty page, put the disk arm in front of page $C_{t,j}$.
   (b) Otherwise, if there is a track t such that $C_{t,j}$ is an unprocessed target page, put the disk arm in front of page $C_{t,j}$.
   (c) Otherwise (i.e. there is not an unprocessed target page in the j-th column), put the disk arm in front of an arbitrary unprocessed target page, say $C_{t,(j+1)\%PT}$.
3. Read unprocessed target pages on track t in sequential fashion until an empty page or an processed target page appears in front of the disk arm. Let $G_{j_{old}}$ be the group in front of the disk arm.

END;

END Look-Back;

In order to illustrate the algorithm *Look-Back* we will again use the example which is depicted in Figure 6.1. The algorithm first reads page A. It then reads page D because the predecessor page on the same track is not a target page (see step 2 (a) in the algorithm). After page E has been processed, the algorithm reads page G. This is an example where algorithm *Look-Back* reads track clusters partly. The remaining target pages are then transferred into main memory in the order B, C, and F. Overall, the algorithm requires 13 transfer units for reading all the target pages. The name *Look-Back* refers to the policy used in step 2 (a) of the algorithm.

The example illustrated in Figure 6.4 shows that the *Look-Back* algorithm does not always compute the optimal schedule for reading the target pages. The algorithm computes the schedule $A, B, \ldots, H$, whereas the optimal schedule is $A, B, \ldots, F, H, G$. As illustrated in our example, optimal read schedules do not always read pages until the end of the track cluster. Sometimes it is beneficial to stop earlier and to perform a head switch.

Before giving the next lemma, let us introduce another notation. For a given state of the algorithm *Look-Back* and for given group G_i, $0 \leq i < PT$, the function $Tracks(G_i)$

Figure 6.4: Cylinder with some target pages

returns the number of tracks occupied by at least one of the unprocessed target pages of group G_i. For our example illustrated in Figure 6.1, $Tracks(G_0) = 2$ and $Tracks(G_3) = 1$ can be observed before the algorithm enters the loop. Moreover, M is defined as $M = \max_{i=0,\ldots,PT-1} Tracks(G_i)$ at the moment of initialization.

Lemma 6.1.2 *Consider that N target pages are distributed among the pages $C_{i,j}$ of a cylinder, $0 \leq i < TC$ and $0 \leq j < PT$. Then, the algorithm* Look-Back *requires at most $PT + 1$ page transfers for decrementing M by 1.*

Proof: Let $Seq = G_{i_1}, G_{i_2}, \ldots, G_{i_k}$, $1 \leq k \leq PT$, be the sequence of the groups which fulfill $Tracks(G_{i_j}) = M$, $0 \leq i_j < PT$ and $1 \leq j < k$, at the moment of initialization. We assume that Seq is ordered according to the index of the groups, i.e. $i_j < i_{j+1}$ is fulfilled for $0 \leq j < k - 1$. Now, we assume (in contradiction to the statement of the lemma) that the following property holds:

(*) There is a distribution of target pages such that algorithm *Look-Back* requires more than $PT + 1$ page transfers to reduce M.

Due to statement (*), there is a distribution of target pages such that for at least one group $G_m \in Seq$ it is not possible to reduce $M = Tracks(G_m)$. Let G_m be the first group in Seq with that property. Then, when the disk arm arrives at G_m,

$$Tracks(G_i) < M \quad \text{for } 0 \leq i < m - 1 \tag{6.1}$$

is fulfilled.

First, let us assume that the disk arm has read the last page from column $(m-2)\%PT$ before it moves to one of the columns of group G_m. Then, G_m is considered as the next group in the algorithm. First, the algorithm would have tried to read one of the target pages in column m. If no target page can be found in column m, the disk arm would have read a target page from column $(m+1)\%PT$. In both situations, the number of occupied tracks in group G_m is reduced and thus, $Tracks(G_m) < M$. Because this is in contradiction to our statement (*), the disk arm has read the last target page from column $(m-1)\%PT$. Thus, the algorithm considers group $G_{(m+1)\%PT}$ as the next group (see step 1 of the algorithm).

Due to step $2(a)$, the algorithm would have tried to read one of the pages from column $(m+1)\%PT$ that do not have a predecessor target page on the same track in column m. If such a page exists, $Tracks(G_m)$ will be reduced (in contradiction to statement (*)). It follows that such a target page does not exist in column $(m+1)\%PT$. The group G_m only consists of target pages which are in column m and of pairs of target pages stored on the same track in column m and $(m+1)\%PT$. Thus, $M = Track(G_m)$ is equal to the number of target pages in column m. However, the number of target pages in column m is a lower bound for $Track(G_{m-1})$ and so, $Tracks(G_{m-1}) = Tracks(G_m) = M$ is true. This result is in contradiction to inequality 6.1.

Statement (*) has been shown to be false and therefore the statement of the theorem is true. □

Theorem 6.1.3 *Consider that N target pages are distributed among the pages $C_{i,j}$ of a cylinder, $0 \leq i < TC$ and $0 \leq j < PT$. The cost of the schedules computed by the* Look-Back *algorithm is then at most $(PT+1) * M$ page transfers.*

Proof: Let us consider the pages that the disk arm passes over as a sequence partitioned into $(M-1)$ subsequences of size $PT+1$ and one subsequence with at most $PT+1$ pages. The first subsequence starts in column 0 of the cylinder. Thus, when target pages are in group G_0, algorithm *Look-Back* decrements $Tracks(G_0)$ by 1. In general, the i-th subsequence starts in the $i\%PT$ column of the cylinder.

From Lemma 6.1.2, we already know that M is decremented by 1 after the first subsequence is processed. What remains to be proven is whether algorithm *Look-Back* guarantees

that the maximum is reduced by i when the disk arm is at the beginning of the $(i+1)$-st subsequence.

Without loss of generality, let us assume that the maximum could not be reduced in the second subsequence. This is only possible if the last page of the first subsequence is read from column 0 and a head switch is performed such that the next page is read from column 2. In that case, the number of occupied tracks in group G_0 is at most $M-2$ (note that $Tracks(G_0)$ was already reduced at the beginning of the first subsequence). Moreover, it follows from Lemma 6.1.2 that the maximum number of occupied tracks can be reduced for groups $G_2, G_3, \ldots$ of the second subsequence. The remaining question is whether $Tracks(G_1)$ can also be decremented in the second subsequence, if $Tracks(G_1) = M-1$ is fulfilled.

Let us assume that $Tracks(G_1) = M-1$ could not be reduced in the second subsequence and that reading starts from the second column, i.e. the right column, of group G_1. Then, the algorithm *Look-Back* cannot find a page in column 2 whose predecessor page (on the same track) is not a target page. Therefore, $M-1$ is equal to the number of target pages that can be found in the left column and thus, group G_0 has at least $M-1$ occupied tracks. This is in contradiction to our statement that $Tracks(G_0) \leq M-2$. Hence, if the maximum $M-1$ is adopted in group G_1, the algorithm *Look-Back* decrements $Tracks(G_1)$ by 1 and thus, the statement of the theorem follows for $M \leq 2$.

The same technique can be used to obtain the result of the theorem for an arbitrary value of M. □

Note that the upper bound $(PT+1) * M$ is indeed close to optimum. In order to show that, let us consider an SLTF schedule S and its transformed schedule S_T. Then, at the moment of initialization, $Tracks(G_i)$ is equal to the number of target pages (including the artificial ones) in the i-th column of the transformed situation. Thus, $(M-1) * PT$ is a lower bound for the cost of any schedule. This shows that the schedules of the *Look-Back* algorithm are also close to optimum.

6.1.4 Comparison

In this subsection, results obtained from an experimental performance comparison are

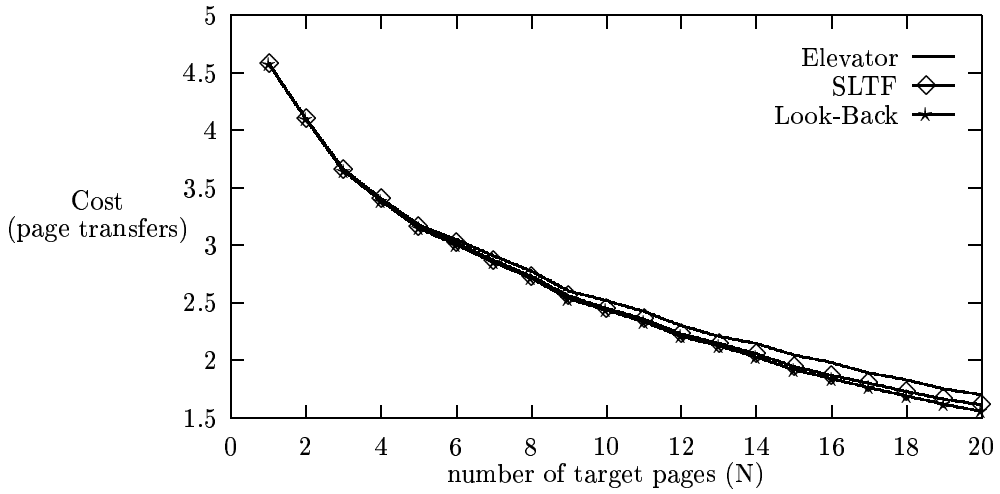

Figure 6.5: The cost of schedules produced by the different algorithms

shown. In Figure 6.5, the average cost is depicted for the schedules produced by the algorithms. The cost (given in page transfers) varies in the number of target pages. The disk consists of 4 tracks and 8 columns ($TC = 4$, $PT = 8$). As illustrated, there is almost no difference in performance between the schedules produced by the different algorithms. The schedules of the *Look-Back* algorithm are slightly better than the ones of the other algorithms only in the case of large numbers of requests. We observed that the difference in performance was even less when the number of tracks was higher.

6.2 Two Cost Estimations

Due to the complexity of obtaining an analysis for the cost of read schedules under the assumption of the idealized disk model (see chapter 5), the reader might guess that an exact analysis is even more complex for a model that takes into account head switch time. In order to obtain cost estimations, our goal is to derive rather simple approximate cost functions for the schedules of the algorithms. In this section, the cost functions are based on the cost function $ICost$, see formula 5.15, which was originally derived under the assumption of the idealized disk model. Target pages are assumed to be on one cylinder in the following

discussion. The more general approach that target pages are distributed among several cylinders can be obtained by using the same techniques already presented in Section 5.5. Furthermore, the following discussion is restricted to the special case that a head switch requires the time of one page transfer.

One essential point is that the cost functions make use of the number of times a head switch has to be performed. A head switch is performed when the next page on the actual track is an empty page. Note that a head switch is actually not necessary when the next target page is read from the same track. However, since the cost is the same as if a head switch was performed, we simply call that operation a head switch as well. A head switch, then, occurs after all pages of a track cluster are transferred into main memory. For the *Elevator* algorithm, the number of track clusters is only a lower bound to the number of head switches. The reason is that when the read/write head is switched to another track, the next read operation may start from a target page that is right of the left-most target page in a track cluster. As a result, one head switch is generally not sufficient to read all pages of a track cluster. Since the number of track clusters is essential to our cost estimations, we first deal with the distribution of track clusters in the next subsection. Next, we will present two cost estimations in the following subsections.

6.2.1 On the Distribution of Track Clusters

Let N be the number of target pages uniformly distributed among the pages of a cylinder. In this subsection, we focus on two important issues: First, computing the expected number of track clusters, and second, deriving the probability that a track cluster contains i, $i \leq PT$, target pages.

An entire track cluster is kept on a single track. Thus, the number of pages in a track cluster is determined by the number of target pages which are on the corresponding track. The probability u_i, $0 \leq i \leq PT$, that a track receives i target pages is given as

$$u_i = \frac{\binom{PT*(TC-1)}{N-i}}{\binom{PT*TC}{N}} \tag{6.2}$$

Let us consider a track that contains i, $1 \leq i \leq PT$, target pages, and a page P stored on the track. For $i < PT$, page P is the first target page of a track cluster, if its predecessor

page is an empty page. The probability that a page on the track is the first page on a track cluster is

$$\frac{\binom{PT-2}{i-1}}{\binom{PT}{i}} = i * \frac{PT-i}{PT(PT-1)} \tag{6.3}$$

and the expected number of track clusters on one track is

$$V_i = \sum_{j=1}^{PT} \frac{\binom{PT-2}{i-1}}{\binom{PT}{i}} = i * \frac{PT-i}{PT-1} \tag{6.4}$$

For $i = PT$, one of the pages is randomly selected to be the first page of the track cluster. Thus, the probability of a page being the first of the track cluster is $\frac{1}{PT}$, and the expected number of track clusters is exactly 1 ($= V_{PT}$).

The general formula for the expected number of clusters on a track can be derived from formula 6.2 and formula 6.4. This yields

$$\sum_{i=1}^{\min(PT,N)} \frac{\binom{PT*(TC-1)}{N-i}}{\binom{PT*TC}{N}} \times V_i \tag{6.5}$$

For a cylinder, the expected number of track clusters is simply given as

$$Cl(N) = TC * \sum_{i=1}^{\min(PT,N)} \frac{\binom{PT*(TC-1)}{N-i}}{\binom{PT*TC}{N}} \times V_i \tag{6.6}$$

Now, let us compute the probability that a track cluster contains j, $j \leq PT$, target pages. Let $Q(i,j)$ denote the probability that a track cluster contains j, $1 \leq j \leq PT$, target pages, under the assumption that i, $j \leq i \leq PT$, target pages are on the track. Consider a track with i, $1 \leq i \leq PT$, target pages. For $i < PT$, a page is the first target page of a track cluster with probability $\frac{i*(PT-i)}{PT*(PT-1)}$. For $i < PT-1$, the track cluster is of length j, $1 \leq j \leq i$ with probability

$$Q(i,j) = \frac{\binom{PT-j-2}{i-j}}{\frac{i*(PT-i)}{PT*(PT-1)}} = \frac{PT*(PT-1)}{i*(PT-i)} \binom{PT-j-2}{i-j} \tag{6.7}$$

For $i \geq PT-1$, we obtain

$$Q(i,j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

6.2.2 First Estimation

Let N be the target pages which are required from a cylinder. Then, we suggest using

$$GCost_1(N) = ICost(N, 1) + \frac{Cl(N)}{N} \tag{6.9}$$

as an estimate for the overall cost. The first term refers to the time required for reading N target pages under the assumption of the idealized disk model. The second term estimates the cost for the head switches. The number of track clusters $Cl(N)$ (see formula 6.6) serves as a lower bound for the number of head switches. For each head switch, a cost of one page transfer is charged.

Note that formula 6.9 can be generalized to the case when head switch time does not correspond to one page transfer. In that case, a factor would be associated with the number of track clusters.

