Query Processing in Tertiary Memory Databases

by

Sunita Sarawagi

Bachelor of Technology, Indian Institute of Technology, Kharagpur, 1991
Master of Science, University of California, Berkeley, 1993

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Michael R. Stonebraker, Chair
Professor Joseph M. Hellerstein
Professor Arie Segev

1996

The dissertation of Sunita Sarawagi is approved:

Chair Date

Date

Date

University of California at Berkeley

1996

Query Processing in Tertiary Memory Databases

by
Sunita Sarawagi

Abstract

Query Processing in Tertiary Memory Databases

by

Sunita Sarawagi
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Michael R. Stonebraker, Chair

This thesis presents the design and implementation of a database query processing engine that is optimized for access to tertiary memory devices. Tertiary memory devices provide a cost-effective solution for handling the on-going information explosion. While cheap and convenient, they pose new optimization challenges. Not only are tertiary devices three orders of magnitude slower than disks, but they also have a highly non-uniform access latency. Therefore, it is crucial to carefully reduce and reorder I/O on tertiary memory using effective query scheduling, batching, caching, prefetching and data placement techniques.

We make two key modifications to an existing query processing architecture to support such aggressive optimizations: The first is a scheduler that uses system-wide information to make query scheduling, caching and device scheduling decisions in an integrated manner. The second is a reorderable executor that can process each query plan in the order in which data is made available by the scheduler rather than demand and process data in a fixed order, as in most conventional query execution engines. The two together provide unprecedented opportunities for optimizing accesses to tertiary memory. We have extended the POSTGRES database system with these optimizations. Measurements on the prototype yielded almost an order of magnitude improvement on the SEQUOIA-2000 benchmark and on queries over synthetic datasets.

We explore data placement techniques on tertiary memory devices to enable better clustering. This thesis concentrates on data placement issues for large multidimensional arrays — one of the largest contributors of data volume in many database systems. We discuss four techniques for doing this: (1) storing the array in multidimensional "chunks"

to minimize the number of blocks fetched, (2) reordering the chunked array to minimize seek distance between accessed blocks, (3) maintaining redundant copies of the array, each organized for a different chunk size and ordering and (4) partitioning the array onto platters of a tertiary memory device so as to minimize the number of platter switches. Measurements on data obtained from global change scientists show that accesses on arrays organized using these techniques are often an order of magnitude faster than on the unoptimized data.

Professor Michael R. Stonebraker
Dissertation Committee Chair

To Sakhu Bai,

and my parents.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisor Mike Stonebraker for his guidance and help throughout my stay at Berkeley. I am greatly indebted to my reader Joey Hellerstein for carefully reading the thesis and providing useful feedback. I would like to thank Professor Arie Segev for reading my thesis in spite of his very busy schedule. Professor Tom Anderson and Dave Patterson served on my qualifying exam committee and I am grateful to them for that.

Life and work at Berkeley would have been incomplete without the company of my wonderful friends and officemates notably Mark Sullivan, Margo Seltzer, Mike Olson, Wei Hong, Jeff Meredith and Carol Paxson in the early part of the PhD process and Paul Aoki, Marcel Kornacker, Andrew Macbridge, Adam Sah, Jeff Sidell and Allison Woodruff in the latter half. Paul Aoki deserves special thanks for providing company and reassurance during the mid-PhD crisis. Friends like Savitha Balakrishnan, Sudha Narayan, Chandramouli Banerjee, Ann Chervenak, Bhaskar Ghosh, Brinda Govindan, Anoop Goyal, Janani Janakiraman, Kamala Tyagarajan and Anu Sundaresan have provided the much needed relaxation and company outside work.

Finally, I cannot forget the constant support and encouragement provided by my family far away. I am especially indebted to the advice and reassurance provided by my parents, my sister Santosh and my father-in-law. I owe starting and finishing my PhD to my husband, Soumen Chakrabarti. His love and respect for knowledge has been the main inspiration for entering the PhD program in the first place.

Chapter 1

Introduction

The ongoing information explosion calls for an overhaul of conventional database systems to support the increasing storage demands of many applications. Applications like EOSDIS [DR91a, Sto91b] are estimated to collect around a petabyte of data per year. This amount of data cannot be stored cost-effectively on magnetic disks [SD91, SSU95]. In view of applications like EOSDIS and other applications like data warehouses [Ome92], image [RFJ+93, OS95] and video storage systems [FR94], there is increasing consensus among database researchers [SSU95, Sto91a, CHL93, Sel93, Moh93] for supporting tertiary memory devices [Ran91]. Not only are all these applications huge, they also require efficient querying and data management facilities, making it necessary to deploy database systems instead of relying on conventional file oriented mass storage systems [N+87, C+82].

A major limitation of traditional DBMSs is the assumption that all data resides on magnetic disk or in main memory. Tertiary memory, if used at all, functions only as an archival storage system to be written once and rarely read. Some database systems [Isa93] allow data to be stored on tertiary memory, but they do so by using a file system to get transparent access to data and store only metadata information in the database system. This means that the tertiary memory is not under the direct control of the database system. One important exception is POSTGRES [Ols92]. POSTGRES includes a Sony optical jukebox [Son89b] as an additional level of the storage hierarchy. The POSTGRES storage manager can move data transparently between a disk cache and the jukebox using a LRU replacement strategy. While this prototype provides the enabling technology for controlling tertiary memory directly from a DBMS, it does not address many of the efficiency related issues for processing queries on tertiary memory.

This dissertation addresses the performance related issues that arise in building a tertiary memory database system. In particular, it deals with two aspects of the problem: processing queries over data residing on tertiary memory and organizing data on tertiary memory for efficient retrieval.

1.1 Background: Tertiary memory devices

A typical tertiary memory device [Ran91] consists of a large number of storage units, a few read-write drives and even fewer robot arms to switch the storage units between the shelves and the drives. Normally, a magnetic disk is used as a cache for staging data in and out of the tertiary memory device. There are a wide variety of tertiary memory devices that differ in capacity, bandwidth, latency and physical configuration (number of drives, recording medium, number of robot arms and number and arrangement of the storage units). We will present a brief taxonomical survey of the existing products.

For most tertiary memory devices, the storage unit, which we generically call a *platter*, is either a tape or a disk. From a performance perspective, the main distinction between the two is that tape-based devices have a predominant seek cost (often two to three orders of magnitude higher than disks) and thus favor sequential access, whereas disk-based devices provide random access much like conventional secondary storage devices. An orthogonal dimension for classifying storage units is the recording medium: magnetic, optical or magneto-optical. Most tapes are magnetic whereas disks are typically optical or magneto-optical. Optical disks can be read-only (CD-ROMs), write-once (WORM, CD-R) or re-writable (CD-E). More recently, optical tapes have also entered the commercial arena chiefly because of their promise of higher reliability and capacity. Tapes are further classified on the recording format: helical-scan versus linear. Helical-scan tapes are known to provide higher storage capacity than linear tapes but at the cost of lower reliability (in terms of mean number of reads before tape failure, as opposed to drive failure). In Table 1.1 we list the important performance characteristics of several existing tape and disk drive products [Lor95, Che94, Son89b, Cora]. Note the wide variation in the transfer rate, access times and capacity of various drives. For instance, the Exabyte 8505XL tape drives has a transfer rate of only 0.5 MB/second whereas the Sony D-1 drives can transfer data at 32 MB/second. Typically, disk-based drives like the CD-ROMs and Magneto-optical drives have lower transfer rates than tape drives. The access time for tapes depends not only on

Storage device	Transfer rate (MB/sec)	Average access time (sec)	Capacity GB	Drive cost $	Media cost $/GB
Magnetic tapes: helical-scan					
Exabyte 8505XL (8mm)	0.5	20	7	1,675	2
Metrum 2150	2	20	18	40,000	1
Sony DIR-1000	32	20	8.7	284,000	1.46
Magnetic tapes: linear					
IBM 3590	9	20.5	10	43,500	7
Quantum DLT 2000	1.25	45	10	3250	4.5
Quantum DLT 4000	1.5	60	20	5000	4.5
Optical tapes					
CREO	3	28	1000	350,000	8.5
Laser Tape 1/2"	6	15	100	25,000	0.7
Optical disks					
sony CDU-541 CD-ROM	1.2	0.400	0.680	700	50
HP rewritable WORM (C1716T)	1.6	0.035	1.4	1979	57.7
Magneto-optical disks					
MaxOptix Tahiti IIm 5.25"	1.5	0.035	1	3,500	83.3
Fujitsu 3.5" (M2512A)	1.7	0.043	0.230	565	147.8
Magnetic disks (secondary)					
IBM UltrastarXP 3192	5.2	0.008	2.3	1000	434

Table 1.1: Performance parameters of various classes of drives. The transfer rate and capacity are for uncompressed data. The costs presented in this table are only approximate costs.

the search or rewind rate but also the length of the tape. Therefore, the DLT 2000 drive has a slower access time than the DLT 4000 although they support the same search and rewind rate. The last two columns in Table 1.1 give the approximate drive and media cost for various storage devices. Notice that the media cost for tapes is almost two orders of magnitude less than for magnetic disks whereas optical and magneto-optical media are only a factor of 4 to 8 cheaper. Although, the tape drives are much more expensive than the disk drives, their higher cost has to be weighed against the fact that these drives are designed to be used with removable media.

Tertiary memory devices also differ widely in the physical configuration and performance characteristics of the automated library systems (jukeboxes) used for holding and transferring the platters. On one end are small single-drive single-arm stackers capable of holding around ten platters, and on the other end are large libraries with up to ten drives and one or more robot arms capable of holding hundreds and thousands of platters. In Table 1.2 we list the performance and physical characteristics for some representative robotic libraries. The switch time in this table is a summation of the average time needed to rewind any existing platter (if tape), eject it from the drive, move it from the drive to the shelf, move a new platter from the shelf to drive, load the drive and make it ready for reading. The switch time is an important performance parameter in evaluating a tertiary memory device. As seen from the table, the switch time for various jukeboxes also differs widely. In general, the switch time for disk-based jukeboxes is smaller than for tape-based jukeboxes because most tapes need to be rewound to the beginning before unloading. The second important parameter is the number of drives or, more importantly, the ratio of the number of drives and arms to platters. Stackers typically have the highest ratio of drives and arms to platters. The large tape libraries have the smallest ratio of drives to platter but they are often significantly cheaper since the cost of the expensive parts (drives, controllers and robotic arms) is amortized over multiple cheap tape cartridges. A more detailed survey of existing tertiary memory products can be found in [Che94, KAOP91, HSa, Corb].

1.2 Research Issues

Tertiary memory devices pose a challenge to database designers because their performance characteristics are very different from those of magnetic disks. In Figure 1.3 we present the ratio of the worst case access time (switch platter + search + seek whole

Library	Number drives	Number platters	Switch time (s)	Capacity GB	Drive type	Cost $/GB
Stackers						
DLT2500	1	5	45	100	linear tape	61
Exabyte EXB10i	1	10	120	50	8mm helical	180
Pioneer	1	6	5	4	optical disk	461
Medium Library						
Exabyte EXB120	4	116	171	580	8mm helical	172.4
HP 120	4	88	8	114	MO disk	460
Large Library						
Metrum RS-600	5	600	58	8500	1/2" helical	24
Sony DMS-300M	3	320	30	13,000	19mm helical	25

Table 1.2: Performance characteristics of various robotic devices. The switch time is a summation of the average time needed to rewind any existing platter, eject it from the drive, move it from the drive to the shelf, move a new platter from shelf to drive, load the drive and make it ready for reading.

tape + transfer time) to the best case access time (transfer time only) for fetching an 8 KB page and a 1 MB page for various tertiary and secondary storage devices. For magnetic disk this ratio is small compared with tertiary devices for both 8 KB and 1 MB transfers. This means that ordering I/O requests is much more important on tertiary memory devices than on disks. A sub-optimal I/O order that requires an extra platter switch can cost almost a billion CPU instruction in many cases. Therefore, one can afford to spend more processing time for making better scheduling, caching and data placement decisions. Conventional query optimization methods on secondary memory aim at reducing the number of I/Os and amount of I/O but often ignore reordering optimizations. We next discuss a number of different ways in which it is possible to reorder and reduce I/O both during query processing and data loading.

1.2.1 Query Processing Issues

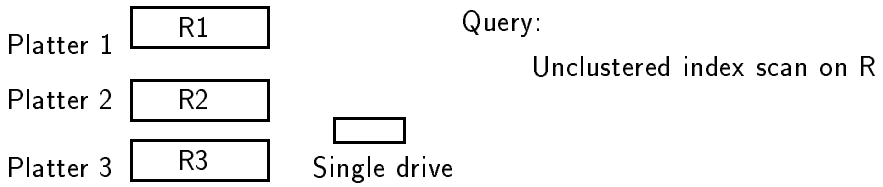
Careful query scheduling must be employed to optimize the order of access to tertiary memory. Query scheduling can be done at various different levels.

First, it is necessary to **avoid small random I/Os**, by scheduling the I/O requests within a single query carefully. Unclustered index scans and joins in limited buffer space can lead to disastrous performance if processed in traditional ways on tertiary memory.

Storage device	Exchange time (sec)	full seek time (sec)	Data transfer rate (KB/sec)	Worst/best access	
				8KB page	1MB page
Sony optical jukebox	8	0.1	800	811	7.5
Metrum RSS-600	58	140	1200	29701	238
Exabyte EXB-120	171	154	470	19094	153
Magnetic disk	-	0.05	4000	26	1.2

Table 1.3: Comparative study of the characteristics of different storage devices. Worst case access time is sum of exchange time, full seek time and transfer time for a page and best case time is the transfer time for a page

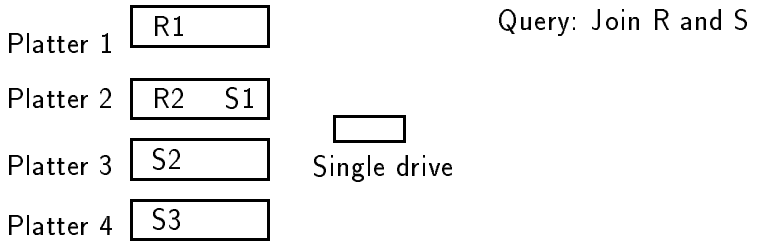
Consider an unclustered index scan query on a relation R divided into three parts $R1, R2$ and $R3$ which are stored on three different platters of a single drive tertiary memory as shown below.



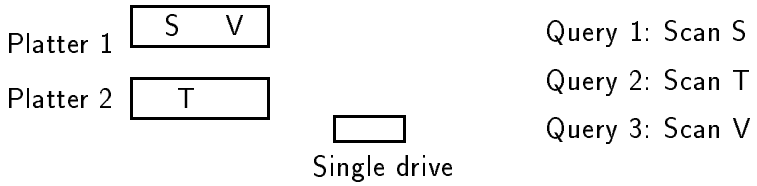
Assume further that the index tree of R is stored on magnetic disk. For such a configuration, if we fetch R using the traditional block-at-a-time approach, a large number of platter switches can result. We can reduce the number of platter switches to a minimum by following the approach of [MHWC90] which reorders I/O requests by first scanning the index tree, sorting the blocks required in storage order and fetching all blocks on one platter before switching to another.

When a query plan accesses multiple relations (as in a join), it is necessary to schedule I/O requests of multiple relations, taking into account both the size of the cache used for staging data in and out of the tertiary memory and the layout of the relation. Consider a two-way join query between relation R stored on platters 1 and 2 and relation S divided between platters 2, 3 and 4 such that each relation is much larger than the cache. A nest-loop join that is oblivious of such a layout cannot do efficient batching of accesses to one platter and may access data randomly across the four platters, leading to many platter switches. For example, instead of scanning all tuples in S for every tuple in R in one pass, if we divide S into parts that fit in the cache and do the nested loop join over each cached part separately, we will incur fewer platter switches. The scheduling and buffering decisions

can get more complicated with multi-way joins accessing more than two relations each of which is spread over multiple platters.

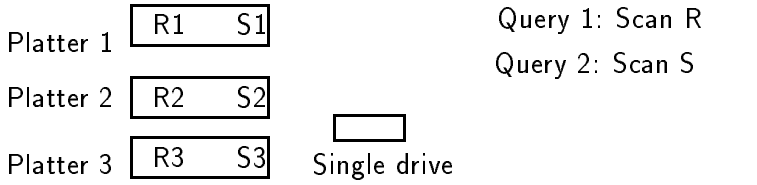


Next, significant gains can be expected by doing **inter-query scheduling**. When queries from different users are executing concurrently, it is important to schedule them so as to avoid undesirable interference between I/O requests of multiple users, and to increase sharing of data accesses. Consider the following configuration.



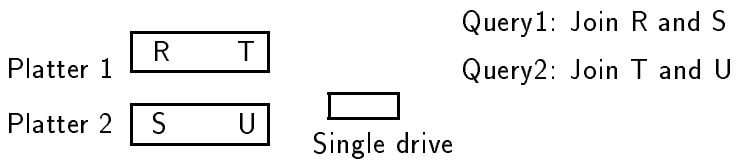
Three different users concurrently submit queries 1,2 and 3 respectively. Queries 1 and 3 access data on platter 1 and query 2 on platter 2. Normally, the execution of the three queries would be started simultaneously. This could result in frequent switches between platters 1 and 2. On the other hand, if we withhold execution of query 2 until queries 1 and 3 are over we can avoid these switches.

There are other cases where **query interleaving** is necessary to reduce I/O cost. Scheduling at the level of whole query alone may not be sufficient. Consider the situation where we have two select queries on relations R and S each of which is spread across platters 1,2 and 3. If we interleave the execution of these queries to first scan $R1$ and $S1$; then scan $R2$ and $S2$ and finally scan $R3$ and $S3$, then each platter will be loaded only once. In contrast, if we do not synchronize the scans thus, the scan on $S1$ could complete before $R1$. If we then start scanning $S2$, the I/O requests on $R1$ and $S2$ could interfere leading to extra platter switches.



Also, further gains can be obtained by **reordering** the normal processing order of queries. For instance, suppose a user is processing a sequential scan query on a relation R and a second sequential scan query on R arrives from a different user when the first user is one-fifth of the way through. It makes sense to let the second user synchronize with the first one and process the query on the remaining four-fifths of R first instead of starting from the beginning. This requires us to reorder the execution of the second query.

The above forms of query scheduling require **unconventional caching and prefetching strategies** for managing the magnetic disk cache. A relation can be cached when its platter is just about to be unloaded even if we do not intend to execute queries on it immediately. For instance, if we have a join query between relation R on platter 1 and S on platter 2 and another join query between T on platter 1 and U on platter 2, it might help to cache both R and T when platter 1 is loaded even if we are scheduling the join between R and S first. Also, when choosing data to replace, we need to take into account the location of the data in addition to its time of last access. For instance, data blocks residing on a loaded platter could be replaced in preference to data residing on unloaded platters since the cost of fetching the former data blocks when needed could be smaller.



There is an additional challenge to solving the above problems on tertiary memory: dealing with the diversity of existing tertiary memory products and the applications that use them. Tertiary memory devices differ widely not only from typical secondary memory devices, but also among themselves. For some devices the platter switch cost is high, making it important to reduce the number of I/O requests, for others the data transfer bandwidth is low, making it important to reduce the amount of data transferred. Tape devices have significant seek overhead whereas disk devices allow random access. Also, one can expect a lot of variation in the applications that use tertiary storage devices. Some involve a large number of relatively small objects whereas others require a small number of very large objects. It is essential, therefore, to identify the key performance metrics and parameterize the query processing and data placements decisions on these metrics.

1.2.2 Data placement Issues

Careful clustering techniques have to be used for data layout to accommodate for the non-uniform access times of tertiary memory. Since updates on tertiary memory relations are infrequent, data clusters are easier to maintain. A large body of work exists in relational databases for organizing tables and clustering them for fast retrieval on secondary storage devices [LY77, YSLM85]. Most of these techniques can be extended to tertiary memory databases. One problem that has not received adequate attention from the database community, even for magnetic disks, is the storage organization of large multidimensional arrays — one of the biggest contributors of data volume in many databases [DR91a]. Therefore, in this thesis, we concentrate on the problem of organizing large multidimensional arrays.

Scientific and engineering applications often utilize large multidimensional arrays. Earth scientists routinely process satellite images in the form of large two or three dimensional arrays [DR91b]. Their simulations of atmosphere and ocean climatic conditions generate large regular arrays of floating point numbers as output [M+92]. For example, typical runs of the UCLA General Circulation Model (GCM) generate five dimensional arrays of size 5 to 50 Gigabytes each. Most existing databases store the arrays as large objects that are treated simply as a homogeneous sequence of bytes. Hence, they do little in terms of optimizing their storage. The traditional method of storing a multidimensional array is *linear allocation* whereby the array is laid out linearly by a nested traversal of the axes in some predetermined order. This strategy, which mimics the way FORTRAN stores arrays in main memory, can lead to disastrous results on tertiary memory devices because of their highly non-uniform access time. Since users typically access large arrays in several different ways, the FORTRAN order will optimize for one access pattern while making others very inefficient. Optimizing the allocation of the array becomes increasingly important and challenging as array dimension and size increases, especially on sequential media like tapes.

Outline of Dissertation

This thesis is composed of seven chapters. Chapter 2 presents the architecture of a modified query processing engine that provides the enabling mechanisms for supporting all the forms of query scheduling, caching and I/O scheduling introduced in Section 1.2.1.

Chapter 3 presents the policies used for scheduling and caching in the context of this architecture. We have extended the POSTGRES database system with the architectural extensions suggested in Chapter 2. Chapter 4 presents the details of this implementation and an experimental evaluation of the system on the SEQUOIA-2000 benchmark and several artificial datasets. Chapter 5 addresses the issues of storing large multidimensional arrays on tertiary storage devices (introduced in Section 1.2.2). Chapter 6 discusses related work. Finally, concluding remarks appear in Chapter 7.

Chapter 2

Query Processing Architecture

This chapter presents our proposed architecture for tertiary memory query processing. We assume a relational architecture with a three level memory hierarchy: tertiary memory attached to a disk cache attached to main memory as shown in Figure 2.1. Most of the user data (relations and large objects) normally resides on tertiary memory whereas system catalog information is stored permanently on magnetic disk. Indices on relations are stored either on magnetic disk or tertiary memory. We assume that all data from tertiary memory needs to be staged on the disk cache (as shown in Figure 2.1) before any query on it can be processed. We discuss the implications of this assumption in Chapter 7 (Section 7.3).

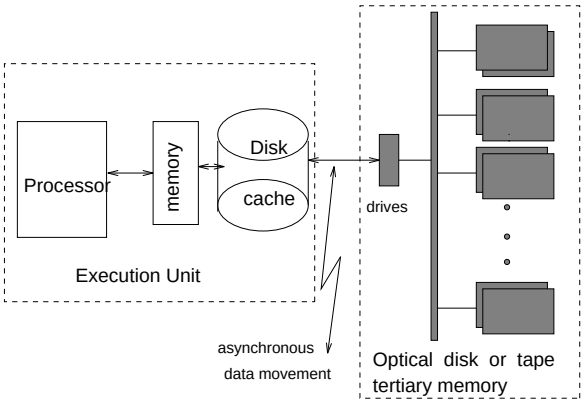


Figure 2.1: The Physical Configuration.

Chapter outline

First, Section 2.1 briefly reviews the architecture of a conventional database system and identify reasons that make them inappropriate for processing queries on tertiary memory devices. Then, Section 2.2 present the architecture of our system. Two distinguishing features of the new architecture are (1) a reorderable executor and (2) a centralized scheduler. Section 2.3 describes the design of the reorderable executor. The design of the scheduler is deferred until Chapter 3.

2.1 Background: Conventional architecture

In the last chapter, we motivated the need for carefully reordering I/O for efficient access to tertiary memory devices. Most conventional database systems can achieve only a limited form of I/O reordering. Typically multiple server processes concurrently execute queries on behalf of different users. These processes normally submit I/O requests a block at a time during query execution. The device scheduler thus has only a few blocks available for reordering at any one time. The maximum number of pending I/O requests for the device scheduler can be at most the number of concurrent user processes. This limits the amount of reordering that the device scheduler can do. Many systems attempt to get around this inefficiency by incorporating some form of bulk prefetch mechanism where an I/O process is used to asynchronously prefetch multiple I/O blocks to hide the inefficiency of block-at-a-time I/O. However, the amount of data that can be prefetched at one time is limited by the size of the disk cache. As we will show in this chapter, one can achieve greater flexibility of reordering and asynchronous prefetching by co-ordinating the device scheduling, cache management and query scheduling decisions.

2.2 Proposed Architecture

Figure 2.2 sketches the process architecture of our database system. The new addition over a conventional process architecture is the scheduler process. As in a conventional system, each user's queries are executed in a different process (marked "user processes" in Figure 2.2). But, instead of directly and independently submitting the I/O requests to the disk cache manager, each user process submits its request for data blocks to the scheduler before starting to execute a query. The scheduler is a centralized entity that (1) schedules

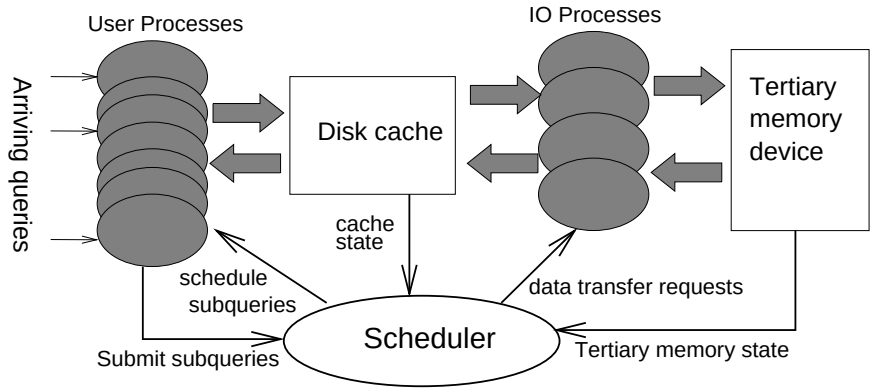


Figure 2.2: Process architecture of the tertiary memory database system.

I/O requests on the tertiary memory, (2) schedules queries of each user process, and (3) decides what data is staged in or evicted from the disk cache. The scheduler maintains a collection of I/O processes for asynchronously moving data from the tertiary memory to the disk cache. Each user-process submits its request for data that it needs to the scheduler, and only after the scheduler has put the data on the disk cache is the query scheduled for execution. Queries are divided into smaller units called subqueries[1] and scheduled in units of subqueries instead of as a whole unit.

In Figure 2.3 we show further details of the interaction between the user-processes and the scheduler. Each user-process breaks its query into a list of subqueries and submits them to the scheduler. The scheduler collects such lists of subqueries from multiple different user-processes and is responsible for deciding the order of executing the subqueries. After submitting the list of subqueries, the user-process waits for the scheduler to make one or more subqueries "ready" for execution. When a user-process has finished executing the ready subqueries, it informs the scheduler of their completion and waits for the next set of subqueries to be "ready". On the scheduler side, the first task is to select the data (required by one or more subqueries) to be fetched next and make space in cache for the selected data items. The scheduler then instructs an I/O process to fetch the selected data items. As data required by a subquery is brought into the disk cache, the data is "pinned" in the cache; when all the required data has been brought in, the subquery is marked "ready" for execution. When a subquery completes, the scheduler "unpins" the data accessed by the

[1]The term subqueries is not to be confused with the SQL notion of subqueries. We use the term "subqueries" to refer to parts of query as explained in greater detail in Section 2.3.

subquery so that some other data items may be put in its place.

This architecture can do better I/O optimizations than conventional systems since a single scheduler process co-ordinates all caching and I/O activities of multiple user-processes and since, each user-process pre-determines all data blocks that will be used before starting to execute a query. This enables the scheduler not only to do better planning of I/O requests within a single query but to also batch one query's I/O requests with those of other queries. However, implementation of this query processing architecture raises a number of new issues, including:

- **The unit of scheduling:** What is the size of the subqueries that are used for scheduling? In particular, scheduling at the level of whole queries may not be feasible since the disk cache may not be large enough to hold all the data required by a query. Also, too large a subquery can adversely affect caching performance since the scheduler might "pin" data required by a subquery for too long. These issues are discussed in Section 2.2.1.

- **The design of the user processes:** How does the user-process extract the list of subqueries needed before execution? It may not be possible to find out in advance all data that are required by a query. How do the user-processes handle such cases — how is the execution unit of a user-process modified to handle the subquery at a time execution paradigm? How does it interact with the scheduler to exchange subquery information? These topics are discussed in Section 2.3.

- **The design of the scheduler:** How does the scheduler decide on the order of executing the subqueries? How does it make the device scheduling and cache management decisions? These topics are discussed in Chapter 3.

2.2.1 Unit of scheduling

Each query on a base relation is divided into a number of smaller subqueries defined on the **fragments** of the base relation. A fragment is a set of tuples in a relation laid out contiguously on a single platter. In general, a relation can be larger than the disk cache, spanning multiple platters and spread arbitrarily across each platter. Such a storage layout can arise when a relation is created by successive appends at different periods of time. In historical and archival systems data is loaded periodically, and it is often too expensive

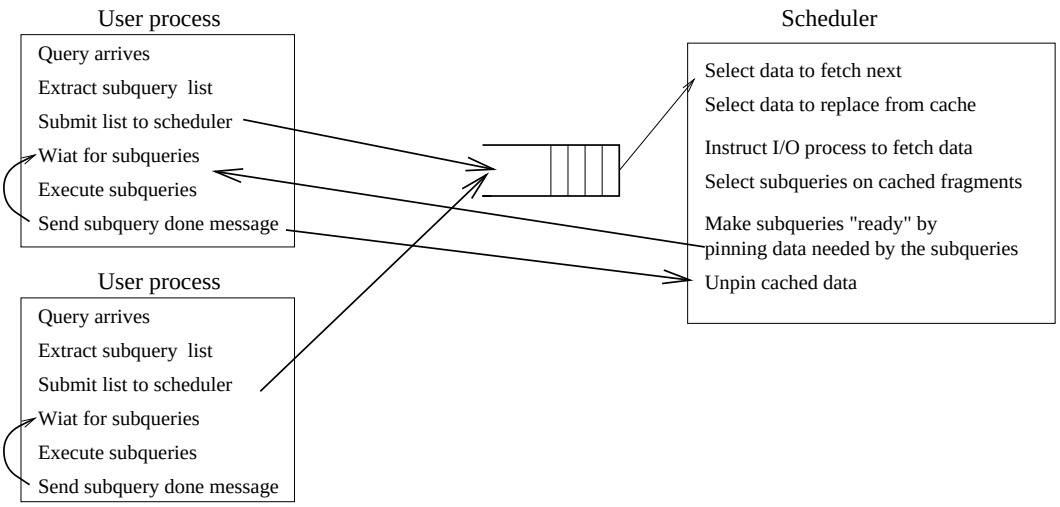


Figure 2.3: Interaction between the scheduler and User-processes.

to reorganize so as to store an entire relation contiguously. Hence the query processing engine should be able to handle bad storage layout. A relation can thus be composed of multiple fragments. A fragment can be fetched as a whole without incurring platter switches and seeks during its transfer. For disk-based (as opposed to tape-based) tertiary memory devices, we treat all the data blocks lying on a single platter as part of the same fragment since the seek cost is not a significant part of the total I/O cost. In contrast, for tape-based tertiary memory only the parts of a relation stored contiguously can belong to the same fragment.