6.2.3 Second Estimation

The second estimate is also computed using the cost function of the IDM. The basic idea is to model a head switch as an additional target page. Let $H(N)$ be the expected number of required head switches (for simplicity it is assumed to be an integer). We suggest using

$$GCost_2(N) = ICost(N + H(N), 1) \tag{6.10}$$

as an estimate of the expected cost of the schedules. Note that our basic assumption on the cost function $ICost$ is that target pages are uniformly distributed over the cylinder. This assumption is not valid when additional target pages are created for modeling a head switch.

In formula 6.9, the number of track clusters is used for estimating the number of head switches. In the following, we present another formula for estimating the number of head switches related to the *Elevator* algorithm.

Consider a track cluster of m target pages, $1 \leq m \leq PT$. If the first page of the track cluster is hit, all of the m target pages are read from the track cluster. Otherwise, when the i-th page of the track cluster is first hit, $1 < i \leq m$, only $m - i + 1$ pages are transferred into main memory. Note that this situation may occur when either the *Elevator* algorithm or the *Look-Back* algorithm is used. After the pages are transferred, the number of track

clusters remains the same and thus the number of head switches is not reduced. In order to obtain the number of head switches, we need an estimate for how many times a track cluster is accessed until it has been completely processed.

In order to make the analysis tractable, we assume that after a head switch is performed, every page of the next track cluster is hit with the same probability. This is obviously a simplification. For example, the *Look-Back* algorithm prefers those track clusters which are completely transferred into main memory. For the *Elevator* algorithm, at least the first page is hit with a higher probability than the other pages of a track cluster.

Let us assume that each page of a track cluster is hit with the same probability. Let X_i, $1 \leq i \leq PT$, denote the expected number of accesses on a track cluster with i pages. It immediately follows that $X_1 = 1$. For $i > 1$, we yield the following recurrence relation:

$$X_i = 1 + \frac{1}{i} \sum_{j=1}^{i-1} X_j \tag{6.11}$$

This recurrence relation can be solved using well-known techniques [GKP89]. The final equation is given as

$$X_i = \sum_{j=1}^{i} \frac{1}{i} \tag{6.12}$$

The number of head switches $H(N)$ is then estimated using the following formula:

$$\sum_{i=1}^{\min(N,PT)} \sum_{j=1}^{i} \frac{u_i}{1-u_0} * Q(j,i) * X_j \tag{6.13}$$

Note that the expression $\frac{u_i}{1-u_0}$ refers to the conditional probability that i target pages are on a track, under the assumption that there is at least one target page on the track.

6.2.4 Experimental Comparison

In the previous subsections, we addressed the problem of read schedules under a disk model that incorporates the time for switching the disk head from one track to another. The graphs in Figure 6.6 demonstrate that the cost of schedules depends substantially on the head switch time. The experiments are performed on a cylinder that consists of 8 columns and 20 tracks ($PT = 8, TC = 20$). The first graph shows the cost for reading target pages without considering head switch time, whereas the other graph shows the cost under the

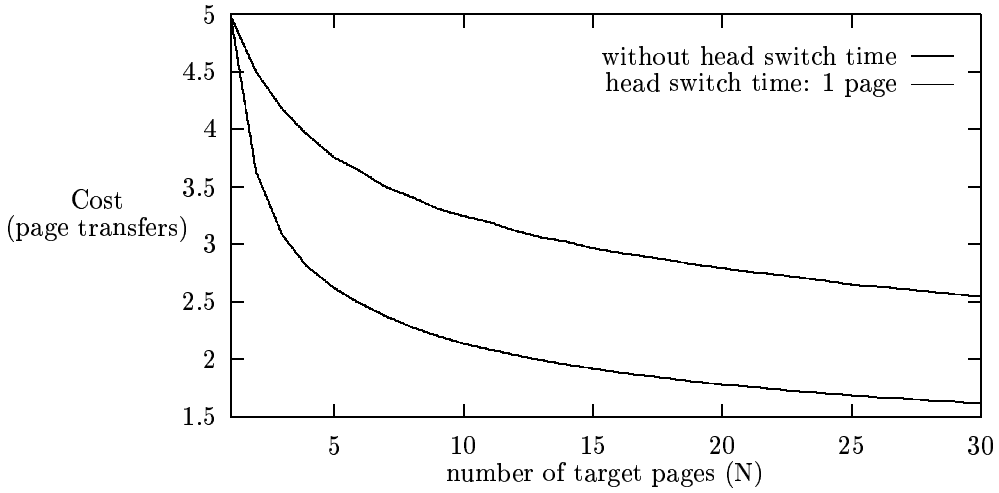

Figure 6.6: The cost of multi-page requests for reading N target pages when head switch time is 0 and 1 page transfer

assumption that a head switch corresponds to one page transfer. The schedules are assumed to be produced by the *Elevator* algorithm. The cost is given in units of page transfers required for reading a target page. The cost varies in N, the number of target pages. Let us take a closer look at $N = 10$. Without considering head switch time, the cost per target page is 2.14 page transfers, while 3.24 page transfers are required per target page when the expense of a head switch is one page transfer. Overall, this experiment demonstrates the necessity for cost functions which consider head switch time.

An obvious question is how accurate are the approximations which are proposed in the previous section. In order to provide an answer, we compared in various experiments the results obtained from these approximate formulas to the ones obtained from simulations. Since all algorithms produce very similar schedules, the simulations are again restricted to the *Elevator* algorithm. A cylinder consists of 8 columns and 20 tracks. The graphs in Figure 6.7 plot the expected cost per target page for a multi-page request. Two of the graphs are plotted using the cost estimations $GCost_1$ and $GCost_2$, whereas the third graph is obtained from simulations. The cost of a multi-page request varies with the number of

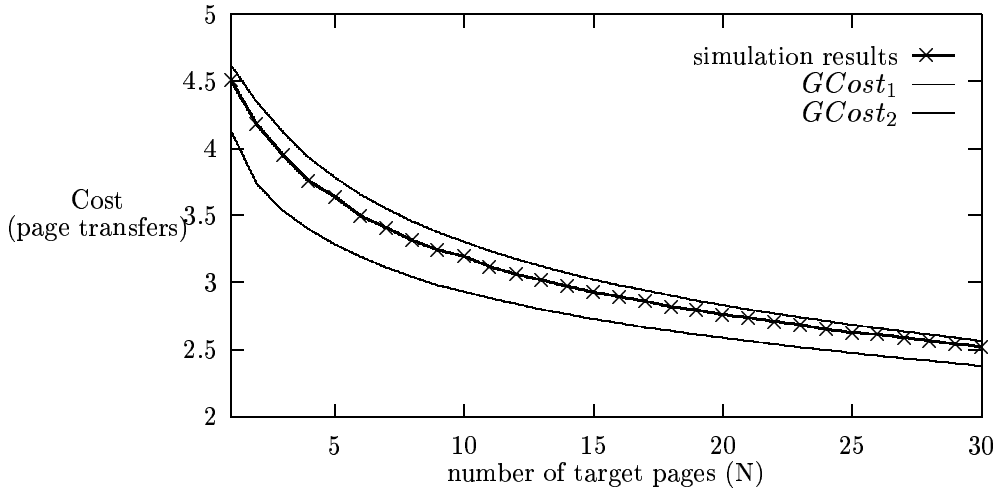

Figure 6.7: A comparison of cost estimates $GCost_1$ and $GCost_2$ with simulation results (varying in N, $TC = 20$, $PT = 8$)

target pages. Both of the approximations produce results close to the actual cost.

In our experiments, the cost estimation $GCost_2$ is almost always more accurate than $GCost_1$. For N sufficiently small, $GCost_2$ overestimates the actual cost by at most a factor of 6%. Only for a large N (i.e. large target sets), does the cost function underestimate the actual cost. We do not observe a different behavior when parameters PT and TC are varied. In Figure 6.8 and Figure 6.9, results of a simulation are compared with the ones obtained from the cost functions. In these experiments, the underlying query requires 10 target pages from a cylinder, where TC and PT are varied. The cost estimate $GCost_2$ shows a high accuracy almost independent of PT and TC, whereas the quality of the estimation $GCost_1$ worsens with an increasing number of columns and with an increasing number of tracks. We conclude that $GCost_2$ provides an accurate cost estimation when the cost for a head switch corresponds to one page transfer.

The two cost estimates can also be generalized to estimate the cost for a head switch time of two (or more) page transfers. For example, we examined the formula $ICost(N + HST * H(N), 1)$ for estimating the cost when head switch time is HST page transfer,

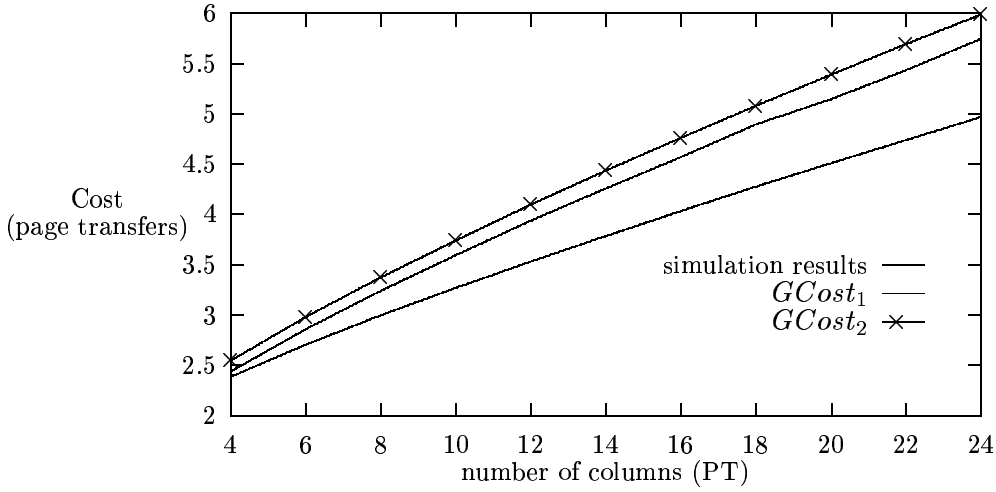

Figure 6.8: A comparison of cost estimates $GCost_1$ and $GCost_2$ with simulation results (varying in PT, $N = 10$, $TC = 10$)

$HST \in \{1, 2, \ldots\}$. However, our experiments showed that the quality of the approximations worsens with an increasing value of HST. When $HST = 2$, the relative error of the second approximation is about 10% compared to the results obtained from simulations.

Both of the cost estimates considered so far present several deficiencies. First of all, the underlying disk model is still rather simple. The cost functions may not be suitable for estimating the cost on a disk whose head switch time does not correspond to exactly one page transfer. Moreover, the model that a cylinder corresponds to a two-dimensional array of pages is not in agreement with the disk geometry of most of today's disks. Instead, a cylinder can be viewed as an array of sectors, and a page corresponds to a contiguous sequence of sectors. Track skewing, for example, shifts the beginning of a track to the right by a certain number of sectors with respect to the beginning of the previous track. Eventually, the capacity PT of a track (in pages) will generally not be an integer, i.e. there are pages in a cylinder which are spread over two tracks. Second, the practical use of these cost functions is questionable. These functions require that two recurrence relations are solved. The computation of these recurrence relations is time-intensive. Moreover, rounding errors may also become a problem.

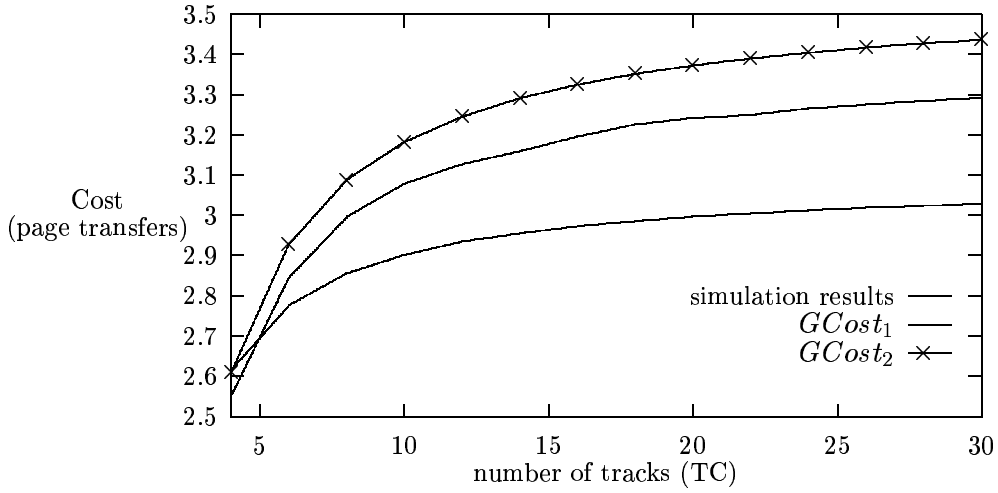

Figure 6.9: A comparison of cost estimates $GCost_1$ and $GCost_2$ with simulation results (varying in TC, $N = 10$, $PT = 8$)

6.3 An Alternative Head-Switch-Time Disk Model

In this section, we address the problem of deriving simple cost formulas which can be employed for estimating the cost of multi-page requests under the assumptions of an alternative head-switch-time disk model. The cost functions will be given in an explicit form without needing to solve recurrence relations. The disk model also considers track skewing and additionally, some of the pages may belong to two adjacent tracks. We refer to the model as the head-switch-time disk model, or *HST model* for short.

For the HST model, a cylinder of the disk corresponds to a sequence C_i of pages, $0 \leq i < PC$. A page occupies SP contiguous sectors in the cylinder, where SP is an integer with $SP \geq 1$. A cylinder consists of TC tracks with addresses $0, \ldots, TC-1$ where TC is an integer. A track is a contiguous sequence of ST sectors where $ST > 0$ is an integer. Note that the parameter $PT = \frac{ST}{SP}$ is generally not an integer. However, we make the simplifying assumption that $PC = PT * TC$ is an integer.

On each of the tracks, a contiguous sequence of pages is stored. The first and the last

page of that sequence are assumed to be only partially on the track. More precisely, for each pair of adjacent tracks, there is a page whose first sector is on the one track and whose last sector is on the other track. The time required for transferring a page which is entirely stored on one track is used to express the cost of a multi-page request. Hence, 1 (page transfer) is required to transfer such a page. In order to transfer a page that crosses a track boundary, a head switch is also required. Thus, a transfer of such a page is $1 + HST$ where HST denotes the time required to perform a head switch. The expected transfer time of a page in a cylinder is then given as

$$1 + HST\frac{TC - 1}{PC} \tag{6.14}$$

The cost analysis is done under the assumption that pages are read with respect to an SLTF schedule. Thus, clusters are read one by one into main memory. The basic idea of the analysis is to compute the expected number of clusters, the expected number of pages in a cluster, and the expected distance between two adjacent clusters. Cost estimations are derived for two cases. The case for $HST \in (0, 1]$ is discussed in the first subsection. In the second subsection, the case for $HST > 1$ is studied. Since a head switch time that is requires more than 2 page transfers very seldom occurs, special attention is given to the case when $HST \in (1, 2]$.