Fragment size: The proper choice of the fragment size is crucial to performance. Too large a fragment can limit the degree of concurrency whereas too small a fragment can increase the overhead of scheduling. The physical layout of the relation determines some boundaries for creating fragments as described above. But, we can divide a contiguously stored fragment of a relation into even smaller fragments. The limits (both maximum and minimum) on the size of fragment can be determined in a number of different ways:

1. One limit on the maximum size of a fragment is imposed by the size of the disk cache, C — we limit the size of each fragment to be no more than $1/n$th of the cache size where n is the maximum number of fragments that are needed together for processing a subquery. For instance, a 2-way nested-loop join subquery requires at least two fragments to be in the disk cache before we can process the subquery. Thus, if we

have a n-way nest-loop join, we should be able to hold at least n fragments in the cache together. In an ad-hoc query processing system, theoretically there is no limit on the maximum number of relations n that can participate in a nested loop join. However, in practice a nested loop join involving too many relations is likely to be too slow to be gainfully used. Therefore, it is possible to determine a value of n in practice.

2. A second limit is imposed by the number of drives — for high utilization it is desirable to be able to transfer concurrently as many fragments as the number of drives, d. This limits the size of each fragment to atmost the ratio of the cache size and the number of drives.

3. Another limit is obtained by the desired degree of concurrency. To exploit the benefits of multiprogramming at the query execution level, if it is necessary to support some x number of users and if each user needs an average of m fragments simultaneously on disk, we would like to be able to cache together at least mx fragments. This limits the size of each fragment to not more than C/mx.

4. Too small a fragment size can increase the overhead of scheduling and hence we must choose a fragment size such that the overhead is a small fraction of total query execution time. If F_{min} is the smallest fragment size for which the scheduling overhead "significantly" exceeds the total query execution time, then we want all fragments to be $\geq F_{min}$. The value of F_{min} depends on the overhead which in turn depends on implementation-specific details that are hard to characterize in a closed-form formula. Thus F_{min} is a parameter to be determined by a DBA based on installation specific details.

5. Each fragment is composed of an integral number of fixed sized storage blocks (in the same way a disk file is composed of fixed sized pages). The size of each fragment has to be larger than the tertiary block size B. For tertiary memory devices the size of the storage block is typically much larger than a normal disk page because of the high access latency. The optimal value of the storage block B depends on the platter switch cost, the transfer cost, the fixed search/rewind time on tape, the seek rate and the distribution of the amount of data needed per access. Too large a block size can increase the amount of redundant data transfered and too small a block size can cause

the seek and search overhead to be incurred too often. This tradeoff is similar to the one encountered when choosing a page size for the disk to main-memory hierarchy. Hence, we rely on similar device and application specific calculations in choosing the block size.

Summarizing the above set of constraints, we limit the size, F of each fragment of a relation as:

$$F \leq min(\frac{C}{n}, \frac{C}{d}, \frac{C}{mx}) \text{ and } F \geq max(F_{min}, B)$$

where

C	cache size
n	maximum number of relations in a nested-loop join
d	number of drives
x	maximum degree of multiprogramming
m	average number of fragments per query
B	block size
F_{min}	minimum fragment size

Section 3.3.1 of Chapter 3. illustrates, how we use these constraints in determining fragment sizes for our experiments.

2.3 Query Execution

This section presents the design of the execution engine of each user process. It covers details of (1) how each query is analyzed before execution to extract the list of subqueries and the data items it needs and (2) how the execution engine cooperates with the scheduler to process a plan-tree in an arbitrarily reorderable fashion. First, Section 2.3.1 lists the specifications that the execution unit of each user-process in our architecture should meet. Then, Section 2.3.2 presents the design of the different phases of query execution that meets these specifications.

2.3.1 Specifications

1. *Submit to the scheduler a list of subqueries to be executed*

   The scheduler does not need to know all the details of the subquery, only which fragments are needed *together* in executing the subquery. Each subquery is thus represented as a list of fragments that are needed for executing the subquery. Consider

a nest-loop join between relations R and S where R consists of three fragments R_1, R_2 and R_3, and S consists of two fragments S_1 and S_2. The list of subqueries submitted to the scheduler, called the `SQ-list`, is:

$$\{(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2), (R_3, S_1), (R_3, S_2)\}$$

Consider another example of a 3-way nested-loop join query between relations R, S and T where T consists of only one fragment T_1. For this query the `SQ-list` is:

$$\{(R_1, S_1, T_1), (R_1, S_2, T_1), (R_2, S_1, T_1), (R_2, S_2, T_1), (R_3, S_1, T_1), (R_3, S_2, T_1)\}$$

In general, a subquery extracted from an n-way nested-loop join query will require n fragments to be present together. The executor attempts to expose the maximum parallelism to the scheduler by submitting as large a subquery list as possible.

2. *Execute subqueries out-of-order*

   The scheduler can mark subqueries ready for execution in an arbitrary order. Therefore, the subqueries in an `SQ-list` should not have any ordering constraints between them. For the two way join example above, the subquery (R_2, S_1), for instance, might be scheduled before the subqueries (R_1, S_1) and (R_1, S_2). This implies that when the operators of a plan tree have precedence constraints on them, the subqueries must be submitted in multiple stages. For example, if for the above two-way join query, we did a hash-join between R and S (with S as the hash-build relation), then we need fragments of S before fragments of R. Thus, the scheduler will be first submitted the `SQ-list` $\{(S_1), (S_2)\}$ and only after all fragments of S are processed and the hash table for S built, do we submit the `SQ-list` $\{(R_1), (R_2), (R_3)\}$ on fragment R.

3. *Execute multiple subqueries together*

   When an executor contacts the scheduler for "ready" subqueries, the scheduler could have more than one subquery ready. We require that the executor be able to process multiple subqueries together and not just one at a time. Executing one subquery at a time can lead to redundant computation for joins, since the scans on the outer relation cannot be shared across multiple fragments of the inner relation. For instance in the two-way join example, if S_1, S_2 and R_3 are cached, the scheduler will "ready" both the subqueries (R_3, S_1) and (R_3, S_2). The executor must be able to join R_3 with both S_1 and S_2 in one scan of R_3. Hence, although executing each subquery separately

would allow for easy implementation, we must provide a means of executing multiple subqueries together.

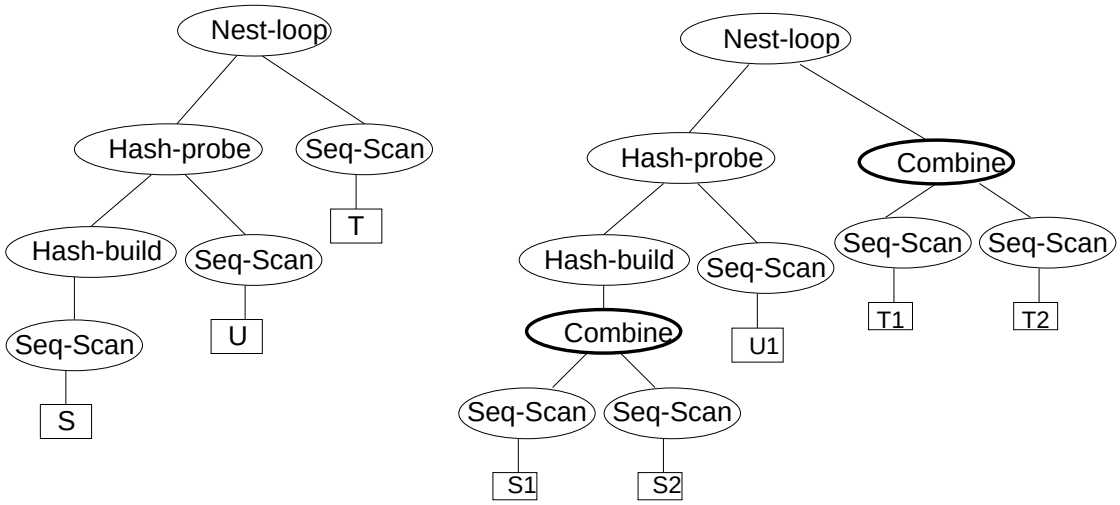
2.3.2 Phases of Execution

This section describes the design of an execution engine that meets the above set of requirements. This discussion is based on the POSTGRES execution engine, in which each query plan is a tree of operators and all operators provide a uniform iterator interface. In accordance with this interface, each operator of the plan tree provides *start*, *next* and *close* calls. Most other relational database systems, including System R [ABC+76], EXODUS [RC87], Starburst [HCF+88] and Volcano [Gra90] have analogous operator-based execution engines and can be extended similarly.

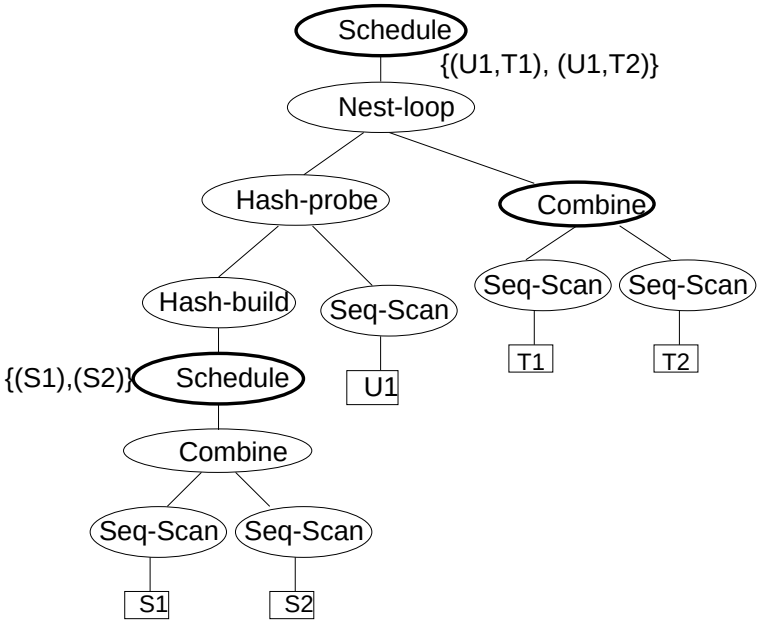
A query is first optimized as usual except for a few minor changes related to sorting via index scans that are discussed in Section 2.3.4. The optimized plan tree then passes through the fragmentation, subquery extraction and execution phases that are described next.

Fragmentation

In the fragmentation phase, each scan node on each base relation is replaced by a *combine node* that contains a list of scan nodes on the fragments of the base relation. The type of scan (sequential scan or index scan) on the fragments is the same as on the base relation. We assume that all the fragments of a relation have the same set of indices. For example, in Figure 2.4(a) we show the plan-tree of a 3-way join with three sequential scan nodes on base relations S, U and T. In Figure 2.4(b) we show the plan-tree after fragmentation. Relation S has two fragments S_1 and S_2, thus we replace a scan node on relation S by two scan nodes on each of the two fragments of S and add a combine node above these two scan nodes. Relation T also has two fragments and we follow the same procedure for T. Relation U has only one fragment, therefore we leave the scan node on U unchanged. During the fragmentation phase we leave the rest of the plan-tree unchanged. Only the scan-nodes on relations residing on tertiary memory are changed.



(a) Plan-tree after optimization (b) Plan-tree after fragmentation



(c) Plan-tree after extraction

Figure 2.4: Example of a three-way join.

Subquery Extraction

In this phase we analyze the fragmented plan tree to extract the `SQ-lists` and insert special nodes called *schedule nodes* that are responsible for communicating with the scheduler and maintaining necessary synchronization information during execution. Each schedule node has an associated `SQ-list`. Because of precedence constraints between operators (Section 2.3.1(item 2)), we could have multiple `SQ-lists` in a plan-tree. For example, the plan-tree in Figure 2.4(b) has ordering constraints between the hash-build and hash-probe nodes. Hence, we added a schedule node before the hash-build node since the hash-build stage has to complete before starting processing on any nodes above it. We add a second schedule node at the top for the rest of plan-tree.

For inserting such schedule nodes in a plan-tree and for constructing the `SQ-lists` we define a "Find-Sub-Query" call for all nodes in the plan-tree. This routine returns the list of subqueries necessary to process the node. We give below the "Find-Sub-Query" routine for a few common nodes.

Find-Sub-Query for various nodes

Combine node:

return list of fragments under the combine node

Hash-build, Aggregate or Sort node:

`query-list` = Find-Sub-Query (subtree underneath node)

if `query-list` non-empty

add a schedule node containing `query-list` below this node

return `empty-list`

Join node:

`listL` = Find-Sub-Query (left subtree)

`listR` = Find-Sub-Query (right subtree)

`query-list` = cross product of `listR` and `listL`

If `listR` is empty, query-list = `listL`

If `listL` is empty, query-list = `listR`

return `query-list`[2]

For our example in Figure 2.4, the "Find-Sub-Query" call on the Hash-build node adds a schedule node containing the list $\{(S_1), (S_2)\}$ and returns the empty-list. The "Find-Sub-Query" call on the Hash-probe node returns the list $\{(U_1)\}$ and on the right branch of the

Nest-loop node returns the list $\{(T_1), (T_2)\}$. The "Find-Sub-Query" call on the Nest-loop node returns the cross product $\{(U_1, T_1), (U_1, T_2)\}$ that is stored in a schedule node at the top of the tree.

Execution

Our goal during the design of the execution engine was to follow the normal mode of processing as far as possible except for occasional communication between the execution engine and the scheduler for passing subquery information, collecting ready subqueries and notifying the schedule of subquery completion. We show here how minor modifications in the scan nodes and the newly introduced schedule and combine nodes enable us to achieve this goal.

For efficiency reasons (discussed in Section 2.3.1, item 3) we want to execute all subqueries of a plan-tree from a single plan-tree data structure instead of building a separate plan-tree for each subquery. This requires us to keep track of what subquery of the plan-tree is currently being executed. We do so by marking the scan nodes of the subqueries currently being executed as `available` and all other scan-nodes `suspended`. The plan-tree is then processed as usual: starting from the root of the plan tree, successive "next" calls are made to each node of the tree. When a "next" call is made on a combine node it submits the "next" call to a scan node underneath it that is marked `available`. A "next" call on a `suspended` scan node does not return a tuple. Thus, only scan-nodes of currently scheduled subqueries participate in execution.

We next discuss how and when the schedule nodes are used for exchanging subquery information. Note that there could be multiple schedule nodes in the plan tree. It is critical to ensure proper interaction between these nodes to prevent deadlocks during execution by (1) submitting the `SQ-list` of a schedule node before the `SQ-list` of any schedule node above it and (2) processing subqueries of one schedule node and notifying the scheduler of their completion before submitting a `SQ-list` of some other schedule node. In addition, we want all these operations to be seamlessly integrated with the normal processing of the plan-tree. We achieve this goal by localizing all communication control into a "next" call of a schedule node. The schedule node is just another node of the plan-tree and during normal processing when a "next" call is made on the schedule node, we take the following steps:

1. Make a "next" call on the node underneath the schedule node to get the next tuple, t
2. If t is valid, return t
3. Else, if first time empty tuple returned
   - Submit the stored `SQ-list` to the scheduler
   - Make a blocking call to the scheduler to get the next collection of subqueries. Let Q be the collection of "ready" subqueries returned by the scheduler.
   - Enable Q for execution by marking all the scan nodes appearing in Q as `available`.
   - Finally, make a "next" call on the node underneath and return the tuple obtained.
4. Else, /* stored list of subqueries already submitted */
   - Inform the scheduler of the completion of the last batch of scheduled subqueries, if any, and mark the scan nodes of those subqueries as `suspended`.
   - Make a blocking call to the scheduler to get the next collection of subqueries. Let Q be the collection of "ready" subqueries returned by the scheduler.
   - If Q is empty, then all subqueries have been executed, therefore return EOF.
   - Else, enable Q for execution by marking all the scan nodes appearing in Q as `available`.
   - Finally, make a "next" call on the node underneath and return the tuple obtained.