6.3.1 Inexpensive Head Switches

In this subsection, we assume that head switch time is at most 1 page transfer ($HST \leq 1$). The target pages are read with respect to an SLTF schedule. In contrast to our previous discussion, an SLTF schedule is not restricted to contain track clusters only, i.e. clusters whose pages belong to one track. Instead, an SLTF schedule reads clusters one by one which refer to a contiguous sequence of target pages in the cylinder. The sequence must either be surrounded by a pair of empty pages or it must be at the beginning or at the end of the cylinder.

First, an analysis is presented for the expected number of clusters in a cylinder. Let N be the number of target pages, $N \leq PC$. A cluster occurs in a cylinder if one of the following two properties holds. First, a cluster is at the beginning of the cylinder if page C_0 is a target

page. The frequency of this event is

$$\binom{PC-1}{N-1}$$

Second, a cluster starts from the i-th page, $1 \leq i < PC$, of a cylinder if page C_i is a target page and page C_{i-1} is an empty page. For a given i, the frequency of that event is

$$\binom{PC-2}{N-1}$$

The expected number of clusters NC is therefore given as

$$NC = \frac{\binom{PC-1}{N-1} + (PC-1)\binom{PC-2}{N-1}}{\binom{PC}{N}}$$

Some basic calculations simplify the equation to

$$NC = N(1 - \frac{N-1}{PC}) \tag{6.15}$$

The time required for performing all head switches can now be computed as $HST * NC = HST * N(1 - \frac{N-1}{PC})$.

Second, let us compute the cost for transferring the pages. This is an easy task. Since an SLTF schedule only transfers clusters and since a cluster only consists of target pages, the expected transfer cost follows from equation 6.14. We obtain

$$N(1 + HST\frac{TC-1}{PC}) \tag{6.16}$$

What remains to be done is to compute the latency when the disk arm moves from the end of a completely processed cluster to the beginning of the next unprocessed cluster.

For a given cluster, let $p \in [0,1)$ be the (relative) starting position in the corresponding track. Since a cluster is aligned to sectors, p adopts an element in the set $\{0, \frac{1}{ST}, \ldots, \frac{ST-1}{ST}\}$. For the sake of simplicity, we assume that the starting position of a cluster is uniformly distributed in $[0,1)$. In addition to a simplified analysis, this assumption also serves for modeling track skewing to a certain degree.

Let $CL_1, \ldots, CL_k$ be the clusters whose starting positions are denoted by $p_1, \ldots, p_k$, respectively $(0 \leq p_i < 1, 1 \leq i \leq k)$. Let x_k, $0 \leq x_k < 1$ be the position of the disk arm. We make the simplifying assumption that positions $p_1, \ldots, p_k, x_k$ are uniformly distributed

in $[0, 1)$. Then, the expected distance from position x_k to the next position where a cluster, say CL_j, begins is given as

$$\frac{1}{k+1} \tag{6.17}$$

After the pages of the cluster CL_j are transferred and a head switch is performed, the disk arm arrives at a new position x_{k-1}, $0 \leq x_{k-1} < 1$. Again, we make the simplifying assumption that positions $p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_k, x_{k-1}$ are uniformly distributed. It follows that the expected distance to the next cluster is $\frac{1}{k}$. This process is repeated until all clusters are processed. Overall, our equation for estimating the latency $LT(k)$ is given as

$$LT(k) = PT * \sum_{i=2}^{k+1} \frac{1}{i} \tag{6.18}$$

The sum is one less than the $(k+1)$-st harmonic number. The computation of the sum can be avoided by using the approximation presented in [GKP89]. We then obtain

$$LT(k) = PT * (\ln(k+1) + \frac{1}{2(k+1)} - \frac{1}{12(k+1)^2} + \gamma - 1) \tag{6.19}$$

where $\gamma = 0.5772156649\ldots$ is Euler's constant.

By combining equations 6.15, 6.16, and 6.19, we obtain our cost estimation Est_{HST}, $HST \in [0, 1)$, as follows.

$$Est_{HST}(N) = 1 + HST(\frac{TC-1}{PC} + 1 - \frac{NC-1}{N}) + \frac{LT(NC)}{N} \tag{6.20}$$

6.3.2 Expensive Head Switches

The case when the cost of a head switch is more than one page transfer follows the same approach as discussed above. We now assume that HST is in the range $(m-1, m]$ for an integer m, $m \geq 2$. In addition to the general cost formulas, simplified ones are derived for the case $m = 2$. Note that the cases for $m = 1, 2$ are the most interesting ones because the head switch time of today's disks is mostly less than two page transfers. For $HST \in (m-1, m]$, a cluster refers to a contiguous sequence of pages with the following properties:

- the first and the last page of the sequence are target pages
- the sequence does not contain a contiguous sequence of m empty pages

- the sequence is delimited on the left and on the right by a contiguous sequences of empty pages (or pages outside of the cylinder) longer than m

Note that this definition of a cluster is almost the same as the one given on page 54. The only difference is that a cylinder is now assumed to be limited in size. Thus, the following analysis is not asymptotic, but exact.

Before going into more details, let us point out an important observation. An algorithm that reads cluster one by one does not give us an SLTF schedule anymore. When an empty page of a cluster is stored on two contiguous tracks, the cost for reading the page would be $1 + HST$. In contrast, an SLTF schedule would have avoided the transfer of the page and a head switch would have be performed instead. Therefore, the cost of an SLTF schedule is less than the cost of a schedule that reads cluster one by one. For practical values of m, the probability is very low that an empty page will cross the end of the track and that the same page is in a cluster. For simplicity, our analysis is restricted to the case that clusters are read one by one.

First, the expected number of clusters is computed again. Consider that N target pages are uniformly distributed among the pages $C_0, \ldots, C_{PC-1}$ of the cylinder. For $i \geq m$, page C_i is the first page of a cluster if C_i is a target page, and pages C_j, $i - m \leq j < i$ are empty pages. This occurs with probability

$$\frac{\binom{PC-m-1}{N-1}}{\binom{PC}{N}}$$

For $i < m$, page C_i is the first page of a cluster, if C_i is a target page and pages C_j, $0 \leq j < i$, are empty pages. Then, the expected number of clusters $NC(N, m)$ is given as

$$NC(N,m) = \frac{(PC-2)\binom{PC-3}{N-1} + \sum_{i=1}^{m}\binom{PC-i}{N-1}}{\binom{PC}{N}}$$

Some simple calculations result in the following equation:

$$NC(N,m) = 1 + \frac{(N-1)\binom{PC-m}{N}}{\binom{PC}{N}} \tag{6.21}$$

For $m = 2$, the formula can be further simplified to

$$NC(N,2) = N * (1 - \frac{(N-1)(2PC-1-N)}{PC(PC-1)}) \tag{6.22}$$

Next, we compute the time required for transferring the target pages of a cluster into main memory. This time corresponds to the time that would be required to transfer all pages of a cluster (including the empty pages). Therefore, we compute the expected number of empty pages in a cluster. For the special case $N = 1$, there is only a cluster consisting of one target page and hence, only one page has to be transferred.

In the following, we assume $N > 1$. In order to compute the expected number of empty pages in the clusters, we investigate the dual problem of computing the expected number of empty pages that are outside the clusters. First, we discuss the case where a contiguous sequence of empty pages is at the beginning and the end of the cylinder. Later, we deal with contiguous sequences of two (or more) empty pages between two clusters. In both cases of the analysis, the following identity [GKP89] is of great importance.

$$\sum_{0 \le k \le l} \binom{l-k}{m} \binom{q+k}{n} = \binom{l+q+1}{m+n+1}, \quad \text{for integers } l, m \ge 0 \text{ and integers } n \ge q \ge 0 \tag{6.23}$$

Consider that pages $C_0, \ldots, C_{i-1}$, $i > 0$, are empty pages and that page C_i is a target page. The frequency of this event is

$$\binom{PC-1-i}{N-1}$$

Thus, the frequency of empty pages being at the beginning of the cylinder is given by

$$\sum_{i=1}^{PC-N} \binom{PC-1-i}{N-1} i = \sum_{i=1}^{PC-N} \binom{PC-1-i}{N-1} \binom{i}{1}$$

Now, identity 6.23 can be applied to the formula on the right hand side. Then, we obtain

$$\binom{PC}{N+1} \tag{6.24}$$

as the frequency of empty pages at the beginning of the cylinder. For $N > 1$, the same formula gives the frequency of empty pages at the end of the cylinder.

Consider a sequence $C_k, \ldots, C_{k+i-1}$, $i \ge m$, of i empty pages in the cylinder such that pages C_{k-1} and C_{k+i} are target pages. The position k of the first page has to be in the set $\{1, \ldots, PC - i - 1\}$. Otherwise one of the target pages would be outside of the cylinder. Thus, there are $PC - i - 1$ possible positions to store such a sequence on the cylinder. For

each position, the frequency that such a sequence occurs is given as

$$\binom{PC-2-i}{N-2}$$

Overall, the frequency of empty pages that are in a contiguous sequence of at least m empty pages is then

$$\begin{aligned} & \sum_{i=m}^{PC-N} \binom{PC-2-i}{N-2}(PC-1-i)i \\ = \; & (N-1)\sum_{i=2}^{PC-N} \binom{PC-1-i}{N-1} i \end{aligned} \tag{6.25}$$

Now, $j = PC - N - i$ is used as the new index of the sum. Also, the order of the sum is reversed. We obtain the following formula:

$$\begin{aligned} & (N-1)\sum_{j=0}^{PC-N-m} \binom{N-1+j}{N-1}\binom{PC-N-j}{1} \\ = \; & (N-1)\Big(\sum_{j=0}^{PC-N} \binom{PC-N-j}{1}\binom{N-1+j}{N-1} - \sum_{j=PC-N-m-1}^{PC-N} \binom{PC-N-j}{1}\binom{N-1+j}{N-1}\Big) \end{aligned}$$

The identity 6.23 can be applied to the first sum. We then obtain

$$(N-1)\Big(\binom{PC}{N+1} - \sum_{j=PC-N-m-1}^{PC-N} \binom{PC-N-j}{1}\binom{N-1+j}{N-1}\Big) \tag{6.26}$$

Let us now consider the remaining sum. The sum can be simplified as follows.

$$\begin{aligned} & \sum_{j=PC-N-m-1}^{PC-N} \binom{PC-N-j}{1}\binom{N-1+j}{N-1} \\ = \; & \sum_{j=0}^{m-1} \binom{j}{1}\binom{PC-1-j}{N-1} \\ = \; & \sum_{j=0}^{PC-1} \binom{j}{1}\binom{PC-1-j}{N-1} - \sum_{j=m}^{PC-1} j\binom{PC-1-j}{N-1} \\ = \; & \binom{PC}{N+1} - \sum_{j=m}^{PC-1} j\binom{PC-1-j}{N-1} \\ = \; & \binom{PC}{N+1} - \sum_{j=0}^{PC-m-1} (j+m)\binom{PC-1-m-j}{N-1} \end{aligned}$$

$$= \binom{PC}{N+1} - m\sum_{j=0}^{PC-m-1}\binom{PC-1-m-j}{N-1} - \sum_{j=0}^{PC-m-1}\binom{PC-1-m-j}{N-1}\binom{j}{1}$$
$$= \binom{PC}{N+1} - m\binom{PC-m}{N} - \binom{PC-m}{N+1} \quad (6.27)$$

This expression replaces the sum in formula 6.26. Then, by a combination of formulas 6.24 and 6.26, we obtain the number of empty pages outside of a cluster as follows:

$$2\binom{PC}{N+1} + (N-1)(m\binom{PC-m}{N} + \binom{PC-m}{N+1}) \quad (6.28)$$

Thus, the expected transfer cost $TR(N,m)$ of the clusters is

$$TR(N,m) = N - 2\binom{PC}{N+1} - (N-1)(m\binom{PC-m}{N} + \binom{PC-m}{N+1}) \quad (6.29)$$

The formula can be simplified for $m = 2$ as follows

$$TR(N,2) = N(1 + \frac{(N-1)(PC-N)}{PC(PC-1)}) \quad (6.30)$$

We are now able to estimate the cost of a multi-page request on a cylinder when $HST \in (m-1, m]$, $m > 0$. Let N be the number of target pages, uniformly distributed among PC pages. Then, $NC(N,m)$, see formula 6.21, gives the expected number of clusters and $TR(N,m))$, see formula 6.29, gives the cost for transferring the clusters into main memory. The latency to move from one cluster to another is estimated by using equation 6.19. The total cost function is then given as follows

$$Est_{HST}(N) = (1+HST\frac{TC-1}{PC})(1+TR(N,m))+HST\frac{NC(N,m)}{N}+\frac{LT(NC(N,m))}{N} \quad (6.31)$$

For $m = 2$, a simplified formula can be achieved by using the formulas 6.22 and 6.30 for computing $NC(N,2)$ and $TR(N,2))$, respectively.

The cost function Est_{HST}, $HST \geq 1$, estimates the cost of multi-page requests under the assumption of the head-switch-time disk model. In contrast to previous cost functions, one of its most important advantages is that the calculation is very simple and inexpensive. What remains to be shown is that its estimations are close to the cost on a real disk. For that discussion, we refer to the next chapter where we first validate and calibrate a disk simulator. After that, we are able to compare the cost functions and the results of the simulator. In order to give a first idea of the cost function, the corresponding graphs are plotted for $HST = 0.125$

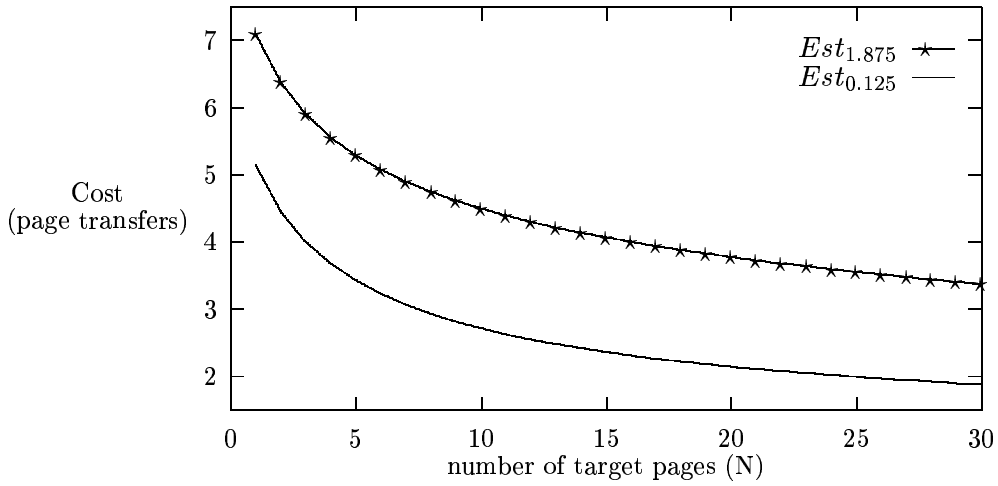

Figure 6.10: Cost function Est_{HST} varying in the number of target pages ($PT = 8$, $TC = 20$)

and $HST = 1.875$ in Figure 6.10. The parameters of the cylinder are $PT = 8$ and $TC = 20$. For both of the graphs, it is assumed that a head switch also occurs when a request for one page is issued. For $HST = 1.825$, the average response time is then not 5, but close to 7 (page transfers).