We will illustrate the above steps with the execution of the plan-tree in Figure 2.4. Initially, all the fragments are marked `suspended`. The first "next" call results in the submission of the list $\{(S_1), (S_2)\}$. Assume the scheduler makes (S_2) available first. As a result, the hash-build operation is partially completed. The scheduler is informed of the completion of subquery (S_2) (so it can uncache S_2 if needed) and a blocking request is made to get the next subquery. When the scheduler makes (S_1) "ready", the rest of the hash-build operation is completed and the scheduler is informed of its completion. Next, the `SQ-list` $\{(U_1, T_1), (U_1, T_2)\}$ on the topmost schedule node is submitted and a blocking request is made for the next subqueries. Assume both the subqueries are scheduled together. All data required by the plan-tree is now available. Hence, execution of the query is completed by pipelining the hash-probe and nest-loop operations and the scheduler is notified of the completion.

The above scheme requires certain caution when scheduling *multiple* join subqueries together to avoid repetition of the following form: Consider the $R \bowtie S$ example of Section 2.3.1. Using the above scheme we first submit the following `SQ-list` to the scheduler.

$$\{(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2), (R_3, S_1), (R_3, S_2)\}$$

Suppose the scheduler next makes (R_1, S_1) ready, the executor finishes processing (R_1, S_1) and asks for the next set of ready subqueries. Suppose the next set of scheduled subqueries is $\{(R_1, S_2), (R_2, S_1), (R_2, S_2)\}$. To execute these three subqueries, scan-nodes of fragments R_1, R_2, S_1 and S_2 will be marked `available`, and the plan-tree will be processed as usual. But, by doing so, we have *repeated* the execution of subquery (R_1, S_1). Note that this form of repetitions can occur only for join subqueries that require more than one fragment together.

To avoid such repetition, the scheduler maintains for each `SQ-list`, the list of subqueries already completed in a special `done-list`. Whenever two or more new join subqueries from a `SQ-list` are marked ready, the scheduler checks for repetitions as follows:

1. Let ready-list = list of ready queries.
2. Let I = list of inner fragments of all the ready subqueries
3. O = list of outer fragments of all the ready subqueries.
4. If the length of either I or O 1, stop since there can be no repetition, otherwise,
5. If the done-list contains a subquery s such that its inner fragment
   i is in I and outer fragment o in O, s will be executed
   again if all subqueries in the ready-list are scheduled together,
   therefore remove o from O. All subqueries in the ready-list whose
   outer relation is o will not be executed in this batch and hence
   they are removed from the ready-list. The subqueries removed in this
   step are postponed for execution later.
6. Redo steps 1 through 5 until, no repetition is detected.

We will illustrate this procedure with the example of the two-way join above. Suppose, the done-list contains subquery (R_1, S_1) and the ready-list contains subqueries $\{(R_1, S_2), (R_2, S_1), (R_2, S_2)\}$. Thus, $I = \{S_2, S_1\}$ and $O = \{R_1, R_2\}$. Condition 5 is satisfied since the done-list contains subquery (R_1, S_1) whose outer fragment, R_1 belongs to O and

inner fragment S_1 belongs to I. By step 6, we remove from the ready-list all subqueries that contain the R_1 as the outer relation since R_1 is the outer relation of s. Therefore, we remove subquery (R_1, S_2) from the ready-list. In the next iteration, O is $\{R_2\}$ and I is $\{S_2, S_1\}$ and there can be no repetition since length of O is 1. Thus, in the first batch subqueries (R_2, S_1) and (R_2, S_2) will be executed and in the next batch subquery (R_1, S_2) is executed.

2.3.3 Handling Dependencies

Sometimes it is not possible to know before execution what subqueries are needed because there is *dependency* (or pointers) between fragments. To determine what fragments are need, some other fragments have to be processed. For example, with index scans, the set of data blocks required can be determined only after partial processing on the index trees. Similarly, with tuples pointing to large objects, the large objects to be fetched can be determined only after selecting the required tuples. To handle dependencies, two changes are needed:

1. First, we augment the plan-tree structure further with a special schedule node called the *resolve node*. The resolve node is added during the extraction phase immediately above the plan-tree node that introduces dependency between fragments. The resolve node, like the schedule node, contains a list of subqueries (`SQ-list`) that need to be executed first to resolve the dependencies. For instance, for an index scan, the resolve node is added immediately above the corresponding combine node and the `SQ-list` is the list of index trees on the indexed fragments as shown in Figure 2.5. The `SQ-list` of the first schedule node above this resolve node cannot be established and hence is marked `unresolved`.

2. Next, we process nodes that introduce dependency in two stages: in the first stage a "ResolveDependency" call is made to compute the dependent list of subqueries and in the second stage after the subqueries are scheduled the rest of the node is processed. For instance, for the index scan node in the "ResolveDependency" stage the index tree is scanned and the list of matching TIDs sorted to get the list of blocks of S_2 and S_1 ($BL(S_2)$ and $BL(S_1)$) that needs to be fetched. In the second stage, after these blocks are fetched we complete the rest of index scan.

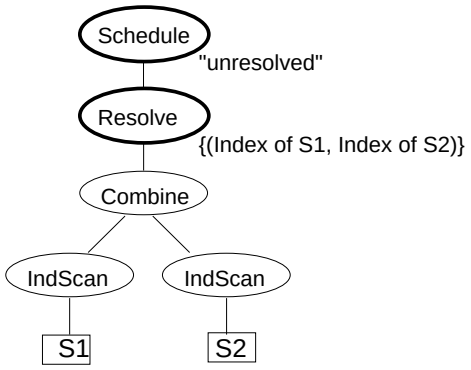


Figure 2.5: Resolve nodes for index scans.

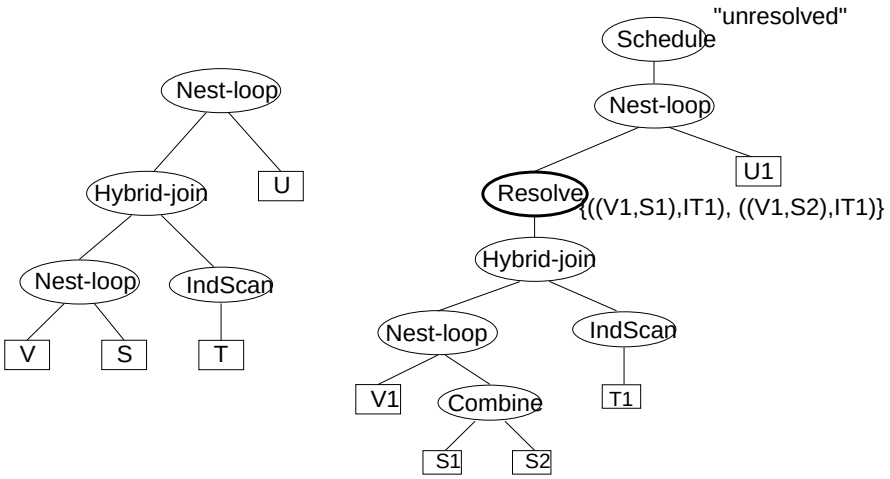


Figure 2.6: Plan-tree with dependency. The right side shows the plan-tree after the extraction phase. Sequential scan nodes are omitted for clarity.

With these modifications we can handle dependencies during execution as follows: when it is time to process a schedule node s marked "unresolved", we make a "resolve-sub-query" call on the node below. The resolve-sub-query call behaves like the "find-sub-query" call for each node of the plan-tree until a resolve node is reached. The resolve node submits its stored `SQ-list` to the scheduler, and as subqueries from this list get scheduled, we make ResolveDependency calls on the node below to get the new `SQ-list`. The final `SQ-list` is then returned and stored in the schedule node and execution proceeds as usual. We now illustrate details of this method with the three normal cases of dependencies in relational engines: index scans, joins with index scans on inner relation and large object access.

Index scans: Consider the example in Figure 2.5. When we reach the schedule node at the top of the plan-tree with "unresolved" `SQ-list`, we make a "resolve-sub-query" call to the resolve node underneath. During this call, we first submit the stored list of index trees (IS_1 and IS_2) to the scheduler. Then, suppose index tree IS_2 is marked "ready". A ResolveDependency call is made on the index scan node where the index tree is scanned and the list of matching TIDs (tuple identifiers) sorted to get the list of blocks of S_2 ($BL(S_2)$) that need to be fetched. The scheduler is sent a "done" message for IS_2 and a request is made for the next subquery. When IS_1 is marked "ready", the list of blocks of S_1 ($BL(S)1$)) is collected similarly. The new SQ-list, $\{(BL(S_1), BL(S_2)\}$ is returned to the schedule node above and execution proceeds as usual. Suppose at some later time $BL(S_2)$ is "ready", the TID list is used to retrieve the matching tuples.

Nested loop join with runtime index on the inner relation: We demonstrate how to execute the hybrid join algorithm [CHH+91], which is an improvement over the standard nest-loop join.

We first describe the hybrid join algorithm. It works in two stages: In the first stage, for each tuple of the outer relation the index of the inner relation is probed and entries are made in an in-memory **join table** for each matched <outer tuple, inner TID> pair (inner TID refers to the identifier of a tuple of the inner relation that is obtained from the index tree). The join table is then sorted in storage order of the inner relation TIDs. In the second stage, the relevant inner relation tuples are fetched in storage order and merged with the join table to form the result tuples.

We adapt this algorithm to our framework. In the extraction phase, we add a resolve node above the hybrid join node as shown in the example of a four-way join in Figure 2.6. The `SQ-list` of the resolve node is a cross product of two subquery lists: one from the outer branch of the join node ($\{(V_1, S_1), (V_1, S_2)\}$) and the other from the inner branch ($\{(IT_1)\}$, the index tree of T_1). Note that the list from the inner branch is the index trees of the inner relation fragments. The schedule node above this resolve node is marked `unresolved`.

During execution, when "resolve-subquery" call is made to the resolve node, we submit the stored `SQ-list` ($\{((V_1, S_1), IT_1), ((V_1, S_2), IT_1)\}$) to the scheduler and wait for "ready" subqueries. When some set Q of subqueries are "ready", we use the hybrid join algorithm to get the list of blocks of the inner fragments. In our example, if $((V_1, S_1), IT_1)$

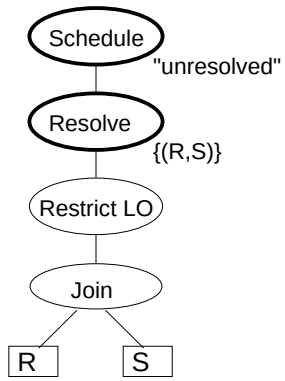


Figure 2.7: Adding resolving nodes for large object access.

is "ready", we construct the hybrid join table using index tree, IT_1 and notify the scheduler of the completion of this subquery. Later, when $((V_1, S_2), IT_1)$ is "ready", we complete the join table. When all subqueries in the `SQ-list` are executed, we extract the list of blocks of the inner fragments that need to be fetched. This completes the ResolveDependency call and we return the list to the schedule node above. The schedule node above the resolve node can then constructs its `SQ-list` and execution proceeds as usual. In our example, the `SQ-list` of the schedule node is $\{(\mathrm{BL}(T_1), U_1)\}$ where $\mathrm{BL}(T_1)$ denotes the list of qualifying blocks of T_1. When this subquery is scheduled, the join is completed using the in-memory join table. The result tuples are pipelined to the Nest-loop join on U_1.

Large objects: To support reordering between large object accesses of different tuples, we add a resolve node after the node that accesses the large objects. The `SQ-list` is derived from the plan tree underneath this node. For example in Figure 2.7, a "selection" clause on a large object is above the join node between R and S. Therefore, the `SQ-list` $\{(R, S)\}$ is stored in the resolve node.

During the resolution phase, we submit the `SQ-list` to the scheduler and when the subquery is "ready" a ResolveDependency call is made to the node accessing the large object. During this call, the join is completed and the resultant tuples along with the IDs of large objects required by them are collected in an in-memory table (as in the case of nest-loop join subquery above). The list of large objects is then returned. The schedule node submits this collected list to the scheduler. The scheduler fetches the large objects in an efficient order. When a large object is "ready" the corresponding tuple is processed further and the scheduler is notified of its completion. Cases where all of the large object

is not needed will require modification of the function that selects the part to be fetched. The execution of the function has to be split into two phases, where in the first phase the function selects the blocks of the large object to be fetched (like in index scans) and in the second phase the function actually processes the data.

Dealing with limited memory: If the in-memory table is larger than the available main memory, then the resolve node cannot complete the construction of the entire table in one pass. Thus, the whole resolve step cannot be completed in one "resolve-sub-query" call and multiple passes are required. Each "resolve-sub-query" call returns only the partial list of data along with an "incomplete flag". The schedule node above the resolve node executes the partial subquery list and submits successive "resolve-sub-query" call until the entire query is completed.

2.3.4 Preventing reordering failures

Free reordering of scans does not yield the correct answer when an index scan is used for getting tuples in sorted order e.g., in a merge join. When sorting order is important, the optimizer adds a modified combine node (called merge-combine) above the index-scanned relation. This modified combine node uses the individual index scans on fragments to get sorted runs that are merged together to sort the entire relation. The "Find-sub-query" call on the merge-combine node is slightly different than on a normal combine node. For the merge-combine node, the "Find-sub-query" call results in the addition of a schedule node containing a single subquery of *all* the fragments and their index trees. Similarly, when accessing large objects, when the sort-order of tuples is important we cannot reorder the processing of tuples. In such cases, we limit the size of the in-memory table to tuples whose results we can buffer. Instead of submitting the entire list of large objects to the scheduler, we submit the list in smaller batches. When a large object is marked ready by the scheduler, the corresponding tuple is processed and the result buffered in the in-memory table until all tuples preceeding it are processed.

2.4 Summary

We presented the design of a query processing strategy that is optimized for accesses to a tertiary memory database. The main features of the new architecture can be

summarized as follows:

- We take a more unified and aggressive approach to reducing and reordering I/O on tertiary memory. Our system consists of a centralized scheduler that keeps system-wide information about the state of the tertiary memory, the disk cache and the queries present in the system. It uses this knowledge to make query scheduling decisions with the goal of maximizing overall throughput. Using a set of dedicated I/O processes, the scheduler directs asynchronous data transfers from the tertiary memory to disk in a highly optimized manner. Each user process executes queries on data cached on disk instead of independently doing I/O on tertiary memory. This enables the scheduler to batch I/O from multiple user processes on tertiary memory and do effective re-use of cached data.

- We employ the notion of a fragment to reveal the layout of the relation on tertiary memory to the scheduler and query executors. Query scheduling decisions are made in units of subqueries on fragments and data movement decisions are made in units of fragments. This avoids small random I/Os, common in many conventional query execution methods.

- We use a modified query execution engine that can (1) extract information about the data items needed by a query before execution begins, and, (2) can work in cooperation with the scheduler to execute queries in a reorderable fashion. By allowing execution to be reordered, we provide more flexibility to the scheduler for optimizing access order. Designing such an executor was challenging because of three factors: (1) there could be ordering constraints between operators (2) there could be dependency between operators and (3) the number and order of subqueries executed together is not known in advance. We propose a design for modifying the execution engine such that the necessary changes are well modularized into three meta-operators and two extra phases before execution. We provide a method for handling dependencies by adding another meta-operator that can extract subquery information after partial execution. This enables us to efficiently handle index scans, nested loop joins (with index scans on the inner), selections on relations pointing to large objects and other operators that have data dependency.

Chapter 3

Subquery Scheduling

This chapter presents the design of the scheduler — an important component of our proposed architecture. First, we briefly review the working of the scheduler in Section 3.1. Then, in Section 3.2, we discuss the scheduling policies. Section 3.3 presents an evaluation of our fragment fetch and eviction policies using simulation experiments.