6.4 Conclusion

In this chapter, we addressed the problem of computing efficient read schedules under the assumption of disk models that take into account head switches. Three algorithms were presented, two of which were shown to produce close-to-optimum schedules under certain assumptions. In an experimental comparison, we showed that all three algorithms produced schedules whose expected cost was almost the same. We then derived two approximate cost functions for estimating the cost of the read schedules. Although the cost functions were in agreement with some of the results obtained from simulation, they did not cover all cases. Moreover, their computation was expensive since the evaluation of two complex recurrence relations was required. Therefore, we looked at an alternative disk model in Section 3. As a

result, we derived a cost function that requires only a few arithmetic operations. In addition to head switch time, the cost function (and the underlying disk model) takes into account track skewing and other properties of a real disk.

Chapter 7

A Comparison of the Disk Models and a Validation

In the last three chapters, we examined several models for magnetic disk drives. Under the assumptions of these models, algorithms were designed for computing efficient read schedules. For these algorithms, exact and approximate formulas were derived for estimating the cost of their schedules.

In the first section of this chapter, the head-switch-time disk model is validated and compared with an implementation on a real magnetic disk drive. As a result, we obtain a disk simulation that behaves almost the same as a real disk drive. The disk simulation gives us freedom to vary the disk geometry and disk speed so that almost any disk drive can be modeled accurately. In the second section, we compare the results of our cost functions with the results obtained from the disk simulation. A third section concludes the chapter.

7.1 Validation of the Disk Model

In order to validate our simulation model, simulation results were compared against experimental results obtained on a Sequent Symmetry, a shared-memory multi-processor system. The system runs DYNIX, a version of the UNIX 4.3BSD operating system. The system has several Fujitsu M2334K disks attached to it whose specifications are given in Table 7.1.

This section is structured as follows. First, the characteristics of the Fujitsu M2344K

Number of cylinders	624
Tracks per cylinder (TC)	27
Sectors per track	66
Sector size [Byte]	512
Track skewing [Sector]	1
Rotational speed [rpm]	3600
Average seek time [ms]	4
Head switch time [ms]	0.25

Table 7.1: Disk parameters of the Fujitsu M2344K

disk are described in more detail. Next, we discuss the most important aspects of the implementation and the simulation. Finally, the results of an experimental comparison are reported.

The Fujitsu M2344K disk drive

In order to verify our disk model, we selected the Fujitsu M2344K disk drive for our experiments. The specifications of the disk are listed in Table 7.1. It is certainly true that the M2344K is a rather old-fashioned disk, but its basic properties still corresponds to those of a modern disk. For example, properties such as track skewing and head switch time greater than zero can already be observed for the M2344K. The disk differs from a modern disk only with respect to the parameter settings. For example, the head switch time of the M2344K is extremely low since the track density is low as a consequence of its 8 *inch* diameter. For our experiments, it was also an advantage that the M2344K does not contain a disk cache. Such a disk cache would have used optimization strategies (e.g. read ahead) which certainly run the risk of conflicting with an efficient read schedule of a multi-page request. Additionally, the performance of a disk without a cache can be more easily predicted than of a disk with a cache. Let us mention that a disk cache has a great impact on the average access cost, as shown in [RW93a]. However, buffers in general do not contribute to reducing the cost of a query that requires target pages only once during query processing. In particular, this holds for data-intensive selection queries on which we have primarily put our emphasis.

Our experiments are restricted to a single cylinder. Thus, seek time is not essential to the validation of our disk model. For the sake of completeness, an approximation for the seek

Figure 7.1: Page layout of the M2344K disk

time is given. In [ZL92], it has been shown that

$$seek\_time(d) = \begin{cases} 4 + \sqrt{d-1} - 0.02653(d-1) & \text{if } d \leq 50 \\ 9.69995 + 0.04066(d-50) & \text{otherwise} \end{cases} \quad (7.1)$$

is a suitable approximation. Seek time is modeled as a function of the seek distance d. The function returns the seek time in ms.

In Figure 7.1, the page layout is depicted for the first tracks of a cylinder of the M2344K. We assume a page size of 4 KB. A page is represented by one or more rectangles. The M2344K offers track skewing, i.e. the beginning of a track is shifted by one sector (0.5 KB) with respect to the beginning of its predecessor track. Assume that the first tracks starts at sector 0. Then the second track starts at sector 1, the third track at sector 2, and so on. In addition to its 66 sectors, a track contains a special sector at the beginning. In Figure 7.1 these sectors are illustrated as black rectangles. Such "free" sectors are rather common on disks mostly because they serve as spare sectors (when a sector on the same track is defective). Since the ratio of sectors per track to sectors per page is not an integer, some of the pages are distributed over two tracks. Note that the transfer time of these pages is increased by two sectors compared to pages completely stored on a single track. The number depicted in a rectangle refers to the address of the corresponding page. If a page crosses a track boundary, it is represented by several rectangles without reporting its address. Instead, the rectangles are illustrated using a common pattern.

Let us discuss the cost of reading pages by the following examples. For simplicity, only the transfer time is considered. When page 2 and then pages 9 are read, the cost is 2.125 (page transfers). Note that these pages can be read in a single disk revolution. When page

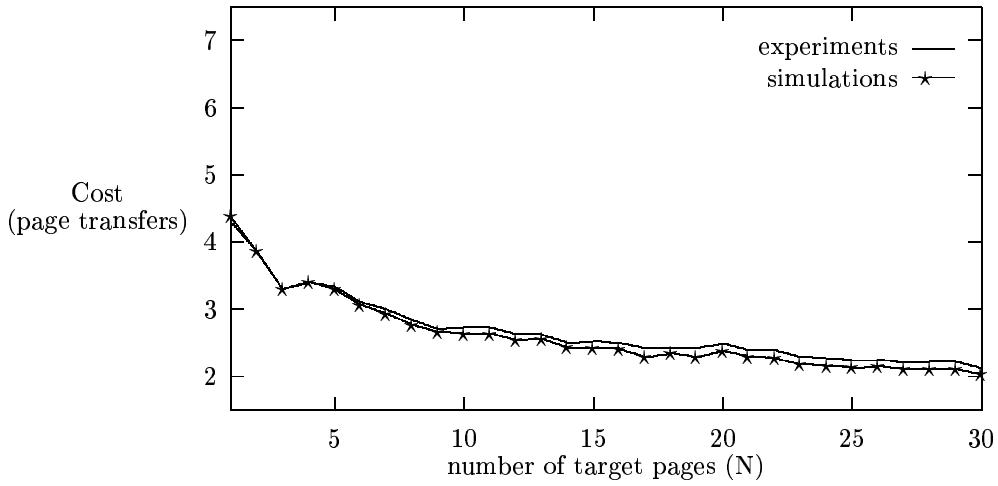

Figure 7.2: Validation of the simulation on the M2344K disk

2 and then page 11 are read, more than one revolution is required. The exact cost is 12.125. Recall that the head switch time of the M2344K is one sector. This allows us to read page 2 after page 9 was read. However, page 2 and page 11 cannot be read in one disk revolution.

Implementation

Multi-page requests are implemented in C using concurrent light-weight tasks, also called *threads* . A thread is similar to a coroutine in Modula-2, and like a UNIX process it presents an independent unit of activity. Our implementation exploits the thread library of the *μSystem* [BS90]. The *μSystem* provides simple but effective mechanisms to deal with threads. In particular, synchronization and execution of threads is under control of the *μSystem*. Context switching between threads is done by by-passing the much slower context switches of ordinary UNIX processes. On the Sequent, a context switch is only 44 μs for the *μSystem*, whereas a context switch of a DYNIX process takes about 10 ms.

Instead of using ordinary UNIX files, the implementation is based on the raw disk for the following reasons. First, I/O-operations on the raw disk by-pass the UNIX buffers and therefore they are performed directly on the disk. The time required for such an operation is

the actual time spent on the disk. Second, a raw disk corresponds to a contiguous sequence of sectors on the disk. Thus, if the layout of sectors is known, it is possible to get control of the physical placement of pages.

In order to process a multi-page request with respect to a given schedule, asynchronous single-page read requests are issued. In our implementation, several reader threads are first created and the page addresses of the multi-page request are written on a stack. A read schedule is then computed that causes a reorganization of the page addresses in the stack. Then, each of the reader threads are concurrently processed as follows: A page address that is fetched from the stack is used for issuing a synchronous read request. The thread waits on the disk until it is completely serviced, i.e. the required page is transferred into main memory. The thread then fetches the next address from the stack, and so on. This processing is repeated as long as the stack contains a page address. In general, at any point in time, one thread occupies the disk, while the others are waiting in a queue in front of the disk. A non-empty queue is essential to our implementation. While one request is on the disk, the disk controller is preparing the waiting requests for their execution. However, this is only possible without delays because context switches produce almost no overhead for the *μSystem*. In particular, the time delay that occurs between two adjacent requests can be neglected. However, it is important to note that the disk controller does not change the ordering of the requests in the queue if the required pages are located on the same cylinder. Thus, it is guaranteed that the ordering in the read schedules is indeed preserved on the disk.

It is interesting to note that our implementation uses the ordinary read command of the DYNIX operating system. For the purpose of our experiments, it was not required to interact with the disk controller or to write channel programs as was reported in [Wei89]. As mentioned above, our implementation runs on a multi-processor machine. So far, the same experiments have not succeeded on a single-processor machines.

Comparison

The main purpose of the first experiment was to validate and calibrate the simulation model. Experimental results obtained from the Sequent machine were compared with simulation results obtained by setting the parameters of the disk model appropriately. The experiment

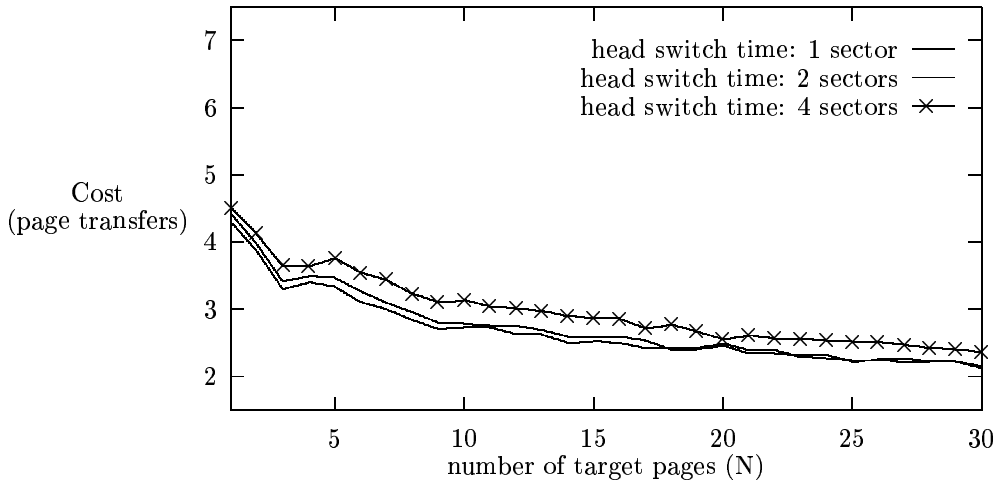

Figure 7.3: Comparison of the cost of a multi-page request for different head switch times

was done using the shortest-latency-time-first policy. For a given number of target pages, 100 target sets of that size are created. Target pages are uniformly distributed among the pages of a cylinder. Thereafter, the pages are read into main memory with respect to the ordering given by an SLTF schedule. Figure 7.2 shows the expected elapsed time for reading one target page averaged over the number of target pages in the 100 target sets. As usual, cost is measured in page transfers. The one curve refers to the simulation results, whereas the other curve depicts the experimental results obtained from our implementation. As demonstrated, the simulation results are in agreement with the experimental results. The relative difference of the experimental results to the simulation results is at most 6% when the head switch time is assumed to be one sector. The relative difference is increased when the number of target pages increases. One reason that may explain this behavior is that the disk controller cannot keep pace with the high transfer and request rate of the disk.

The same experiment was repeated under the assumption that the head switch time corresponds to two and four sectors. The experimental results are depicted in Figure 7.3. There is almost no difference in the results when head switch time is one or two sectors. This can be clearly observed for a large number of target pages. However, the results obtained for

a head switch time of 4 sectors do not approach the other results. Moreover, we observed that the difference between the experimental results and the simulation results is indeed less when the head switch time is set slightly higher than the actual head switch time. For a head switch time of two sectors, the relative difference in the costs was not more than 2% in our experiments.

Overall, we conclude that the results obtained from our experiments are in good agreement with the simulation results. Therefore, it is justified that in our following experiments I/O requests are simulated rather than performed on a real disk. In addition, the simulation approach offers the advantage that an arbitrary disk can be modeled by setting the parameters appropriately. The essential parameters of our simulation model are the number of tracks per cylinder (TC), the number of pages per track (PT), page size, track skewing, and head switch time (HST). For most disks, the head switch time determines track skewing and therefore, it is not considered as a separate parameter in the following.

7.2 A Comparison of the Different Cost Estimates

In this section, simulation results are compared against the cost estimations obtained from our cost functions. The following cost functions are considered in our comparison:

- $ICost$ (see formula 5.15),
- $GCost_1$ (see formula 6.9),
- Est_{HST} (see formula 6.20 and 6.31).

For all our experiments, a sector consisted of 512 bytes and the page size was assumed to be 8 sectors. Parameters PT and HST are reported in units of pages. For example, a head switch time of 1 sector results in $HST = 0.125$.

In Figure 7.4, results are reported from experiments with the following setting of values for the parameters: $PT = 8.25$, $TC = 24$, and $HST = 0.125$. This setting refers to the specification of the Fujitsu M2344K with the exception that TC is chosen slightly lower. There are three curves in Figure 7.4 which are varying in the number of target pages. Each of the curves estimates the expected cost for reading a target page. The cost is given in page

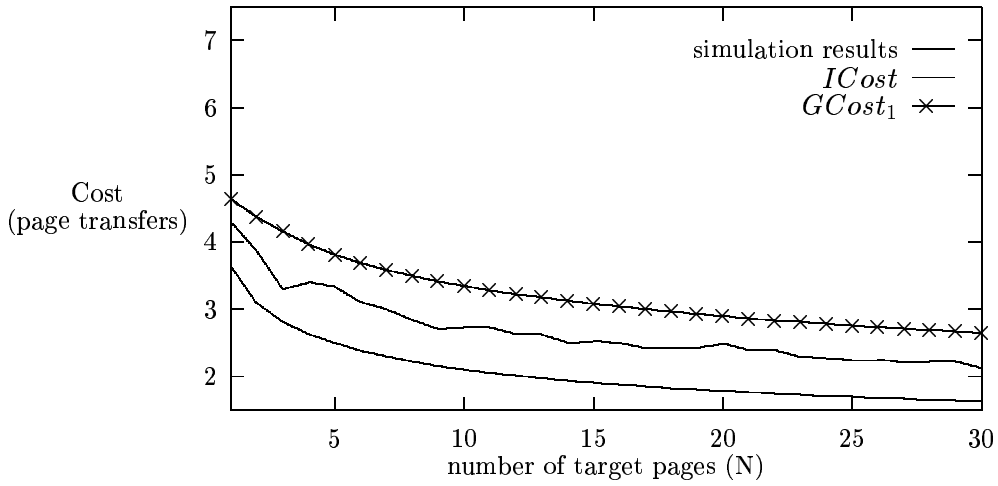

Figure 7.4: Simulation results compared with cost functions $GCost_1$ and $ICost$ (varying in N, $PT = 8.25$, $TC = 24$, $HST = 0.125$)

transfers. The lower curve refers to $ICost$, the cost function of the idealized disk model. The middle curve is obtained from simulations and the upper curve refers to the cost function $GCost$. Note that $GCost$ assumes a head switch time of one page transfer. This may be one of the reasons that it overestimates the actual cost. In contrast, the cost function $ICost$ underestimates the actual cost since head switch time is considered to be zero in the idealized disk model.

For the same experiment, the curve of the cost function $Est_{0.125}$ is reported in Figure 7.5. For comparison, the curve of the simulation results is plotted as well. As demonstrated, there is good agreement between these curves. For small N, the cost function underestimates the actual cost, whereas for $N \in \{5, \ldots, 25\}$ there is an excellent match between the simulation results and the cost function. For large N, the cost function underestimates the simulation results. The relative errors between the cost function and the simulation results is always below 10%. Except for $N = 4$, relative errors are even below 6%. We conclude that cost function $Est_{0.125}$ provides good cost estimations in the experiment.

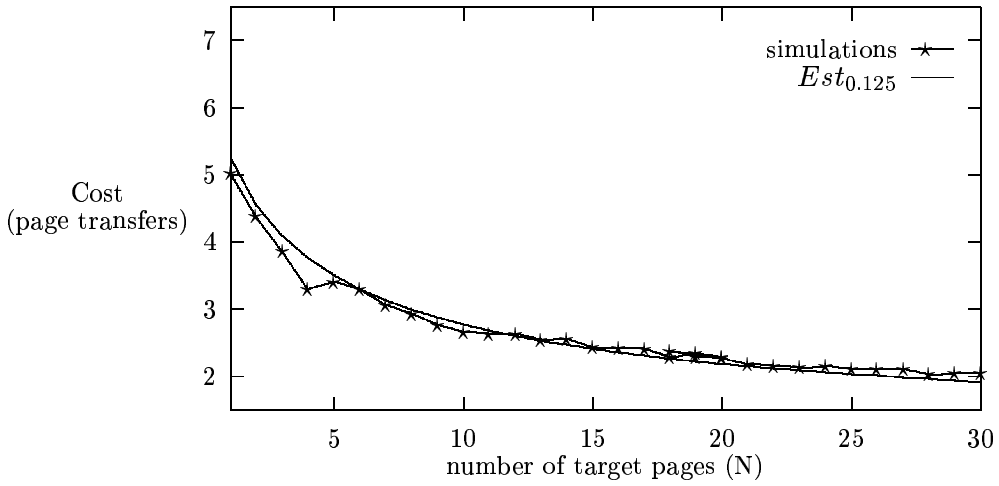

Figure 7.5: Simulation results compared with cost function $Est_{0.125}$ ($PT = 8.25$, $TC = 24$, $HST = 0.125$)

In summary, the cost function of the idealized disk model already shows considerable errors for small head switch times. Since we might expect an increase in the errors for larger values of HST, we disregard that cost function in further experiments. The same is true for the cost function $GCost$. However, we expect to obtain more accurate results when HST is closer to 1. Since the cost function Est_{HST} shows the best match among these cost functions, further experiments are restricted to a comparison of Est_{HST} against simulation results.

In the next sequence of experiments, the cost function Est_{HST} was compared against simulation results. In Figure 7.6, results are depicted for varying N (number of target pages). In contrast to the previous experiment, the head switch time was set rather high ($HST = 1.875$). Let us emphasize that the cost for a head switch is now almost double the cost for a page transfer. As a result, the expected cost for reading one page ($N = 1$) is about 7 page transfers. For $N = 30$, the expected cost for reading a target page is only half of the expected cost for reading a single page. Although performance improvements are now not as high as for low head switch times, there is a considerable performance gain when pages are read with respect to an efficient schedule. Furthermore, Figure 7.6 again demonstrates that the cost

function EST_{HST} is indeed an accurate estimation of the cost.

In Figure 7.7, results are depicted for varying TC (number of tracks). We assumed a target set of 10 pages ($N = 10$). The two curves again show an excellent match for various TC. The relative difference between cost estimation and simulation results slightly increases with an increasing TC. For $TC = 28$, the relative difference is about 7%. However, disk drives with more than 28 surfaces are seldom found. Moreover, today's disk drives tend to have less surfaces than the ones manufactured five to ten years ago. This experiment also shows how clustering on a cylinder affects the cost of a read schedule. If the target pages are distributed on 4 and 28 tracks, the cost per target page is 3.5 and 4.9, respectively. The difference in performance increases with an increasing head switch time, i.e. the higher the head switch time, the more impact clustering has (in a cylinder) on the performance of read schedules.

In Figure 7.8, results are depicted for varying HST (head switch time). We again assumed a target set of 10 pages. The graphs clearly show a linear dependency between cost and HST. As for our previous experiments, there is an excellent match between cost estimation and simulation results. Furthermore, the experiment demonstrates the importance of small HST for the performance of a multi-page request. The cost for a multi-page request on a disk with $HST = 2$ is about 5, i.e. 75% more than the cost on a disk with $HST = 0.125$.

These results are interesting with respect to assessing the performance of disk systems. The old-fashioned Fujitsu M2344K disk drive offers a head switch time of 0.125. From Figure 7.8, we know that the expected cost for a multi-page request of 10 pages is 2.8 (page transfers). A modern disk like the Quantum ProDrive 1050S [Qua92] has a head switch time of 1.75 (for a track that contains 66 sectors). The cost of the multi-page request would be 4.75. However, the advantage of the Quantum disk is that it rotates at 5400 rpm, a factor of 1.5 faster than the Fujitsu disk. In order to compare the cost measures, the result obtained on the Quantum disk is divided by a factor of 1.5. The (normalized) cost is then 3.167, which is still higher than the cost for the request on the Fujitsu disk. This example shows that current developments in disk technology do not always result in improved performance for multi-page requests. Paradoxically, older disk might be faster.

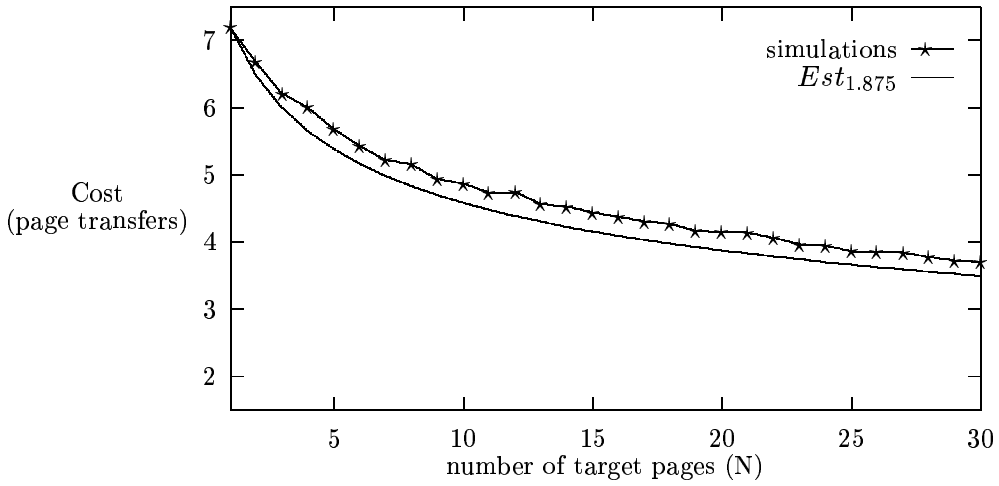

Figure 7.6: Simulation results compared with cost function $Est_{1.875}$ ($TC = 24$, $PT = 8.25$, $HST = 1.875$)

Finally, in Figure 7.9, results are shown when the number of pages on a track (PT) is varying. So far, our cost is given in page transfers required for reading a target page. In this experiment, the cost unit of page transfers is not suitable anymore since the number of pages on a track is not a constant. Instead, cost is measured in the number of disk revolutions required for reading a target page. Additionally, the head switch time is assumed to be 15% of a total disk revolution. The two curves in Figure 7.9 are again in good agreement. They show that best performance can be achieved when the number of pages on a track is high.

7.3 Conclusion

In this chapter, various comparisons were presented on the cost of multi-page requests. First, details were given on how to implement multi-page requests on a UNIX system. The basic idea of our implementation is to perform a multi-page request as a sequence of single page requests. Each request for a single page is performed in an asynchronous fashion using light-weight tasks (threads). Instead of using an ordinary UNIX file, the raw disk interface is used,

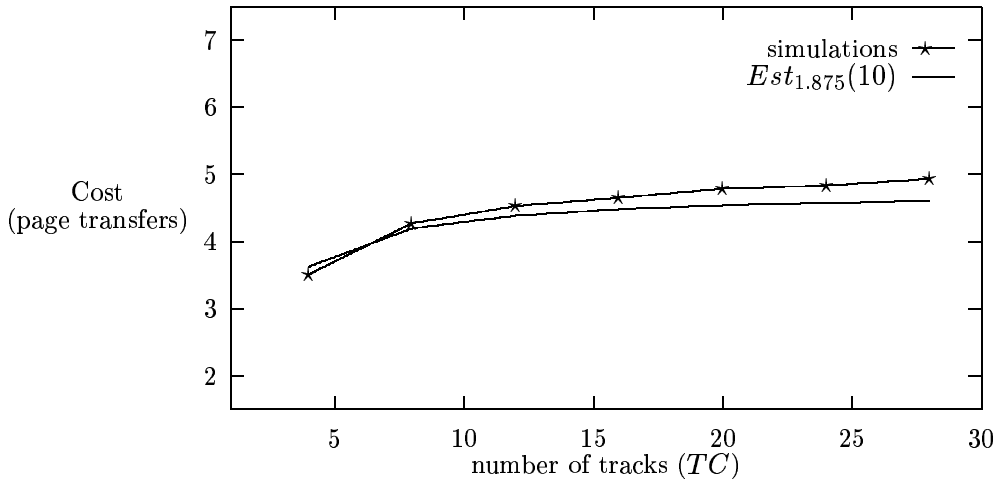

Figure 7.7: Simulation results compared with cost function $Est_{1.875}(10)$ ($N = 10$, $PT = 8.25$, $HST = 1.875$)

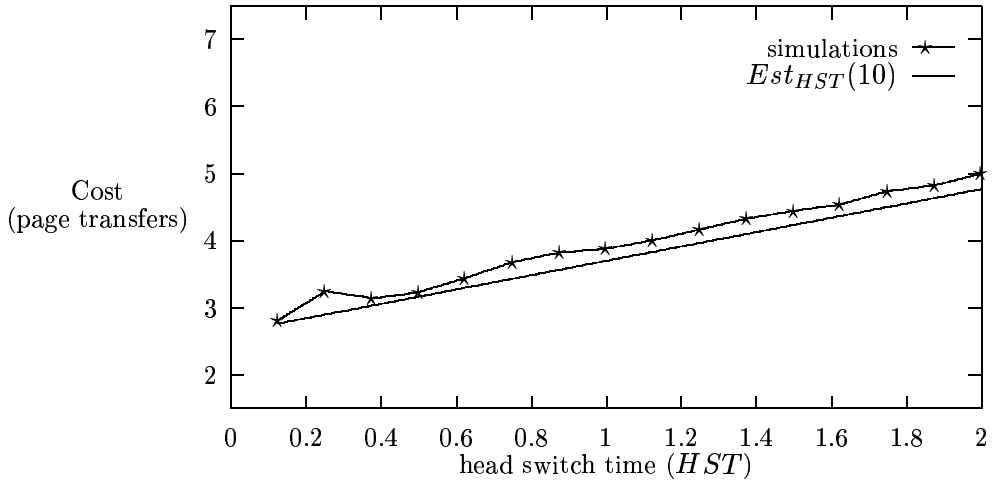

Figure 7.8: Simulation results compared with cost function $Est_{1.875}(10)$ ($N = 10$, $PT = 8.25$, $TC = 24$)

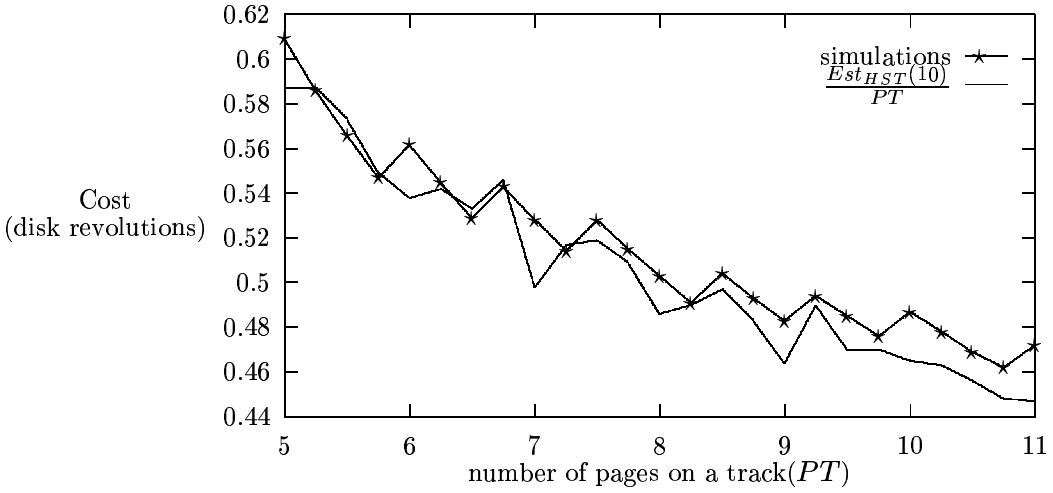

Figure 7.9: Simulation results compared with cost function $\frac{Est_{HST}(10)}{PT}$ ($N = 10$, $PT = 24$)

which allows control of the physical page layout on disk.

Second, a disk simulation was implemented and simulation results were compared against the ones obtained from a real disk. We showed that both results were in excellent agreement. The simulation basically varies in four parameters: the pages on a track, the tracks in a cylinder, the head switch time and the number of target pages. By setting these parameters appropriately, our disk simulation can be adapted to any given real disk.