3.1 Working of the scheduler

The scheduler is a centralized unit that collects subqueries from each user-process and decides on the order of scheduling them. To make query scheduling decisions for overall global benefit, the scheduler maintains system-wide information about the state of the tertiary memory, the contents of the cache, and the data requirements of each subquery in the system. It uses this global knowledge for making decisions with the goal of maximizing overall system throughput. The various steps involved in the query scheduling process are:

1. From the pool of pending subqueries (each associated with the fragments it needs), the scheduler selects the fragment to be fetched next when an I/O process becomes free. Section 3.2.1 discusses how the scheduler makes this decision.

2. Then it selects data to be replaced from the disk cache to make space for transferring the selected fragment. Sometimes, there might be no data eligible for eviction from the cache. In such a case, the scheduler waits until some subquery completes, releasing space occupied by the data the subquery used. The fragment eviction policies are discussed in Section 3.2.2.

3. The scheduler then instructs the I/O process to transfer the selected fragment.

4. Then it scans the list of pending subqueries and marks "ready" the subqueries that access only cached data. In addition, any subquery for which the I/O process has been instructed to transfer data are also marked "ready", even if the transfer is not completed. This helps to reduce the wait time since the data transfer can be pipelined with the execution of the subquery.

5. After selecting the "ready" subqueries, the scheduler pins all data that are touched by these subqueries. Pinning data involves incrementing the reference count on the data. Pinned data cannot be removed from the disk cache. This ensures that the data needed by a subquery is cached during the entire duration of its processing.

6. When the scheduler is notified of the completion of a subquery, it decrements reference count of all data the subquery touched. When the reference count of a fragment is 0 it is "un-pinned".

3.2 Scheduling policies

The scheduler has a collection of subqueries obtained from multiple user-processes. As far as the scheduler is concerned each subquery is simply a collection of fragments that are needed together. We first concentrate on scheduling subqueries that require at most two fragments in the cache together and require whole fragments instead of particular subsets of blocks in the fragment. Later, Section 3.4 discusses extensions to handle these other cases. The pending collection of subqueries can be represented as a *query graph* where the nodes denote the fragments and the edges denote the joins between two fragments. In such a graph, an edge between two nodes implies that both the fragments represented by these nodes must reside in the cache together for the subquery to be processed. Fragments which do not join with any other fragment are represented as isolated nodes. Figure 3.1 shows the query graph corresponding to the following collection of four queries:

- Query 1 is a nest-loop join query between relations R and S. R consists of three fragments, R_1, R_2, R_3 and S has two fragments, S_1 and S_2. Thus the `SQ-list` for query 1 is:

$$\{(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2), (R_3, S_1), (R_3, S_2)\}$$

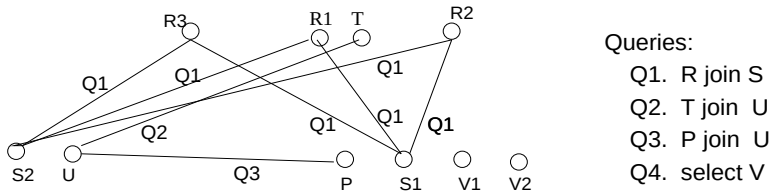


Figure 3.1: An example query graph.

This yields six edges between fragments of R and S as shown in Figure 3.1.

- Query 2 is a join query between relations T and U consisting of one fragment each and thus yields an edge between T and U.
- Query 3 is another join between relations P and U consisting of one fragment each and thus yields an edge between P and U.
- Query 4 is a select query on fragment V. V consists of two fragments V_1, V_2. Thus, the graph has two nodes without any edges from them.

Given such a query graph, the task of the scheduler is to decide on the order of fetching and evicting fragments given the constraints of the limited cache. Typically, the size of the disk cache is less than the sum of the sizes of the fragments queried. Further, the query graph is not static; it changes as new subqueries arrive and pending subqueries get scheduled. Thus, decisions on the order of fetching fragments should be on-line or incremental instead of being off-line or batch. In this on-line setting, at any point in time some of the fragments of the query graph might already be in the disk cache whereas others might need to be fetched from tertiary memory. Of the fragments to be fetched from tertiary memory, some might reside on platters that are currently loaded and others might reside on unloaded platters. Figure 3.2 shows an example setting that can arise for the query graph in Figure 3.1. The fragments R_1 and T are already in the disk cache, fragment S_1 joins with the cached fragment R_1 and resides on platter 1 that is currently loaded whereas fragment U that joins with T will require a platter switch for transferring. Given this setting, our objective is to transfer queried fragments from tertiary memory with the goal of minimizing the total time spent doing I/O on tertiary memory.

This problem is NP-complete. The proof of this follows from the proof of NP-completeness of a much simpler, off-line version of our problem discussed in [MKY81, PI85]

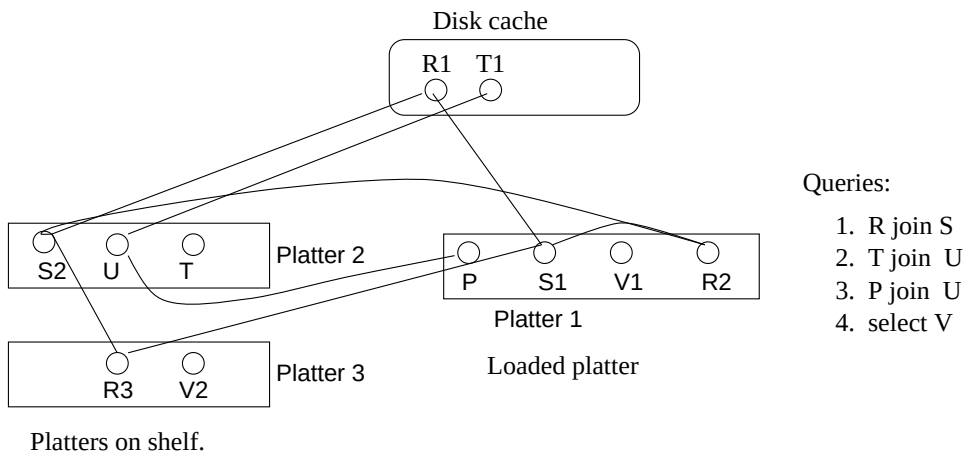


Figure 3.2: A typical online setting.

in the context of two-way join query graphs. [MKY81] formulates the two-way join query as a bipartite graph where the nodes are the pages of the individual relations and an edge between two nodes denotes the presence of a matching tuple between the two pages. Their goal is to find a schedule for transferring pages from disk to a limited main memory buffer so as to minimize the number of pages transfered. By reducing the Hamiltonian path problem to this problem, [MKY81] proves the two-way join problem to be NP-complete. Our problem has various additional challenges:

- fragments can be of varying length whereas pages are fixed length;
- the I/O cost has three components; transfer cost, switch cost and seek cost and sometimes optimizing for one cost component could make another cost component higher; and,
- the query graph changes dynamically.

Hence, an algorithm that finds the optimal solution is likely to be too expensive to be useful. Consequently, we use a number of heuristics for reducing the search space.

We used extensive simulation to aid us in the search for good heuristics and evaluated a large number of variations. For clarity, in this section, we will only present our final resulting set of heuristics. Later, in Section 3.3, we will present experimental justification of our choice.

3.2.1 Fragment fetch policies

When each subquery requires exactly one fragment, it is easy to solve this problem optimally (for the batch case) using the following simple policy. "Fetch fragments from loaded platters in their storage order. Schedule all subqueries on the cached fragment before replacing it". The reason it is optimal (in terms of I/O cost) is because each queried fragment is fetched only once, each platter is loaded only once and from the loaded platters fragments are fetched in their storage order to minimize total seek cost. For the on-line version of this problem, it is not possible to find the optimal solution since we cannot know the future fragment requests. However, we can use the batch solution, to construct an on-line policy as follows:

> MIN-LATENCY: Fetch the fragment from the loaded platters with the smallest latency of access. If there is no queried fragment on a loaded platter, choose the fragment with the largest number of queries. Schedule all queries on the cached fragment before replacing it.

Complications arise because of the presence of joins and limited cache space. Joins require more than one fragment to be present in cache. If the cache space were unlimited, we could still use the MIN-LATENCY policy. The only difference is that we would have to cache a fragment until all fragments that it joins with had been fetched. Since the cache space is limited, the decision to fetch a fragment can be influenced by what is already cached. We cannot fetch a fragment based simply on its location on tertiary memory as in the MIN-LATENCY policy. For instance, the cache could be filled with fragments, all of which require some other fragment to be fetched from cache before any subquery on them can be scheduled. In such a case, we must choose a fragment that joins with cached fragments and thus relieves the load on the cache even if this choice requires forsaking the MIN-LATENCY policy. A natural choice in such cases would be a fragment that relieves the cache of maximum load by joining with maximum sized cached fragments. This alternative criteria for fetching fragments is expressed by the FREE-CACHE policy below.

> FREE-CACHE: Fetch fragment that joins with the largest sum of sizes of cached fragments.

The main issue next is to identify cases where we should choose one policy versus the other. Ideally, we want a fragment that satisfies both the policies. But if that is not possible we need to choose between the two. Clearly, this choice will depend on the

contents of the cache. When the cache gets filled with fragments that need some other fragment from tertiary memory before any subquery on cached fragments can be scheduled then we should use the FREE-CACHE policy. Otherwise, either the cache has lots of unused space or there are no pending joins. Hence, we can reduce I/O cost by fetching fragments using the MIN-LATENCY policy. The final policy, with further details on how we resolve ties between fragments, is given below.

FINAL-POLICY

1. Fetch next fragment that joins with cached fragments and satisfies policy MIN-LATENCY.
2. If no such fragment exists, choose between the two policies
   /* choice technique is explained later in the section */
3. If (policy chosen is FREE-CACHE)
   3a. Fetch next fragment from the loaded platters using FREE-CACHE policy.
      If no such fragment,
         3b. Switch to the unloaded platter that has fragments
            that join with largest sum of sizes cached fragments
            3c. Fetch fragment from the chosen platter using the FREE-CACHE policy
4. Else /* policy chosen is MIN-LATENCY*/
      4a. Fetch fragment from loaded platter using policy MIN-LATENCY.
         If no fragment on the loaded platters is of use,
            4b. Load the platter which satisfies maximum pending queries
            4c. Fetch fragment using policy MIN-LATENCY from the newly loaded platter.

Explanation of the algorithm:

- In step 1, we first attempt to fetch a fragment that satisfies both policies. Note that, if there is no fragment that joins with any cached fragment, policy FREE-CACHE is trivially satisfied. Also, when a fragment does not have any join query on it (for example, fragment V_1 in Figure 3.2), we assume that it satisfies policy FREE-CACHE. For the example in Figure 3.2, fragment P is nearest to the tape head and hence would be chosen by policy MIN-LATENCY, but it does not join with any of the cached fragments. Hence, step 1 fails to return a fragment.

- In step 2, we choose between one of the two policies. Qualitatively speaking, we choose the FREE-CACHE policy if the *load* on the cache is high enough that the cache

is a limited resource. The load is high when the cache is filled with fragments that have many pending join queries on them. Quantitatively, we define a notion of *cache pressure* to characterize the load on cache as follows:

$$\text{Cache pressure} \quad = \quad \frac{\texttt{join-cached} + \texttt{join-uncached}}{C},$$

where C is the size of the cache, `join-cached` is the total size of cached fragments with pending joins and `join-uncached` is the total size of uncached fragments that join with cached fragments. Hence (`join-cached` + `join-uncached`) is an estimate of the amount of cache space that will be needed in the future. The cache pressure expresses potential demand for the cache as a fraction of the cache size. In Figure 3.2, suppose the size of each fragment is 1 GB and the size of the cache is 4 GB, then the cache pressure is:

$$\frac{(|R_1| + |T|) + (|S_1| + |S_2| + |U|)}{|C|} = (2+3)/4 = 1.25$$

We can use cache pressure to determine if there is any room for fragments that do not join with any of the already cached fragments. We next define a constant threshold (α) such that if "cache pressure > threshold", we choose the FREE-CACHE policy, otherwise we choose the MIN-LATENCY policy. We determined the value of the threshold empirically. Section 3.3.2 discusses how this value was chosen.

- If the test in step 2 results in the choice of policy FREE-CACHE, we need to fetch a fragment that joins with one of the cached fragments. In step 3(a), we first attempt to choose such a fragment from one of the loaded platters, if possible. Note that a fragment so found will not be the optimal choice in terms of seek cost since step 1 has failed. If we cannot find such a fragment, we need to load a new platter to get a fragment that joins with one of the cached fragments. In step 3(b), we choose a platter that contains the fragments that join with the largest sum of sizes of cached fragments. The reason for such a choice is to be able to free as much cache space as possible in one platter load. Next, in step 3(c) we choose a fragment from the newly loaded platter using the FREE-CACHE policy. We resolve ties by choosing a fragment that has lowest seek cost. In Figure 3.2, suppose that the value of the threshold α is 1. Then, since the cache pressure is 1.25, we need to choose a fragment using policy FREE-CACHE. We choose fragment S_1 since it joins with a cached fragment

and resides on a loaded platter(step 3(a)). After S_1 is cached, the subquery (R_1, S_1) is scheduled.

- If the test in step 2 chooses policy MIN-LATENCY, then we simply fetch a fragment from the loaded platter with the goal of reducing the seek cost. We resolve ties by choosing a fragment that has more queries on it. When the currently loaded platter has no queries on it, we need to choose a new platter. In step 4(b) we choose the platter with the maximum number of queries on it for loading next. Finally, we select a fragment from this newly loaded platter using policy MIN-LATENCY.

3.2.2 Fragment eviction policies

Once a fragment is selected for fetching, we choose fragments to be replaced on the cache to make space for the selected fragment. Like the fetch policy, our eviction policy is also based on the careful combination of a number of simple heuristic policies.

The classical cache replacement policy is LRU when all objects are of the same size, and WEIGHTED-LRU when the objects are of varying size. These policies are not appropriate in our case for two reasons: First, we might have to evict fragments which have pending queries on them. This makes policies like LRU and WEIGHTED-LRU inappropriate since we already know that the fragment will be used in the future. Second, the cost of fetching data varies widely depending on its location. When choosing to replace a fragment, it helps to take this cost difference into account, instead of simply relying on the time of last access.

We now discuss how we choose a fragment to evict. Our choice is restricted to fragments that are not pinned. Also, as soon as all data required by a subquery is present in the cache, we schedule the subquery and pin the data it requires. This implies that the candidates for replacement are either fragments that only join with uncached fragments or fragments with no pending query on them.

We first discuss how we choose amongst fragments that have pending queries on them. Our choice of a victim fragment is based on the probability that the fragments it joins with will be fetched in the near future. This probability depends on the scheduling policy. When the cache pressure is low, we use the MIN-LATENCY policy for fetching fragments and when it is high we use the FREE-CACHE policy. Correspondingly, we define two policies for evicting fragments.

The LEAST-LATENCY policy is used for evicting fragments when the cache pressure

is low. This policy chooses a cached fragment F based on the latency of accessing the fragments that F joins with. For instance, when the cache pressure is low we fetch fragments from the loaded platter first. Thus, any cached fragment that joins with many fragments on the loaded platter has a higher probability of access in the near future than the one that joins with fragments on unloaded platters. The LEAST-LATENCY policy is defined as:

> LEAST-LATENCY: Fetch fragment with the smallest rank F where rank of a fragment, F is the sum of the inverse latency of accessing each fragment that joins with F.

For example in Figure 3.2, the rank of fragment R_1 is

$$\frac{1}{\text{seek cost to } S_1} + \frac{1}{\text{platter switch} + \text{seek cost to } S_2}$$

Similarly, the rank of T is

$$\frac{1}{\text{platter switch} + \text{seek cost to } U}$$

Thus, policy LEAST-LATENCY would choose T to be replaced in preference to R_1.

When the cache pressure is high, fragments will be fetched based on the FREE-CACHE policy that chooses fragments based on the joins it forms with cached fragments. In such a case, it is necessary to base eviction decisions on the contents of the cache and the relationship of the fragment with other cached fragments. For example, in Figure 3.3 fragments F_1, F_2, F_3 are cached, fragment F_1 and F_2 both join with fragment F_4 whereas F_3 joins with a different fragment F_5. If we need to evict a fragment, it is better to choose F_3 instead of F_1 or F_2 since the latter two join with the same fragment. If F_1 and F_2 are in the cache together, then we could complete both joins when F_4 is fetched next. This motivates policy LEAST-OVERLAP defined as:

> LEAST-OVERLAP: Choose the fragment with the least overlap between fragments that join both with the given fragment and other cached fragments. The overlap for a fragment, F is the sum of the size of the cached fragment, G whenever both F and G join with a common fragment.

The next issue is how we select between fragments without any queries on them. The classical method is to use WEIGHTED-LRU. But for tertiary memory devices an important consideration is the cost (in terms of seek and switch cost) involved in fetching a fragment. For example, everything else remaining the same, it is better to replace a fragment that lies on the currently loaded platter because the cost of re-fetching it is smaller

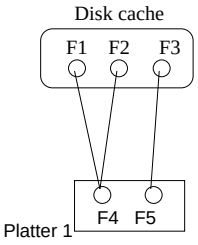


Figure 3.3: Illustration of LEAST-OVERLAP policy.

than for fragments on unloaded platters. This cost must also include the probability of accessing the replaced fragment in the future. For fragments with pending queries, this probability is 1 whereas for others we approximate it with one over the time since last access. This yields the MIN-COST policy that chooses the fragment with the smallest value of the product of the cost of replacement and the probability of future access. This policy is similar to the one discussed in [Yu95] for replacing blocks on the disk cache.

The final policy after combining these three policies is:

If (cache pressure is low)
    Choose fragment using LEAST-LATENCY
    Resolve ties using LEAST-OVERLAP
    Resolve further ties using MIN-COST
Else
    Choose fragment using LEAST-OVERLAP
    Resolve ties using LEAST-LATENCY
    Resolve further ties using MIN-COST

Note that the decision to evict a fragment G is made only after another fragment, F, is chosen for transfer. This enables us to avoid evicting fragments that join with F. Also, after choosing a fragment G for eviction we use the above merit criteria to compare it with F — only if F joins with one of the cached fragments or F has higher merit than G to be in cache (based on the above criteria), do we choose to replace G for F. Otherwise, the scheduler waits until some pinned fragments gets unpinned as a result of the completion of scheduled subqueries on them.

Example: We will continue with our example in Figure 3.2 to illustrate the working of the final fragment fetch and eviction algorithm. Assume the tape head is at the beginning of platter 1 and the order of labeling fragments from left to right in the figure corresponds to the access order with the lowest seek cost. Step 1 of the fragment fetch policy fails since

fragment P satisfies policy MIN-LATENCY but not FREE-CACHE. The cache pressure at this point is 1.25 (shown earlier) which is greater than the threshold of 1. Hence we choose fragment S_1 (using step 3a). Since, both R_1 and S_1 are cached we schedule subquery (R_1, S_1). Next, we choose fragment V_1 (via step 1) since it satisfies both the policies. We schedule the only subquery on it. Next we choose fragment R_2 (via step 1) since it satisfies both the policies. The cache has space for only four fragments, hence we need to choose a fragment to replace with R_2. We replace V_1 based on policy LEAST-LATENCY. Subquery (S_1, R_2) is now ready for execution. In choosing the next fragment, step 1 again fails since fragment P joins with a fragment that is not cached. Note that at this point, the fragments in cache are R_1, T, S_1 and R_2. In step 2, we evaluate the cache pressure to be

$$\frac{(|R_1| + |T| + |S_1| + |R_2|) + (|S_2| + |U| + |R_3| + |S_3|)}{|C|} = \frac{4+4}{4} = 2$$

The cache pressure is > 1, so we resort to the FREE-CACHE policy. Step 3(a) fails, hence we choose platter 2 to load next according to step 3(b). We then choose S_2 via the FREE-CACHE policy (step 3(b)). To make room for S_2 we have to replace a fragment. R_1 and R_2 join with S_2, thus the choice is between S_1 and T. We choose to evict T since the cache pressure is high and S_1 has greater overlap than T (S_1 and S_2 join with the same fragment R_3). After caching S_2, we schedule subqueries (R_1, S_2) and (R_2, S_2). We next choose fragment U using step 1. The only un-pinned fragment is S_1 but U scores lower than S_1 in terms of merit for being cached. Hence, we wait for some fragment from R_1, S_2 and R_2 to be unpinned. Suppose both subqueries are completed. We evict R_1 and R_2 and replace them with U and T and schedule subquery (U, T) on them. The cache now has fragments S_1, S_2, U and T. When choosing the next fragment, step 1 again fails since the loaded platter 2 does not contain any fragment that joins with cached fragment. We re-evaluate cache pressure to be $\frac{2+1}{4} = 0.75$ which is less than the threshold 1. Hence, we go to step 4 and choose a new platter to load. Between platters 1 and 3, we choose platter 3 since it has more queries on it (step 4(b)). After loading platter 3, we fetch fragments R_3 (step 1), evict T to make space for R_3, schedule subqueries (R_3, S_1) and (R_3, S_2), choose V_2 (step 1), replace R_3 to make space for V_2 after subqueries (R_3, S_1) and (R_3, S_2) complete (via MIN-COST), load new platter 1 (step 4b), choose fragment P (via step 1), replace fragment S_2 (via policy MIN-COST) and finally schedule subquery (P, U).

	Sony	Exabyte	Metrum	DMS
classification	small optical jukebox	small tape library	large tape library	large tape library
switch time (sec)	8	171	58.1	39
transfer rate (MB/sec)	0.8	0.47	1.2	32
seek rate (MB/sec)	-	36.2	115	530
seek start (sec)	0.5	16	20	5.0
number of drives	2	4	5	2
platter size (GB)	3.27	5	14.5	41
number of platters	100	116	600	320
total capacity (GB)	327	580	8700	13120

Table 3.1: Tertiary Memory Parameters: The switch time is a sum of the average time to rewind any existing platter, eject it from the drive, move it from the drive to the shelf, move a new platter from shelf to drive, load the drive and make it ready for reading. The seek startup cost is the average of the search and rewind startup cost and the seek rate is the average of the search and rewind rate.

3.3 Simulation

This section presents empirical evaluation of our fragment fetch and eviction policies. The final policy presented in Sections 3.2.1 and 3.2.2 is the result of an extensive simulation study. In designing these policies, we first started with a number of different heuristics that made intuitive sense and evaluated these heuristics under a variety of different parameters. No single policy that we originally started with performed well under all different parameter settings. But this evaluation isolated cases where certain heuristics performed better than the others. This guided the design of the final fetch and eviction policy presented earlier.

First, Section 3.3.1 presents details of the simulator used for comparing the policies. Section 3.3.2 contains an empirical method for finding the best threshold value for the fragment fetch policy in Section 3.2.1. Section 3.3.3 contains definition of various other fragment fetch heuristics that we considered and discusses our motivation for choosing them. Section 3.3.4 presents a comparison of these heuristics and the final policy (described in Section 3.2.1) against different parameter settings including, tertiary memory characteristics, size and number of relations, number of concurrent users, cache size and the percentage of join queries. Finally, Section 3.3.5 contains an evaluation of different fragment eviction policies.

Dataset	# relations	range of sizes	total size
SMALL-DATASET	2000	5 MB to 50 MB	50 GB
MEDIUM-DATASET	400	250 MB to 2.5 GB	500 GB
LARGE-DATASET	80	12.5 GB to 125 GB	5 TB

Table 3.2: Datasets: sizes of the relations are uniformly distributed across the given range

3.3.1 Simulation setup

Our simulator consists of a centralized database system serving requests from different query streams. We model a closed queuing system consisting of multiple users who submit a query, wait for the result, and then think for an exponentially distributed time before submitting the next query. Table 3.1 lists the performance specifications of the four tertiary memory types we used in our study: (1) the Sony WORM optical jukebox, (2) the Exabyte 8500 tape library, (3) the Metrum RSS6000 tape jukebox and (4) Sony's DMS tape library. These devices were chosen so as to cover adequate representatives from the diverse tertiary memory hardware in existence today. Table 3.2 lists the three datasets that we used as the underlying database. Each dataset is characterized by the range of sizes of the relations and the number of relations. The size of a relation is assumed to be uniformly distributed within the range specified by the dataset. Note that the total size of some of the datasets (e.g., LARGE-DATASET) is larger than the capacity of some tertiary memory devices (e.g., the Sony WORM and the Exabyte). Thus, some of the <tertiary-memory, dataset> combinations are missing in our experiments. The default size of the cache and the number of users is given in Table 3.3. Further details about the simulator are given below:

Relation layout For laying out the relations on tertiary memory we use the following approach: We divide any relation larger than the platter capacity into partitions of size equal to the platter capacity. These partitions are laid out contiguously on the platters. A partition is stored with equal probability in one of the partially filled platters that has space for it or a new platter if one is available. The space between two adjacent partitions is uniformly distributed between 0 and the total free space left on the platter divided by the number of partitions assigned to the platter.

Workload Table 3.3 summarizes the relevant workload parameters and their default values. We simulate a stream of single relation queries and two-way joins. Base relations for queries are chosen using the 80-20 rule i.e, 80% of the accesses refer to 20% of the relations. The scan on the base relation can be either a sequential scan, a clustered index scan or an unclustered index scan as summarized in Table 3.3. We assume in these experiments that all indices reside on magnetic disks and the index tree is pre-scanned to get a list of fragments that contain qualifying tuples. Various workload parameters for constructing queries are given in Table 3.3.

Execution model The processing time of a query after the component fragments are fetched from tertiary memory is computed as the sum of the time needed to read/write data between disk and the main memory and the CPU processing time. In Table 3.3, we list the number of instructions required for various query types. The time to process a join is derived assuming a nest-loop join method. The time to read a page from disk is modeled as a sum of the average seek time and the time to transfer a page of data.

Fragment Size For a given database and tertiary memory, we determine a maximum fragment size, F. Any partition larger than size F is divided into fragments of size at most F. As shown in Section 2.2.1 of Chapter 2, the maximum size of a fragment is limited by a number of factors, including, cache size, number of fragments per subquery, number of concurrent users, number of drives, scheduling overhead, block size for tertiary memory etc. Based on these factors we derived a number of constraints for guiding the choice of the fragment size. We used these constraints for determining F for each tertiary memory, dataset pair as follows:

1. The smallest value of the cache size C in our experiments is 1% of the database size. Thus, $C = 500$ MB for SMALL-DATASET, 5 GB for MEDIUM-DATASET and 50 GB for LARGE-DATASET. The maximum number of fragments per subquery, n, is 2 since we allow only 2-way and single relation queries. Thus, fragment size has to be ≤ 250 MB for SMALL-DATASET, 2.5 GB for MEDIUM-DATASET and 25 GB for LARGE-DATASET.

2. The number of drives, d is 2 for the Sony WORM, 4 for Exabyte, 5 for the Metrum and 2 for the DMS. For each tertiary memory, dataset pair this limits fragment size to $\leq C/d$ where C is defined for each dataset in constraint 1 above.

Description	Default
Workload	
Mean think time	100 sec
Number of queries per run	800
Number of users	80
% of join queries	50
% sequential scans	20
% clustered index scans	40
% unclustered index scans	40
Selectivity	0.1-0.2
Execution Model	
MIPS	50
Instructions for seq scan	100 per tuple
Instructions for index scan	200 per tuple
Instructions for join	300 per tuple
Instructions for starting a scan	20,000
Tuple size	400 bytes
Disk Characteristics	
Average seek time	20ms
Data transfer rate	5 MB/sec
Cache size	3% of database size

Table 3.3: Simulation Parameters and their default values.

Tertiary Memory	SMALL DATASET	MEDIUM DATASET	LARGE DATASET
Sony	4	-	-
Exabyte	16	128	-
Metrum	16	128	6272
DMS	32	128	6272

Table 3.4: Maximum fragment size (in MB) for each tertiary memory and dataset pair

3. The number of concurrent users is 80 (from Table 3.3), hence $x = 80$ and m is 1.5 (from Table 3.3). Thus, fragment size is desired to be ≤ 4.1 MB for SMALL-DATASET, 41 MB for MEDIUM-DATASET, and 416 MB for LARGE-DATASET.

4. To keep the scheduling overhead low, we allow no more than 20 fragments per relation. This limits the value of F_{min} to (50 MB/20) = 2.5 MB for SMALL-DATASET, (2500 MB/20) = 125 MB for MEDIUM-DATASET and (125 GB/20) = 6250 MB for LARGE-DATASET(refer Table 3.2).

5. The value of B depends on the ratio between the transfer rate and the latency of accessing a block from the device. This limits the value of B to 256 KB for the Sony WORM, 16 MB for the Exabyte and Metrum and 32 MB for the DMS.

Table 3.4 shows the final limit on the fragment size obtained based on the above constraints. Note that the choice of the fragment size for SMALL-DATASET had to override constraint 3 in order to satisfy constraint 5 which was more important for performance in our case.

3.3.2 Choosing threshold value

The remaining issue in the design of our fetch policy is the value of the threshold α. In Figure 3.4 we show the total I/O time for different values of α for nine different <tertiary-memory, dataset> pairs for the workload defined earlier. The X-axis represents different values of the threshold and the Y axis is the total I/O time divided by the total I/O time for the FCFS policy. For all our graphs, we use this normalized I/O time as the Y-axis since it enables us to represent widely varying I/O times (for instance, arising from different cache sizes) in the same uniform scale.

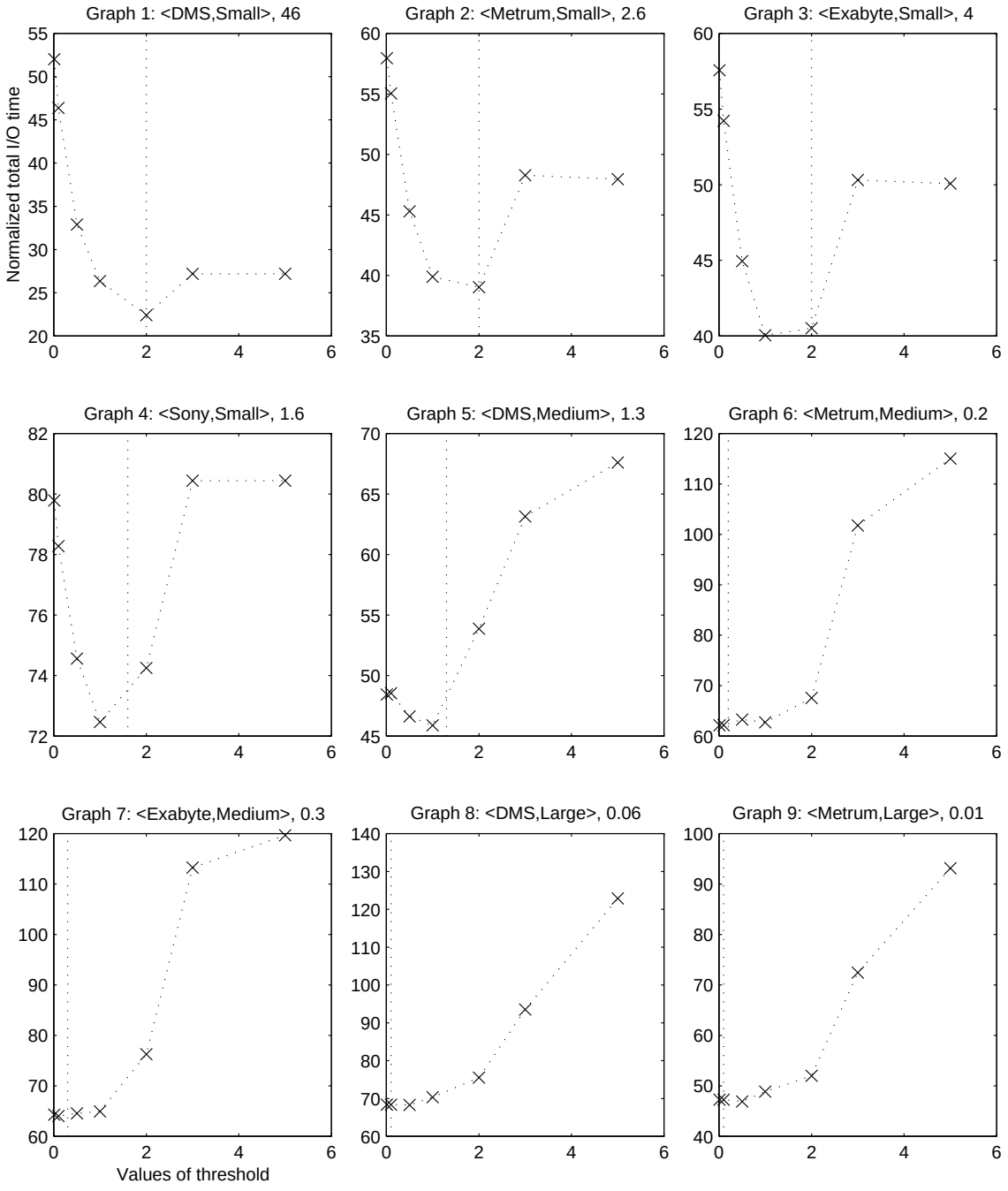


Figure 3.4: Choosing value of threshold. The nine graphs correspond to nine <tertiary-memory, dataset > pairs. The X-axis is threshold values and the Y-axis is total I/O time normalized by the time taken for the FCFS policy.