Third, three of the cost functions derived in the previous sections were compared against the results of our implementation. The cost function Est_{HST}, see formula 6.20 on page 115 and formula 6.31 on page 119, was shown to be very accurate for various settings of the parameters. In all experiments, the relative difference between the cost function and the simulation results was less than 10%. We thus conclude that the cost function may serve as an excellent estimate for the cost of a multi-page request.

Chapter 8

Tuning Index Structures

So far, we have studied the very general problem of computing efficient read schedules for multi-page requests. This problem occurs in different areas of computer science such as computer architecture, operating systems and database systems. In this chapter, the benefits of efficient read schedules will be demonstrated for query processing in a database system.

There are many operations in a DBS which may exploit multi-page requests and efficient read schedules. Weikum [Wei89] has shown in various experiments that large objects can be read from magnetic disk more efficiently when multiple pages are transferred in a single request. This is also called set-oriented I/O. Zheng and Larson [ZL92] found that external sorting consistently performs better (in comparison to traditional approaches) when the required pages are read following a well-computed schedule. Brinkhoff and Kriegel [BK94] have exploited multi-page requests for processing spatial joins.

In this chapter, we focus on the problem of evaluating *data-intensive* selection queries on a file using an index structure. In contrast to an exact match query (on a primary key), a data-intensive selection query is expected to be satisfied by several records in the file. In our discussion on query optimization, we showed that I/O performance of complex queries largely depends on how efficiently selection queries (operators) are performed. The importance of supporting this type of query efficiently is also demonstrated by the fact that database benchmarks [Gra91] put their emphasis on these queries. Range queries are one of the most frequently used selection queries.

The B^+-tree is the most important index structure, we can even say data structure, in a

DBS especially designed for supporting range queries on a dynamicly growing and shrinking file. Any commercial DBS offers B^+-trees for indexing records. In the following, it is assumed that the reader is familiar with the concept of B^+-trees. Descriptions of B^+-trees can be found in various textbooks, see [Wie88] for example. In our discussion, a B^+-tree implements a so-called sparse index. The records of the file are stored, with respect to their order, in the leaf nodes of the B^+-tree. These index structures are also termed *local order preserving* [HSW88], because the proximity of records is preserved in a data page, but not beyond a page. Since each node in the B^+-tree directly corresponds to a page on secondary storage, we also call the leaf nodes and the branch nodes data pages and directory (index) pages, respectively.

There are many implementations of B^+-trees, but we are not aware of any which make use of efficient read schedules. There are two reasons that may give an explanation. First, most of the previous studies measure performance of range queries by simply counting the number of page accesses. Under such a cost model, the read schedule does not influence performance and therefore, it is not considered in these studies. Second, other studies have differentiated the cost for a page access into positioning time and transfer time. VSAM [KL74], for example, reduces positioning time by clustering data pages, whose records are close to each other, in a preallocated partition of the disk (e.g. a cylinder). When a new page is required and no space is left in the partition, a new partition will be allocated. The first half of the pages remains in the old partition, whereas the other half is moved to the newly allocated partition. Obviously, clustering of data results in reducing the seek time. However, as shown previously, query performance is not solely determined by clustering, but there is still a great potential for improving seek time and rotational delay by using efficient read schedules. Similarly to VSAM, the SB-tree [O'N92] is a modified B^+-tree with the object of supporting data-intensive range queries efficiently.

In the following, we present a generally applicable approach to improve the performance of a broad class of index structures. Our approach is easy to implement assuming that the underlying system offers multi-page requests and possibilities to cluster pages in a file. For the sake of concreteness, we base the presentation in this thesis to B^+-trees; it is implicit how to apply our approach to other methods, e.g. R-trees. Our variant of the B^+-tree is termed CB^+-tree. The CB^+-tree exploits clustering similar to VSAM, as well as efficient read schedules.

In comparison to VSAM, the CB$^+$-tree offers an improved approach to preserving clustering dynamically. The reorganization process of a cluster is distributed among several insertion and deletion operations such that the cost of these operations is only slightly higher than the cost of the same operation executed on an ordinary B$^+$-tree. Moreover, the accumulated cost of clustering (in terms of page accesses) can almost be neglected in comparison to the cost of building up the index. In addition to clustering, the CB$^+$-tree exploits efficient read schedules for an efficient processing of range queries and other data-intensive selection queries.

The remainder of the chapter is organized as follows. In the next section, our work is motivated by following a simple example. The design of the CB$^+$-tree is given in section 2. In section 3, we introduce an abstract data structure that allows us to implement clustering without dealing with the physical properties of the underlying disk. In section 4, we present an approach for improving the local page layout in a cylinder of the CB$^+$-tree. In section 5, we report the results of an experimental performance comparison. In section 6, our approach is compared with an alternative approach. Finally, a summary on the most important results is given in section 7.

8.1 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 8.1. 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 8.1, the disk consists of five cylinders. The parameters of the cylinders are given as follows: $TC = 4$, $PT = 4.25$ and $HST = 0.25$.

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 according to the number of required data pages, 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 8.1 already shows that the number of page accesses is only a coarse 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 8.1: An example of a B^+-tree and its data page layout on disk

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 8.2 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 8.2. 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 8.1.

In the following, we discuss 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

Figure 8.2: The example after all records are inserted

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 clustered not only on 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.

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

8.2 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. Our intention is not primarily the design of a new B$^+$-tree, but to provide efficient techniques that improve the performance of queries for a broad class of access methods. B$^+$-trees are only our running example in this chapter. The section is structured into four subsections. First, the data structure is briefly discussed. Splitting is then explained for directory pages and clusters. Finally, the algorithm for performing range queries is outlined.

8.2.1 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 *bags*. A bag is best compared with a small file which can contain at most c data pages. In general, the pages of a bag 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 bag completely belongs to one cylinder of the disk. Thus, one multi-page request can read all pages of a bag into main memory. Important to our design is that deletions of arbitrary pages and insertions of pages are supported as basic operations on bags. The requirements on the underlying file system will be discussed in more details in section 8.3.

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 bag. 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), bags 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 bag (with c pages). This is called a *bag overflow*. Similar to an ordinary page overflow, a bag overflow is eliminated by splitting the full bag 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.

8.2.2 Splitting of a Bag

The approach to splitting a bag is first a direct adaptation from the splitting algorithm of the B^+-tree. A similar approach has been already proposed for VSAM [KL74]. The basic idea is as follows. First, space for a new bag is allocated on a new cylinder and the second half of the data pages in the full bag is moved to the new bag. Next, the corresponding references in the subpage are updated and the directory page is written back to disk. This is called a *complete reorganization*. In contrast to the original VSAM algorithm, pages which are inserted in the new page are removed from the original bag. Moreover, the new bag occupies only disk space for the new pages (and does not pre-allocate space for c pages). Otherwise, as it can be observed for VSAM, average storage utilization would drop below 50% which is not acceptable for many applications.

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 a 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 bag consists of at least $\lceil c/2 \rceil$ data pages. Instead, it is only required that a bag with less than $\lceil c/2 \rceil$ pages has a left sibling bag such that both have more than c pages. Such a pair of bags is also called a *reorganization pair*. When a bag 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 bag 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 bags. Such a reorganization step is triggered when a new page is inserted in one of the bags of the reorganization pair. Then, the r rightmost pages (with respect to the key-order) are moved from the left bag to the right bag. 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 a decreasing value of r.

8.2.3 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 considers only those possibilities for a split which do not result in cutting a subpage or a pair of subpages which belong to the same 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.

8.2.4 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 8.1. 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

Figure 8.3: Double buffering for performing range queries in a CB^+-tree

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 bag.

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 bags (subpages). First, the (qualifying) data pages of the first bag are read into the first buffer. Then, while the pages of the second bag 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 bag 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 8.3. 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 8.3 shows that page H is already in the first buffer, whereas pages G and I still have to be retrieved.

8.3 File System Support for Bags

Our discussion on CB^+-trees is based on the following two assumptions:

- For a split of a data page, it is always possible to allocate a new page on the same cylinder where the other pages of the bag are located.
- Whenever a bag is split into two, we can find a cylinder with $\lceil c/2 \rceil$ free pages.

In general, however, these assumptions are not fulfilled. The reason is simply that an almost full disk does not give any freedom in clustering data. In practice, the relationship between disk storage utilization and performance is already observed. A rule of the thumb [Gel89] is therefore to keep only 50% of the disk space occupied.

In order to present a complete solution, we introduce another level of abstraction between the file system and the disk system. In our approach, a file consists of several *logical cylinders*, also called *bags* previously, which the underlying file system assigns to one or more physical cylinders of the disk. Whenever it is possible (in case of low disk occupancy), a bag should be kept on only one physical cylinder. An interface for dealing with bags is made available to a programmer. In contrast to ordinary file processing, the management of files with bags is different with respect to the following operations.

First of all, when a file is created, the maximum size of a bag is given as an additional parameter to the file system. In addition, a bag is created by default. The size of a bag is required to be below a threshold that depends on the underlying disk.

If a new page is allocated in the file, the following strategy will be employed. In addition to the file handle, the identifier of the desired bag is given as a parameter to the file system. If possible, the file system allocates a new block on one of the physical cylinders where the pages of the bag are kept. Otherwise, a new physical cylinder will be selected which is adjacent to the other physical cylinders of the bag. One obvious requirement for allocating a new page is that the bag is not already full.

In addition to pages, a file can also be expanded by a new bag. In particular, a new bag is necessary when a new page is allocated, but all bags are already full. A new bag is located on an arbitrary physical cylinder that contains a large number of free blocks. Thus, physical clustering is only preserved in a bag, but not beyond a bag.

In order to maintain clustering in dynamic files, a function is provided for copying and moving pages from one bag to another. Such a function, for example, can be used for implementing the split policy of the CB^+-tree.

The concept of bags is an elegant way to avoid dealing with physical dependencies, without losing control on clustering. Whenever possible, pages of a bag (logical cylinder) are assigned to a single physical cylinder. If not, a few contiguous cylinders will keep the pages of a bag. Moreover, the programmers of an index structure can control the assignment of pages to bags, which is particularly suitable for dynamic files.

8.4 The Organization of Pages in a Cylinder

Up to now, we have only studied the problem of mapping the pages of the CB^+-tree to cylinders. The local organization of pages in a cylinder, however, also has a substantial impact on query performance. Let us consider an example where a range query supported by a CB^+-tree requires two pages from a cylinder. When the pages are on different tracks and one page is above the other page (i.e. on the same sector), more than two disk revolutions are required for reading these pages, in the worst case. Otherwise, when pages do not overlap, only two page transfers are required in the best case and a full revolution is required in the worst case. This simple example already gives the basic idea how pages should be organized in a cylinder.

For the sake of simplicity, let us assume a disk that follows the idealized disk model (i.e. alignment of pages, no head switch time). Hence, only the column (sector) where the page is stored influences performance, but not its track. Let PT denote the number of columns on a cylinder. Consider a subpage of the CB$^+$-tree with c references to data pages. The layout of the data pages would be optimal if the i-th page of a container, $1 \leq i \leq c$, is assigned to the (i % PT)-th column. Then, when a range query requires n, $1 \leq n \leq c$, of the c data pages, the transfer time of the request is minimal, i.e. n page transfers. This approach is very beneficial for static files. For dynamic files, it would be very expensive to maintain the proposed ordering of the data pages on a cylinder. When the i-th data page, $1 \leq i \leq c$, is split into two, the j-th page has to be copied from column j % PT into column $(j+1)$ % PT for all $j \in \{i, i+1, \ldots, c\}$.

In order to avoid expensive reordering of pages on a cylinder, we present an alternative approach. This approach [SL91] has already been proposed for distributing pages of a B$^+$-tree over a set of magnetic disks. The new algorithm is motivated by two goals:

- In order to reduce the cost for large range queries (i.e. large multi-page requests), it is sufficient to distribute the data pages evenly over the columns of a cylinder. As a side effect, such an even distribution also reduces the expected cost for multi-page requests of arbitrary size.
- In order to reduce the cost of small range queries, a new page is assigned to a column that does not contain a nearby page. Nearby here means within a window of size $PT-1$ or $PT-2$ pages centered around the new page.

These requirements should be fulfilled without reorganizing pages, i.e. once a page is assigned to a column, it will stay there forever. The algorithm for finding an appropriate column for a new page is as follows.

Algorithm FindColumn(page);

1. $CS := \{0, .., PT-1\}$; left := page; right := page;
2. FOR i := 1 TO $\lceil \frac{PT}{2} - 1 \rceil$ DO
   right := RightPage(right);

left := LeftPage(left);

$CS := CS \setminus \{$ Column(left), Column(right)$\}$;

END;

3. RETURN ($\min\{i | load[i] = \min_{j \in CS} load[j]\}$);

END FindColumn;

The page for which we require a column is used as the input parameter of the algorithm *FindColumn*. In case of a split, *FindColumn* 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 columns which can be used for storing the pages. The main part of the algorithm is in the second step. All columns 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 in a container, respectively. If there is no neighboring page in the same container, the address of the given page is assumed to be returned. The function *Column* returns the column number of the input page. If the input page is not currently assigned to a column, *Column* returns some value greater than PT. In step three, ties are resolved. The array *load* is assumed to contain the number of pages assigned to each column in a cylinder. The columns with the lowest load are first selected and, if there is more than one such column, the one with the lowest column number is chosen. The functions *LeftPage* and *RightPage* can be easily implemented by investigating the pointers in the subpage of the CB$^+$-tree index. Since a subpage is kept on a single directory page, no additional page request is required.

An example subtree of a CB$^+$-tree is shown for $PT = 5$ in Figure 8.4. We assume that the capacity is three and ten for data pages and subpages, respectively. The number on top of a data page indicates on which column the page is stored. (This information is stored as part of the pointer referring to the page.) The subtree of the CB$^+$-tree in Figure 8.4 corresponds to the ideal case where the load is optimally balanced over the columns. Every column (except the last) contains the same number of pages and, for any range query, the transfer cost of the multi-page request is minimal.

However, this ideal situation cannot be preserved under further insertions without a global

Figure 8.4: Example for distributing data pages on a cylinder with 5 columns

Figure 8.5: The example after split of page D

reordering of the pages. 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 *FindColumn* is called. After step one of *FindColumn* we have: $CS = \{0, 1, 2, 3, 4\}$, $left =$ J and $right =$ J. The loop will be executed twice, yielding $left =$ C, $right =$ F and $CS = \{1\}$. The new page is thus assigned to column one. Figure 8.5 shows the resulting subtree.

Theorem 8.4.1 *If pages have been assigned to columns using algorithm FindColumn and there have been no deletions, then, for $PT < c$, no two pages of a contiguous subsequence of $\lceil \frac{PT}{2} \rceil$ or fewer data pages in a container are stored on the same column.*

Proof: It is easy to verify that, if the CB$^+$-tree consists of at most PT data pages, the algorithm will assign each page to a different column. In this case, the theorem is trivially true.