These graph in Figure 3.4 show that relationship between the threshold value and the I/O time varies widely as we change the <tertiary memory, dataset> pairs. For instance, the graphs for SMALL-DATASET(graphs 1 to 4), favor larger value of thresholds than those for LARGE-DATASET(graphs 8 and 9). Even for the same dataset, the relationship between the threshold value and I/O time varies, depending on the device characteristics. For instance, for MEDIUM-DATASET, the DMS tape library (graph 5) favors larger values of threshold than the Metrum (graph 6). An interesting trend to be observed from these graphs is the relationship between the best threshold values for each <tertiary-memory, dataset> pair and the *ratio of the platter switch time to the average transfer time incurred in fetching a fragment* (called the s-t ratio here). The s-t ratio for each <tertiary-memory, dataset> pair is shown above the title corresponding to the graph in Figure 3.4.[1]. Note that the graphs with the same value of the s-t ratio have the same relationship between threshold and I/O time. For instance, the graphs for <Metrum,MEDIUM-DATASET> (graph 6) and <Exabyte,MEDIUM-DATASET> (graph 7) have very close s-t ratios and hence have similar shapes of the graph. Similarly, <DMS,LARGE-DATASET>(graph 8) and <Metrum, LARGE-DATASET> (graph 9) show the same trend. Thus, <tertiary memory, dataset> pairs with the same value of this s-t ratio favor the same choice of α. This motivated us to choose the value of α as a function of this ratio. Ideally, we would like to get a closed form function that has been analytically derived. However, the complexity of the problem space prevents us from getting one. But, these experiments indicate that a good choice of α is

$$\alpha = \min(2, \text{s-t ratio})$$

In Figure 3.4, the vertical dotted line corresponds to this choice of α. Note that in all cases the vertical line is very close to the minimum I/O time.

The above result for the optimal threshold value also makes intuitive sense. The s-t ratio is essentially the average access latency to fetch a fragment from an unloaded platter in units of the average fragment transfer cost. When I/O latency (switch + seek) is small compared to the fragment transfer time, we select a low value of α. This biases our final policy more towards the FREE-CACHE policy. Whereas when the I/O latency is high, the s-t ratio is high, leading to a higher value of α and greater bias towards the MIN-LATENCY policy. Thus, by choosing α as a function of the s-t ratio we can select the MIN-LATENCY

[1]Note that the platter switch time in Table 3.1 includes the average seek cost whereas when computing the ratio α we subtracted the seek cost from the platter switch cost.

policy to reduce platter switch and seek cost when these costs are high and we can select the FREE-CACHE policy to do better caching when the transfer cost is high.

3.3.3 Fragment fetch heuristics

In this section we define the various fragment fetch policies against which we compared our final policy.

1. MIN-LATENCY: Choose the fragment on loaded platter with the smallest latency of access. If no such fragment exists, choose the platter with the maximum number of queries. The motivation for this policy is to reduce seek and platter switch cost by selecting fragments that incur the smallest access latency. Typically, a device scheduler would choose a policy like this to reduce I/O cost. For the example in Figure 3.2, this policy would choose fragment P next.

2. MAX-WORK: Choose the fragment with the largest number of queries. The motivation for this policy is to increase the number of queries that can be concurrently executed. For Figure 3.2, this policy will result in choice of fragment S_1 or S_2. We resolve ties in favor of fragment on loaded platter.

   The above two policies do not take into account the contents or size of the cache. The following two policies do that.

3. FREE-CACHE: Choose the fragment that joins with the largest sum of sizes of cached fragments. Resolve ties using MIN-LATENCY policy. The motivation for this policy is to make best use of cached data by fetching fragments that they join with. For Figure 3.2, this policy will result in choice of fragment S_1 or U.

4. MIN-COVER: Choose the fragment with the smallest value of *cover*. The *cover* of a fragment f is defined as the total size of uncached fragments that join with f minus the total size of the cached fragments that join with f. The motivation behind this policy is to restrict the amount of data that needs to be cached for joining with fragments already cached. In Figure 3.2, this would result first in the choice of fragments V_1, V_2 and then fragment U in preference to S_1 since U has a cover value of zero (since U joins with one cached and one uncached fragment of 1 GB each) whereas S_1 joins has a cover value of 1 (since S_1 joins with one cached and two uncached fragments of 1 GB each.)

The above four policies make decisions on a per-fragment basis. An alternative is to make fragment fetch requests based on subqueries to be scheduled next. For these classes of policies, we first choose a query and then schedule fetch requests on the fragments required by this query one after another. We tried the following two variations:

5. LATENCY-QUERY: Select query requiring fragments with the smallest access latency. In Figure 3.2, this would result first in the choice of subquery (R_1, S_1), since R_1 has zero tertiary memory access latency and S_1 requires only additional seek cost but no platter switch. Since, R_1 is already cached, we schedule transfer of S_1 next. Next we will choose subquery (V_1) and then subquery (S_1, R_2) and so on.

6. CACHE-QUERY: Select query requiring minimum amount of extra data to be cached. In Figure 3.2, if fragment V_1 were smaller than S_1 we would select subquery (V_1) first using this policy.

3.3.4 Evaluation of fetch heuristics

In this section we compare the fragment fetch heuristics under different settings of important parameters like the device characteristics, size and number of relations, cache size, number of users and fraction of join queries.

Effect of cache size

In Figures 3.5, 3.6, 3.7 we show the performance of various heuristics for cache size varying from 1 to 30% of the database size for different <tertiary memory, dataset> pairs. These sets of graphs show the effect of the device characteristics, the database characteristics and the cache size on the different heuristics. From these sets of graphs we can make a number of interesting observations.

- The FINAL-POLICY is the best (or very close to the best) compared with all other policies in all cases. Most of the other policies perform close to the FINAL-POLICY in some situations but their performance can be very bad in other cases. For instance, the FREE-CACHE policy is almost identical to the FINAL-POLICY for LARGE-DATASET (the two graphs in Figure 3.7) but for the SMALL-DATASET (graphs in Figure 3.5) the FREE-CACHE policy is bad. For SMALL-DATASET, each relation is small enough

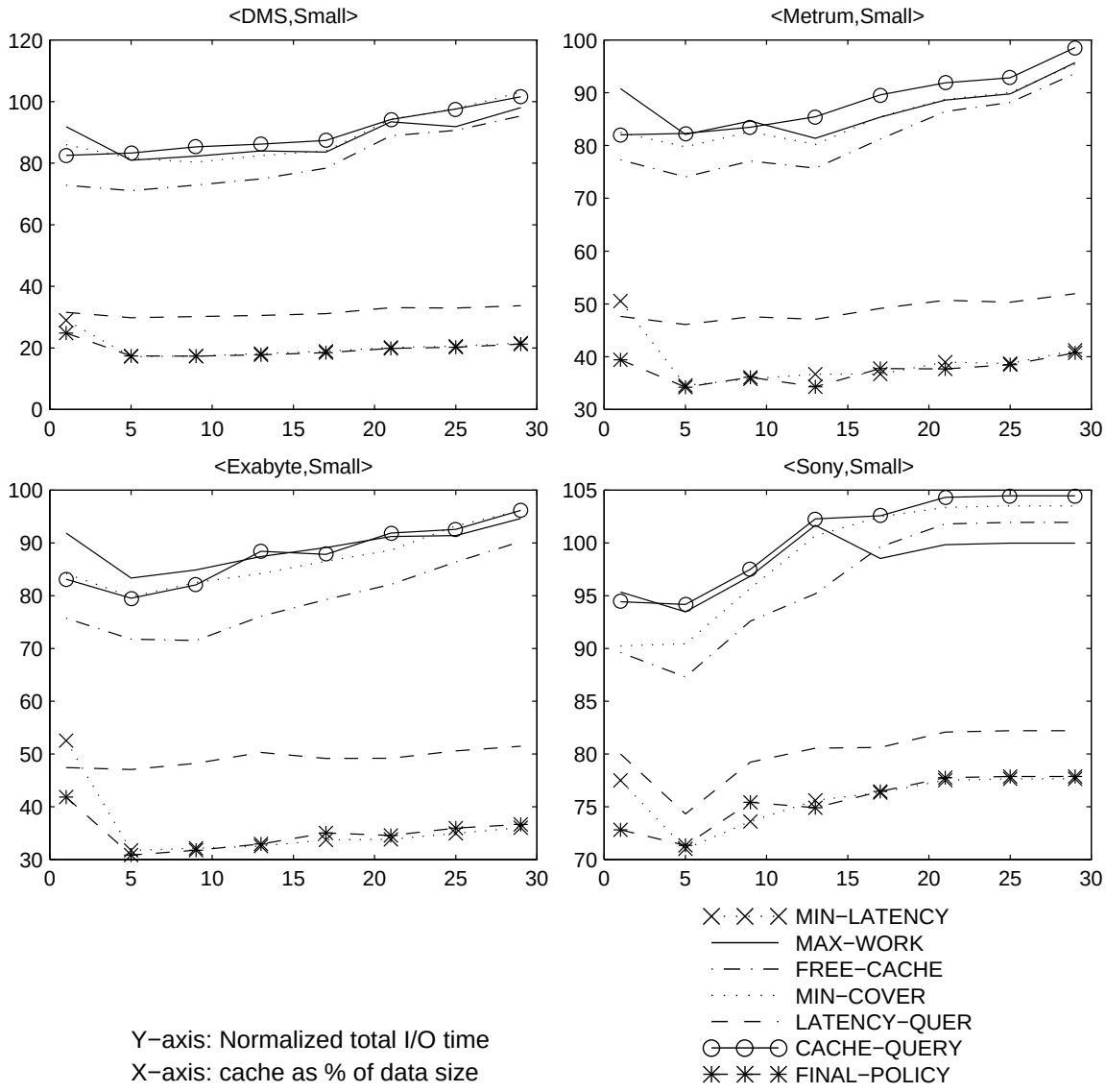


Figure 3.5: Performance of fragment fetch heuristics under varying cache sizes for SMALL-DATASET.

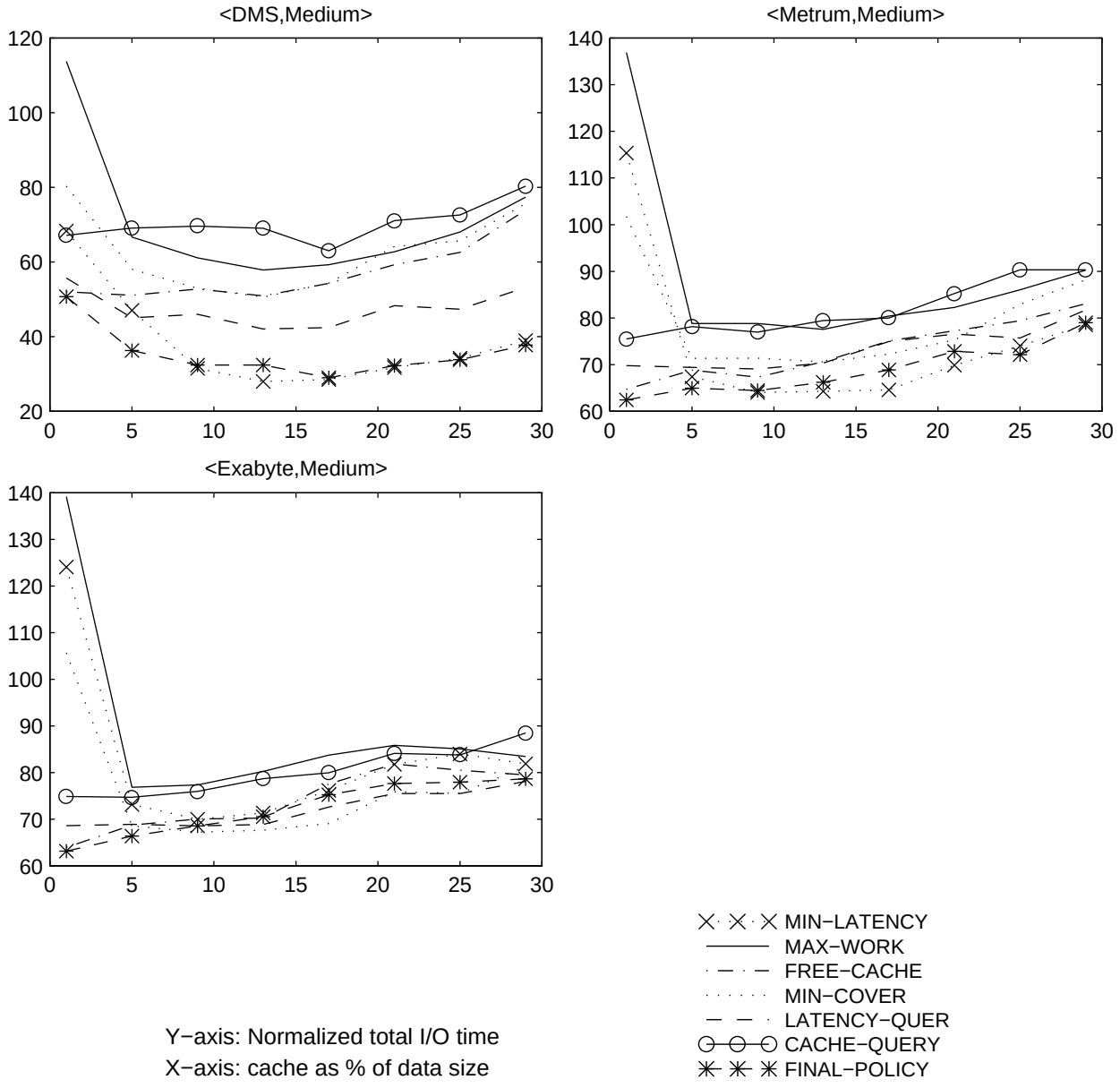


Figure 3.6: Performance of fragment fetch heuristics under varying cache sizes for MEDIUM-DATASET.