Now assume that the file consists of more than PT data pages and consider the assignment of a new page in a given container. 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 column in such a way that none of the $\lceil \frac{PT}{2} - 1 \rceil$ pages to the left and none of the $\lceil \frac{PT}{2} - 1 \rceil$ pages to the right of the new page are stored on the same column of the container. All contiguous subsequences of at most $\lceil \frac{PT}{2} \rceil$ 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 minimum number of disk revolutions we can expect for s qualifying data pages in a container is $\frac{s}{PT}$. Our CB$^+$-tree is close to that, even in the worst case, which is expressed in the following corollary.

Corollary 8.4.1 *Let q be a range query. For a given container, let s, $s \leq c$, be the number of data pages required by the range query. If pages have been assigned to columns using algorithm FindColumn and there have been no deletions, then an upper bound for the number of disk revolutions is given as follows:*

$$\lceil \frac{s}{\lceil \frac{PT}{2} \rceil} \rceil$$

Our CB$^+$-tree guarantees that, for range queries that retrieve no more than $\lceil \frac{PT}{2} \rceil$ data pages from a cylinder, no column will be accessed more than once. In other words, less than one disk revolution is required. For range queries, which require more than $\lceil \frac{PT}{2} \rceil$ data pages we can guarantee that the number of disk revolutions required for reading the target pages of a container is not more than twice the optimal number. 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 8.4.1 is satisfied. We can apply the same solution proposed for the multi-disk B$^+$-tree to the CB$^+$-tree. The interested reader is therefore referred to [SL91].

8.5 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 4.2.

8.5.1 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

	#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 8.1: Performance for creating some B^+-trees and CB^+-trees

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 8.2. The capacity of the bags 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 8.1 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). Recall that bags occupy disk space only for page which actually contains data records of the CB^+-tree. This property is not fulfilled for the extents in VSAM. 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.

8.5.2 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 thesis, 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 bags which include at least one of the required data pages. If l, $1 \leq l \leq c$, pages of the bag 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

$$\frac{u * seek_{avg} + \sum_{j=1}^{c} v_j * Cost(j)}{q} \tag{8.1}$$

Here $Cost(j)$ corresponds either to the cost function of the idealized disk model (see formula 5.15 on page 80) or to the one of the model that considers head switch time (see formula 6.31 on page 119).

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 bags	c	32
size of reorganization step	r	$\lceil c/2 \rceil$
query size	α	0.5%
number of records		100,000

Table 8.2: Parameters of the disk and the CB$^+$-tree

In the following sets of experiments, the disk parameters are specified as given in the first rows of Table 8.2. 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 8.2 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 default values.

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 8.6. 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

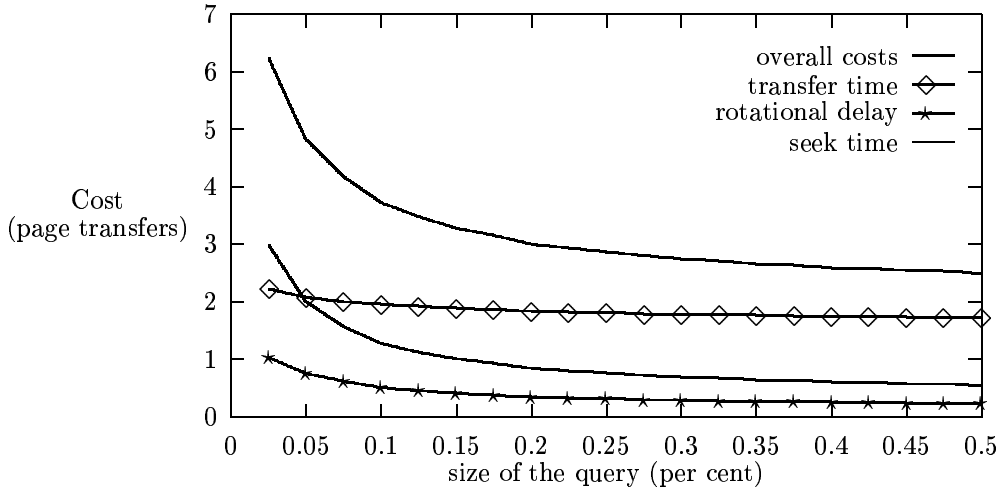

Figure 8.6: Cost for retrieving a qualifying page of a range query (as a function of the query size)

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.

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.

Varying the capacity of the bags

In our second set of experiments, results vary with c, the capacity of the bags (subpages). A split of a bag was performed in a single reorganization step (i.e. $r = \lceil c/2 \rceil$). 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 8.7, the cost has been depicted for retrieving

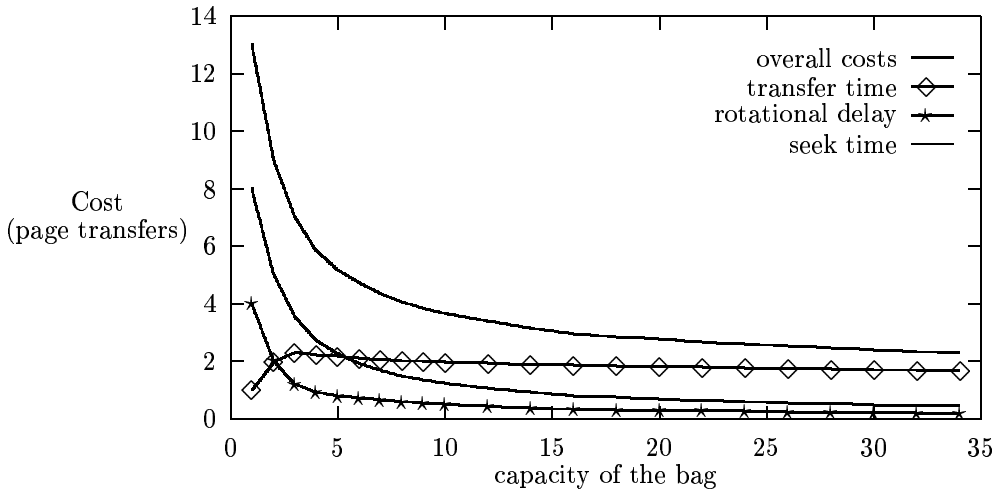

Figure 8.7: Cost for retrieving a qualifying page of a range query (as a function of the capacity of the bags)

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 bags 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 8.7. 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

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

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.

Varying the size of the reorganization steps

The results of the previous experiments are obtained from a CB$^+$-tree whose bags 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 8.3. 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 of r.

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 8.8, 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 time (hst) which varies from 0 to 4 ms. Disks with head switch time higher than 4 ms are seldom found in

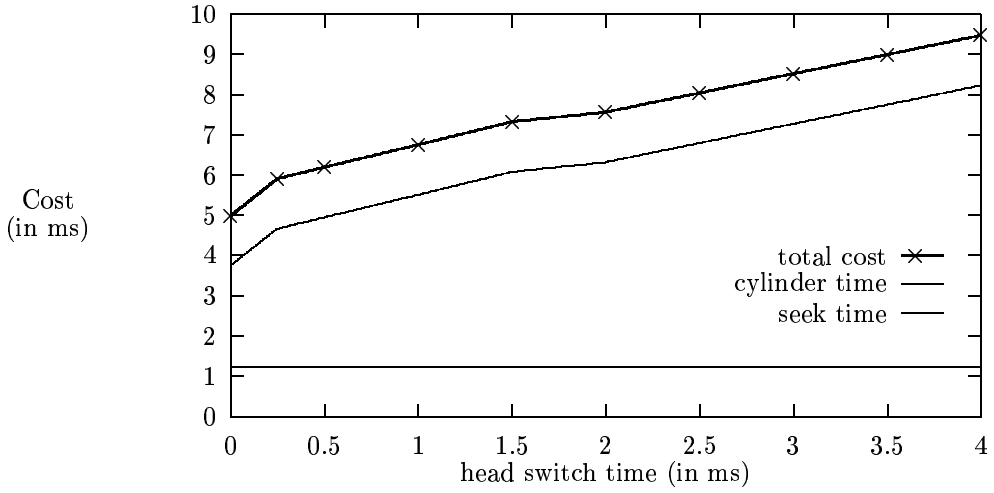

Figure 8.8: Cost for retrieving a qualifying page of a range query (as a function of the head switch time)

practice. There are three graphs in Figure 8.8. 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.

The effect 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 distribution of pages on a cylinder does not have any influence on the seek time. Thus, a lower bound for the expected cost of range queries is given by formula 8.1, when $Cost(j) = 1$, $1 \leq j \leq PT$. The formula can then be modified to

$$\frac{u * seek_{avg}}{r} + 1$$

We will refer to this formula as the *lower bound*.

In order to improve the local organization of pages in a cylinder, we investigated the following two strategies. The first strategy does not distribute the pages over all of 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. The second strategy followed the approach presented in section 8.4. This strategy is called *balanced assignment* in the following.

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 Fig 8.9, results are plotted varying in the size of queries from 0.025% to 0.5%. Two of the graphs depict the results when the first 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, this strategy gives best performance for $t = 4$. [2] A third graph shows the results obtained from our strategy of balanced assignment. A fourth graph depicts the lower bound of the cost.

[2] Notet that the B$^+$-tree with large pages (see our discussion in the next section) cannot give better query performance than the CB$^+$-tree in combination with this strategy for $t = 4$.

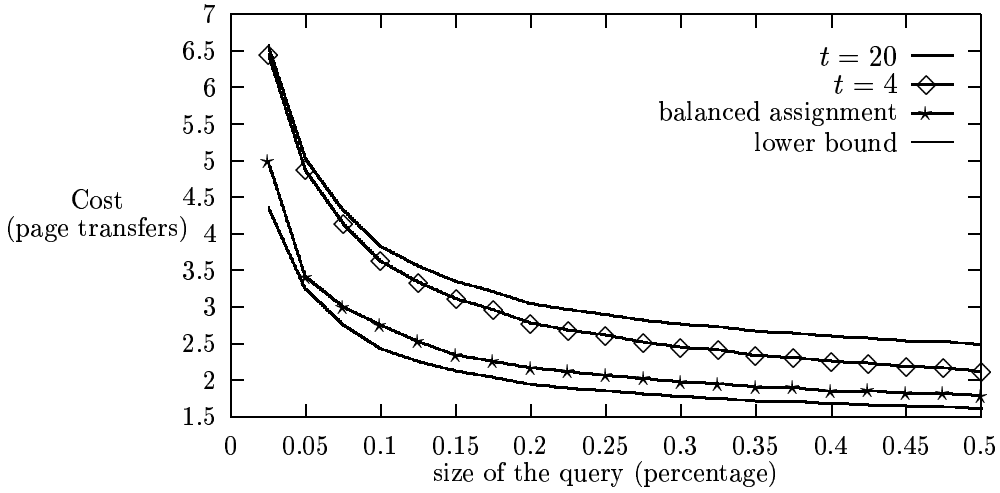

Figure 8.9: Expected cost of range queries for different methods of clustering pages locally in a cylinder

The results presented are interesting. 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 might be found. For larger range queries, the performance is slightly better for $t = 4$. Our strategy of balanced assignment already shows for small range queries that it is close to the lower bound. For queries retrieving 0.1% of the records, the strategy of balanced assignment is about 25% faster than the other methods. The difference between balanced assignment and the lower bound seems to be almost independent of the size of the query.

8.6 Large Pages: An Alternative to Clustering?

In order to support range queries efficiently, another approach can be pursued for B^+-trees that seems to be much simpler than the one presented in the previous sections. The basic idea is to use *large data pages*, see [Lom88]. The advantage of large pages are that the number of index pages is considerably reduced and that large range queries are supported

Figure 8.10: An example for a partition of an R-tree and for a window query

very efficiently.

For a range query that requires n data pages, every record in $n-2$ of the n data pages, $n \geq 2$, satisfies the range condition, and only for the other two pages records might be found that do not satisfy the query. Such records are also called *false drops*. Now, increasing the page size, say by a factor of d, reduces the number of seek operations almost by a factor of d, whereas the transfer time slightly increases because the number of false drops is increasing as well. However, it is always guaranteed that at most two pages contain false drops. Obviously, small range queries are performed less efficiently when the page size is large. Moreover, a small range query occupies a (relatively) large fraction of the buffer. Also, in comparison to an ordinary B^+-tree, more buffer space is required for an insertion and a deletion of a record. In addition, other disadvantages occur when a B^+-tree with large pages is integrated in a DBS. In general, a DBS supports only one page size and therefore, a B^+-tree would not have been able to use a different page size.

In general, a DBS provides the user with several index structures and each of these index structures should take advantage of tuning techniques, particularly from large pages. Of particular interest are multi-dimensional index structures. One of the most important multi-dimensional index structures are R-trees [Gut84], particularly R*-trees [BKSS90]. It is assumed here that the reader is familiar with the basic concepts of R-trees. There are great similarities between R-trees and B^+-trees with respect to splitting a page. Therefore, we do not see any difficulties for incorporating our approaches into R-trees.

In contrast to B^+-trees, a multi-dimensional index structure cannot considerably improve the performance of range queries by simply increasing the page size. A multi-dimensional index structure partitions the data space into regions. A data page is associated with a region such that all multi-dimensional records stored in the data page must be in the corresponding region. For most multi-dimensional index structures, regions correspond to rectilinear rectangles. An example for a partition is given on the left hand side in Figure 8.10. This example shows a partition of an R-tree. In our example, there are 7 regions $R_1, \ldots, R_7$ and so the R-tree consists of 7 data pages. As illustrated, R-trees allow that regions have a common intersection.

The *window query* is the most common query which is supported by an R-tree. A window query, also called multi-dimensional range query, searchs for records that are in a multi-dimensional rectilinear rectangle (window). In order to perform a window query, an R-tree proceeds as follows. For each region that intersects the window, the corresponding page is read from secondary storage. Thereafter, each record of that page is checked for being in the window or not. For example, a window query is illustrated on the right hand side of Figure 8.10. The query window illustrated as a gray rectangle intersects with regions R_2, R_4, R_5, R_6, R_7. Thus, 5 pages have to be retrieved from secondary storage. Only the page associated with region R_7 does not contain false drops, whereas all the other pages contain false drops. In general, for a given query, false drops can be found in every target page. As a consequence, increasing the page size does not necessarily improve the query performance since the number of target pages may not decrease. Moreover, large pages can result in performance loss since the number of false drops increases with an increasing page size.

In summary, the use of large pages is an interesting alternative to our approach. In particular, large range queries can take advantage of that approach since the number of seeks is expected to be substantially reduced. However, there also several drawbacks when index structures exploit large pages. First, there is a high consumption of buffer space, particularly for small queries. Second, the performance improvements of multi-dimensional index structures is expected to be low due to an increase in the number of false drops. Finally, it is difficult to integrate the approach of large pages into a DBS.

8.7 Summary

In this chapter, we dealt with the design of index structures that exploit clustering and efficient read schedules. Although emphasis was put on B^+-trees, we want to stress that the proposed techniques are generally applicable. In particular, multi-dimensional index structures may also take advantage of these techniques.

A new type of B^+-trees was proposed, called a CB^+-tree. The CB^+-tree behaves very similarly to the B^+-tree with respect to insertions and deletions of records, but it offers substantially better performance for range queries. In contrast to previous approaches, clustering of data pages is dynamically maintained. Moreover, it is almost given for free in the CB^+-tree. In addition to clustering, we showed that efficient read schedules are essential to an efficient processing of range queries. In order to obtain hardware independence, we proposed a new concept, called bag (logical cylinder), that allows programmers of index structures (and other disk data structures) to cluster its data without dealing with physical constraints (e.g. data layout on disk). Special attention was given to the local layout of pages of the CB^+-tree on a physical cylinder. We presented a method that balances the load among the columns of a cylinder and that keeps adjacent pages on different columns.

A performance comparison based on a simulation demonstrated that range queries are performed several times faster when the CB^+-tree is used, in comparison to ordinary B^+-trees.

Chapter 9

Conclusions and Future Work

The success of DBSs depends not only on the degree of hardware independence, but also on the ability to translate queries into an equivalent sequence of operations which are efficiently implemented on the underlying hardware. The problems of how to find an optimal execution plan and how to implement the basic operations have created an important and fertile area for DBS research. Due to the continued development of the hardware components, queries suffer more and more under a severe I/O bottleneck and therefore it is of vital importance to counteract the bottleneck on all implementation levels of a DBS.

In order to improve the I/O performance of today's DBSs, this thesis has stressed the importance of modeling magnetic disk drives more accurately than is generally the case. In particular, we showed that data-intensive queries, which frequently occur in engineering applications, can be improved by using multi-page requests rather than reading one page at a time. Reading multiple pages in a single request is, of course, not a new idea, but so far, it was not studied, analyzed and implemented in its full generality. Only a special and limited form of multi-page request, known as prefetching, is generally accepted as a method for improving the I/O performance of a computer system. In order to incorporate multi-page requests into a DBS it is not sufficient to only design algorithms, but to derive cost formulas which are required by the query optimizer to find a good execution plan.

The main contributions of the thesis are several analyses of the cost of retrieving a given number of qualifying pages (so-called target pages) from a magnetic disk drive. The various analyses are different with respect to both algorithms and the underlying disk models. The

first analysis has assumed a disk as a linear sequence of pages. Due to its simplicity, we were able to integrate the size of the buffer into our model. We showed that performance improves substantially although buffers are rather small (i.e. about 10 pages). Larger buffers still result in improved response times, but at a much slower rate. Despite its simplicity, the practical importance of the approach has been demonstrated in [BK94] for reading large objects.

The second analysis was performed under a much more complicated disk model which takes into account the disk components, namely cylinders and tracks. Moreover, the access time of a page depends on the position of the page previously read. In contrast to our first model, we assumed an infinitely large buffer into which the disk pages are transferred. Moreover, the model assumes that a head switch causes no time delay. When target pages are on a cylinder, we derived the exact cost formula which unfortunately requires the solution of two complex recurrence relations. This suggests that exact cost formulas will be very difficult to obtain for a more complex disk model.

The third analysis provides the most important contribution of the thesis. The underlying disk model is rather complex, and in particular, includes track skewing and head switch times greater than zero. A simulation showed that the model gives almost the same results as the ones observed on a real disk. Despite its complexity, the derived cost function is simple to compute, requiring only a few arithmetic calculations. Moreover, the results of the cost function were shown to be in excellent agreement with the results obtained from our simulation.

The approach of multi-page requests is then applied to improve the performance of the B+-tree. We designed a new variant of the B+-tree, called CB+-tree, that offers the property of clustering adjacent pages close to each other on disk. The combination of both clustering pages and using multi-page requests results in a new approach to index structures. Data-intensive queries such as range queries can be performed much more efficiently on a CB^+-tree than on a B^+-tree. Several experimental comparisons demonstrated the advantages of the CB^+-tree over the B^+-tree. The approach used for improving the B^+tree can be directly applied to other index structures such as the R-tree, one of the most important structures for indexing multidimensional spatial data.

The three analyses and the new approach to index structures represent important con-

tributions to the development of a completely new approach to query processing in a DBS. Instead of reading one page at a time, we expect that future DBSs will perform their data-intensive queries using multi-page requests. The contributions of this thesis are basic to the effective use of magnetic disk drives and provide a foundation for further study not solely limited to DBSs. In comparison to other approaches for I/O optimization such as disk arrays and caches, the technique of multi-page requests is an algorithmic solution and therefore, it will not increase the cost of a system when it is introduced. However, we believe that only a combination of all these approaches might prevent the threatening I/O crisis.

Future Research

The very general approach of multi-page requests creates a rich source of research problems. Some of them are discussed below.

One direction of future research is to investigate the impact of multi-page requests on the replacement policy of a buffer. In order to find the next page for replacement, the policy generally uses the last reference (sometimes the last few references) to pages without considering the actual cost of reading these pages. This might be a reasonable solution when pages are read one at a time. However, when a page is read as one of several pages in a multi-page request, the cost of reading the page is low. Therefore, the priority of keeping such a page in the buffer should be lower than for those pages which are read one at a time.

Other promising research problems are related to exploiting multi-page requests on optical and magneto-optical storage devices. Such devices have some unique characteristics and advantages that make them very attractive storage devices for the future.

Moreover, another interesting research question is about the impact of multi-page requests on the design of disk arrays. So far, the only operations considered for the evaluation of disk arrays are the ones that read (write) a logically contiguous sequence of bytes. However, such an access pattern is seldom found when data-intensive queries are supported in large databases. In particular, this holds when queries are evaluated by using multi-page requests.

Abbreviations

bpm	bits per millimeter
CAD	computer aided design
CPU	central processing unit
DBS	database system
DRAM	dynamic random access memory
I/O	Input/Output
KB	2^{10} bytes
LRU	least recently used
MB	2^{20} Bytes
ms	10^{-3} seconds
MTTF	mean time to failure
MTTR	mean time to repair
NVS	non-volatile storage
OS	operating system
RAID	redundant and inexpensive (independent) disks
rpm	(disk) rotations per minute
RPS	rotational positioning sensing
SLTF	shortest latency time first
SSD	solid state disk
SSTF	shortest seek time first
tpm	tracks per millimeter
VSAM	virtual storage access method
μs	10^{-6} seconds

Index

Bibliography

[ABC+76] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, jun 1976.

[AC75] M.M. Astrahan and D.D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, oct 1975.

[AG92] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, 1992.

[BG88] D. Bitton and J. Gray. Disk shadowing. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 331–338, 1988.

[Bil92] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. of the ACM SIGMOD*, pages 276–285, 1992.

[BK94] T. Brinkhoff and Hans-Peter 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.

[BP88] A. J. Borr and F. Putzolu. High performance SQL through low-level system integration. In *Proc. of the ACM SIGMOD*, pages 342–349, 1988.

[BS90] P. A. Buhr and R. A. Stroosbosscher. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software - Practice and Experience*, 20(9):929–964, 1990.

[CABK88] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in Bubba. In *Proc. of the ACM SIGMOD*, pages 99–108, June 1988.

[Car75] A. F. Cardenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, 1975.

[CD85] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 127–141, 1985.

[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.

[CHMWS87] P. Christmann, T. Härder, K. Meyer-Wegener, and A. Sikeler. Which kinds of OS mechanisms should be provided for database management. In J. Nehmer, editor, *Expiriences with distributed systems*, pages 213–252. Springer-Verlag, 1987.

[Chr84] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, jun 1984.

[CJL89] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 397–410, 1989.

[CK89] E. E. Chang and R. H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In *Proc. of the ACM SIGMOD*, pages 348–357, 1989.

[CKB89] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. *IBM System Journal*, 28(1):62–76, 1989.

[CKKS89] George Copeland, Tom Keller, Ravi Krishnamurthy, and Marc Smith. The case for safe RAM. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 327–335, 1989.

[CKR72] E. G. Coffman, L. A. Klimko, and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal of Computing*, 1(3):269–279, 1972.

[CKV93] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. of the ACM SIGMOD*, pages 257–266, 1993.

[CLSW84] J. M. Cheng, C. R. Loosley, A. Shibamiya, and P. S. Worthington. IBM Database 2 performance: design, implementation, and tuning. *IBM System Journal*, 23(2):189–210, 1984.

[Dei90] H. M. Deitel. *Operating Systems*. Addison-Wesley, 1990.

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

[DF82] L. W. Dowdy and D. V. Foster. Comparative models for the file assigment problem. *Computing Surveys*, 14(2):287–313, 1982.

[EH84] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, 1984.

[Fra69] H. Frank. Analysis and optimization of disk storage devices for time-sharing systems. *Journal of the Association of Computing Machinery*, 16(4):602–620, 1969.

[Ful74] H. Fuller. Minimal-total-processing-time drum and disk scheduling disciplines. *Communications of the ACM*, 17(7):376–381, 1974.

[GD87] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, 1987.

[Gel89] J. P. Gelb. System-managed storage. *IBM System Journal*, 28(1):77–103, 1989.

[GHW90] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 148–159, 1990.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

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

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Comp. Surveys*, 25(2):73–170, 1993.

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

[Här87] T. Härder. Realisierung von operationalen Schnittstellen (in german). In P.C. Lockemann and J.W. Schmidt, editors, *Datenbank Handbuch*, pages 167–342. Springer Verlag, 1987.

[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.

[HD90] H. Hsiao and D. J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proc. of the IEEE Conference on Data Engineering*, 1990.

[HGPG92] Robert Y. Hou, Gregory R. Ganger, Yale N. Patt, and Charles E. Gimarc. Issues and problems in the I/O subsystem, part I — The magnetic disk. In *Proceedings of the 25th Annual Hawaii Conference on System Sciences*, pages 48–57, 1992.

[Hoa85] A. S. Hoagland. Information storage technology – a look at the future. *IEEE Computer*, pages 60 – 67, 1985.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1990.

[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.

[IC91] Y.E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD*, pages 268–277, 1991.

[JCL90] R. Jauhari, M. J. Carey, and M. Livny. Priority hints: an algorithm for priority-based buffer management. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 708–721, 1990.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comp. Surveys*, 16(2):111–152, 1984.

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

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. of the ACM SIGMOD*, pages 148–157, 1991.

[Kin90] Richard P. King. Disk arm movement in anticipation of future requests. *ACM Transactions on Computer Systems*, 8(3):214–229, 1990.

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

[Kol78] J. G. Kollas. An estimate of seek time for batched searching of random and index sequential files. *Computer Journal*, 21(2):132–133, 1978.

[Lom88] D. Lomet. A simple bounded disorder file organization with good performance. *ACM Transactions on Database Systems*, 13(4):525–551, 1988.

[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.

[MC93] J. Menon and J. Cortney. The architecture of a fault-tolerant cached RAID controller. In *Proc.IEEE*, pages 76–86, 1993.

[McF90] Ronald G. McFadyen. *Sequential Access in Files used for Partial Match Retrieval.* PhD thesis, University of Waterloo, 1990.

[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.

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

[OOW93] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the ACM SIGMOD*, pages 297–306, 1993.

[Ouc78] N. K. Ouchi. Systems for recovering data stored in failed memory. Technical Report #4,092,732, US Patent, 1978.

[OV91] M. Tarmer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[PBD93] C. A. Polyzois, A. Bhide, and D. Dias. Disk mirroring with alternating deferred updates. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 604–617, 1993.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD*, pages 109–116, June 1988.

[PSS+87] H.-B. Paul, H.-J. Schek, M. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt Database Kernel System. In *Proc. of the ACM SIGMOD*, pages i196–207, 1987.

[PZ91] M. Palmer and S. B. Zdonik. FIDO: A cache that learns to fetch. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 255–264, 1991.

[Qua92] Quantum Cooperation. *ProDrive 700/1050/1255 Product Manual*, 1992.

[Rah92] Erhard Rahm. Performance evaluation of extended storage architectures for transaction processing. In *Proc. of the ACM SIGMOD*, pages 308–317, 1992.

[Ram87] M. V. Ramakrishna. Computing the probability of hash table / urn overflow. *Comm. in Statistics - Theorey and Methods*, 16(11):3343–3353, 1987.

[RW93a] C. Ruemmler and J. Wilkes. Modeling disks. Technical Report HPL-93-68 (revision 1), HP Laboratories, 1993.

[RW93b] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Technical Report HPL-92-152 (revision 1), HP Laboratories, 1993.

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

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD*, pages 23–34, Boston, MA, 1979. acm.

[Sci93] Science, August 1993.

[SCO90] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter USENIX Conference*, pages 313–323, 1990.

[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.

[SG76] B. Shneiderman and V. Goodman. Batched searching of sequential and tree structured files. *ACM Transactions on Database Systems*, 1(3):268–275, 1976.

[SGH90] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proc. IEEE*, pages 64–75, 1990.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *Proc. of the IEEE Conference on Data Engineering*, pages 336–342, 1986.

[SL91] B. Seeger and P.-Å. Larson. Multi-disk B-trees. In *Proc. of the ACM SIGMOD*, pages 436–445, 1991.

[SLM93] B. Seeger, P.-Å. Larson, and R. McFadyen. Reading a set of disk pages. In *Proc. of the Conf. on Very Large Databases (VLDB)*, pages 592–603, 1993.

[Smi76] A. J. Smith. Sequentiality and prefetching. *ACM Transactions on Database Systems*, 8(3):223–247, 1976.

[Smi81] A. J. Smith. Input/output optimization and disk architectures: A survey. *Performance and Evaluation*, 1:104–117, 1981.

[SO90] J. A. Solworth and C. U. Orji. Write-only disk caches. In *Proc. of the ACM SIGMOD*, pages 123–132, 1990.

[SS86] G. M. Saco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, 1986.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[TG84] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM System Journal*, 23(2):211–218, 1984.

[TP72] Toby J. Teorey and Tad B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177 – 184, 1972.

[Tri79] K. S. Trivedi. An analysis of prepaging. *Computing*, 22:191–210, 1979.

[Wat76] S. J. Waters. Hit ratio. *Computer Journal*, 19(1):21–24, 1976.

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

[WGP94] B.L. Worthington, G.R. Ganger, and Y.N. Patt. Scheduling for modern disk drives and non-random workloads. Technical Report CSE-TR-194-94, University of Michigan, Ann Arbor, 1994.

[Wie88] Gio Wiederhold. *File Organization for Database Design*. McGraw-Hill, 1988.

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

[Woo90] R. Wood. Magnetic magabits. *IEEE Spectrum*, pages 32–38, 1990.

[WSZ91] G. Weikum, P. Scheuermann, and P. Zabback. Dynamic file allocation in disk arrays. In *Proc. of the ACM SIGMOD*, 1991.

[Yao77] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, 1977.

[ZL92] L. Q. Zheng and P.-Å. Larson. Speeding up external mergesort. Technical Report CS-92-40, Comp. Science Department, University of Waterloo, 1992.