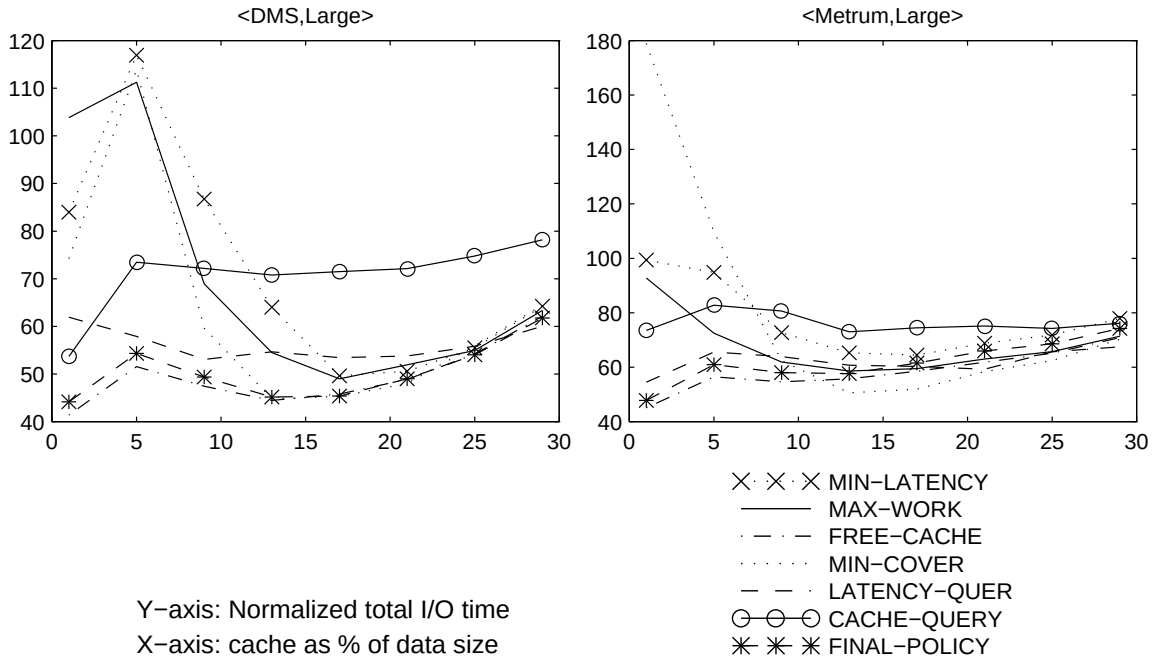


Figure 3.7: Performance of fragment fetch heuristics under varying cache sizes for LARGE-DATASET.

that the transfer time is only a negligible fraction of the total cost. The dominant cost component is platter and switch cost and since the FREE-CACHE policy does not pay any attention to minimizing that cost, its performance suffers for SMALL-DATASET. However, for LARGE-DATASET the transfer cost is the dominant cost. Thus, it is important to minimize the amount of data transfered by doing better cache management than reducing platter switch and seek cost. The MIN-LATENCY policy, therefore, is bad for LARGE-DATASET. On the other hand, for SMALL-DATASET it is almost identical to the FINAL-POLICY.

- The size of the cache is another important parameter. The MIN-LATENCY policy is almost a factor of 2 to 3 worse than the FINAL-POLICY for MEDIUM-DATASET(Figure 3.6) when the cache size is only 1% of the dataset size. However, as we increase the cache size to 10% the MIN-LATENCY policy is almost identical to the FINAL-POLICY.

- LATENCY-QUERY is the only other policy that is close (within 20%) to the FINAL-POLICY in most cases. Hence, this policy can act as a reasonable easy-to-implement substitute. The LATENCY-QUERY policy chooses a subquery first and then fetches *all* the fragments of the subquery in the minimum latency order. This policy is therefore

better than the MIN-LATENCY since it ensures that the cache does not get filled with fragments each of which require some other uncached fragments. It is worse than the FINAL-POLICY policy because it cannot interleave the fetching of fragments belonging to two different subqueries. For instance, in Figure 3.2 we have two join subqueries (R_2, S_2) and (P, U) where R_2, P reside on platter 1 and S_2, U reside on the platter 2. The FINAL-POLICY policy could first fetch R_2, P on platter 1 and then fetch S_2, U from platter 2 whereas the LATENCY-QUERY policy would fetch both fragments belonging to one of the two subqueries first and thus would require one more platter switch.

Effect of number of users

In Figure 3.8, 3.9, 3.10 we show the effect of varying the number of users from 1 to 120 for the different <tertiary memory, dataset> pairs. Again we observe that even for wide changes in the number of concurrent users the FINAL-POLICY performs better than all the others. When the number of users is one, all the policies perform almost identically since there is limited flexibility for optimizing. As we increase the number of users, the number of possibilities for scheduling increases, thus increasing the distinction between the various policies. Also, for higher number of users, the cache starts to become a bottleneck. This causes policies like MIN-LATENCY to perform badly (see the graph for <Exabyte,MEDIUM-DATASET> in Figure 3.9 as an example).

Varying percentage of join queries

From the graphs in Figures 3.11, 3.12, 3.13 showing the total I/O time versus the percentage of join queries for different <tertiary memory, dataset> pairs we can make the following observations:

- The FINAL-POLICY is again better than all the other policies in all cases.
- In all the graphs, when the percentage of join queries is zero, the MIN-LATENCY policy performs identically to the FINAL-POLICY. The FINAL-POLICY in this case reduces to the MIN-LATENCY policy since the cache pressure is always zero.
- As we increase the fraction of join queries, the gap between the two policies grows. For instance for the <Metrum,LARGE-DATASET> pair (in Figure 3.13) the MIN-LATENCY

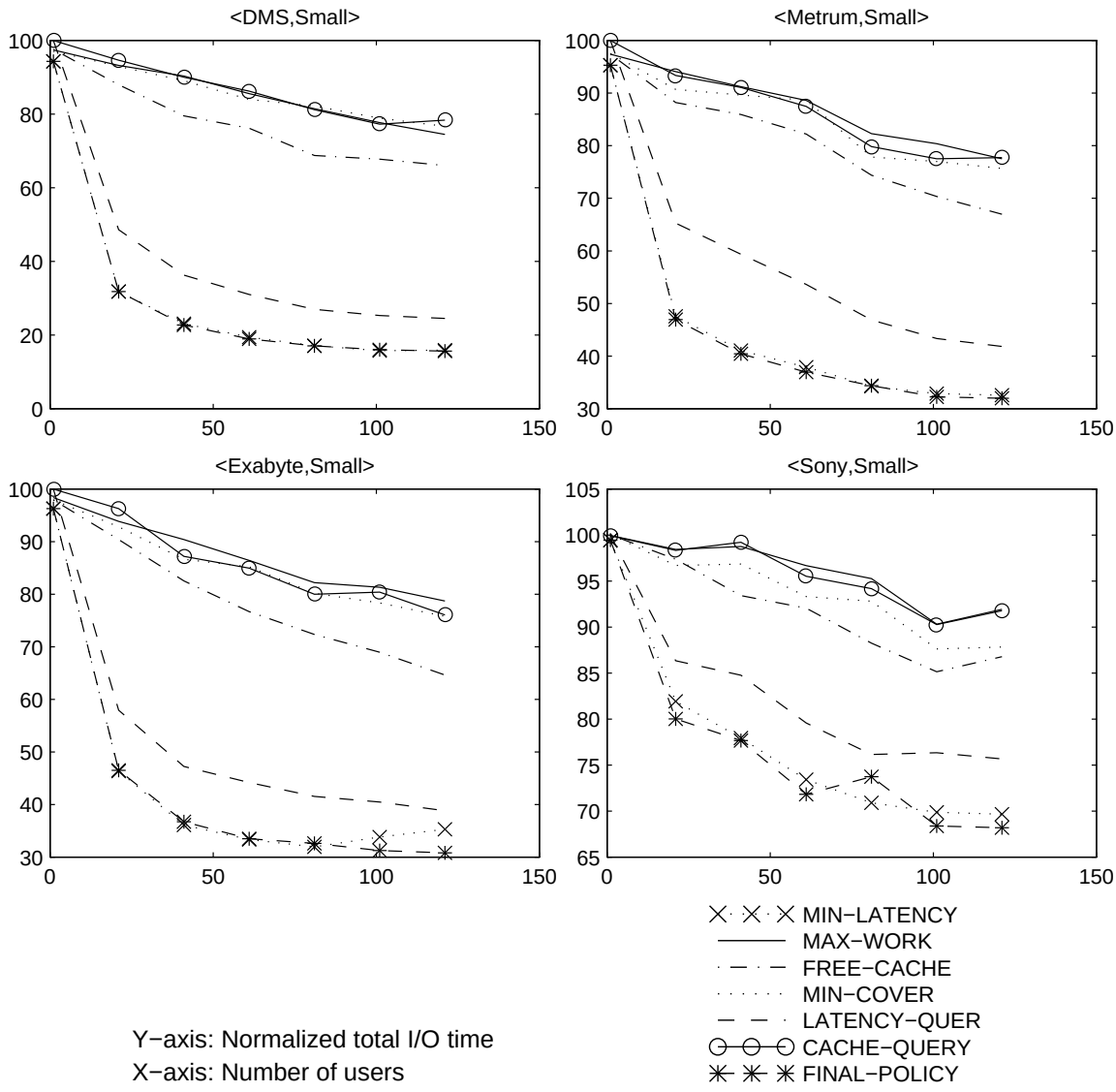


Figure 3.8: Performance of fragment fetch heuristics under varying number of users for SMALL-DATASET.

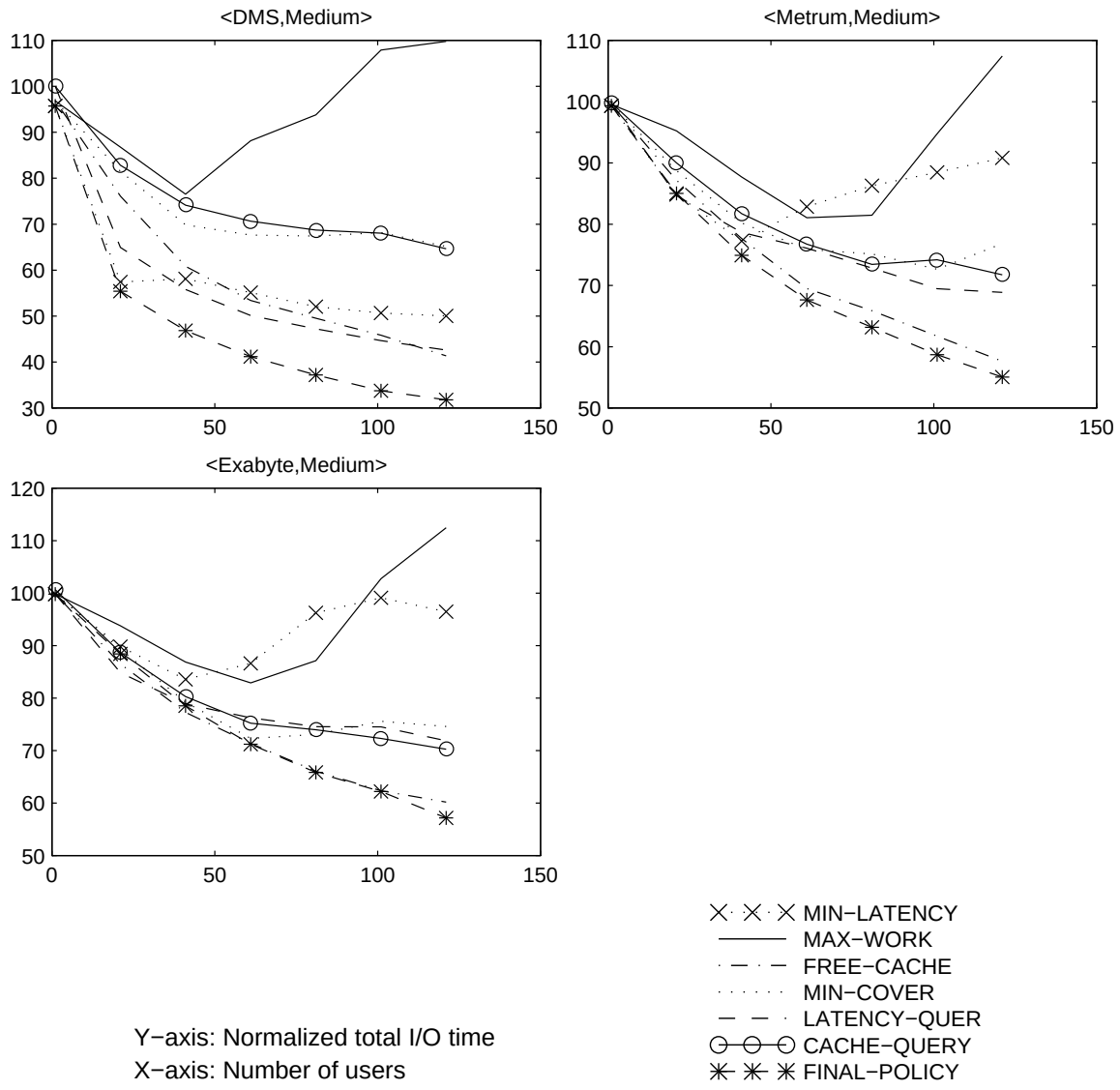


Figure 3.9: Performance of fragment fetch heuristics under varying number of users for MEDIUM-DATASET.

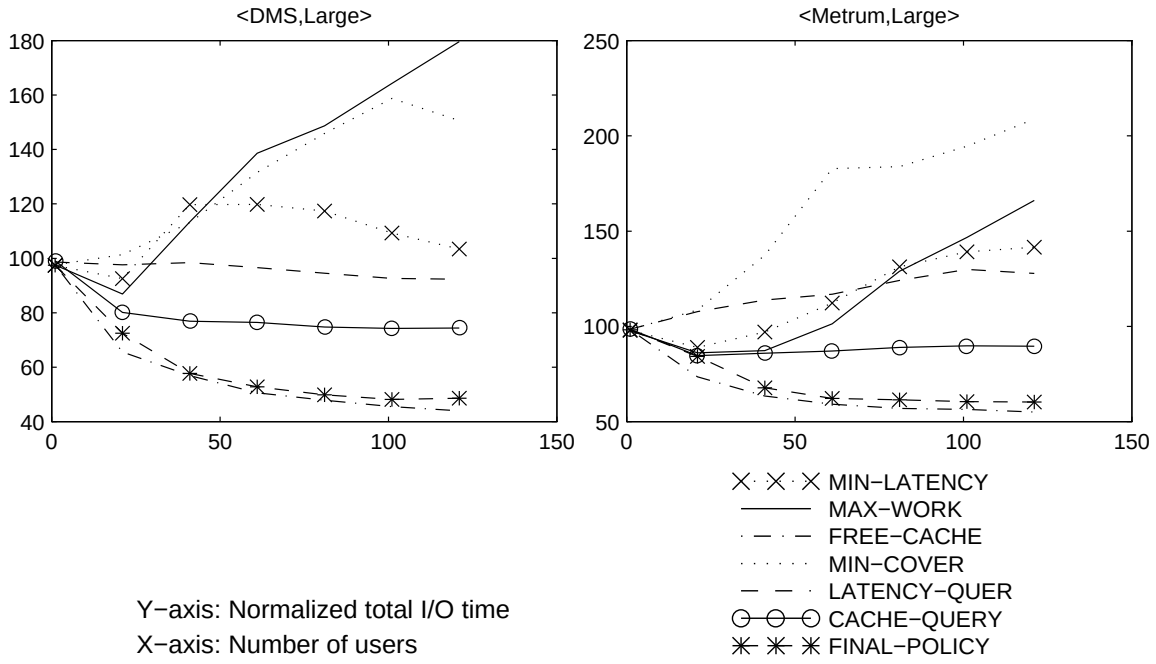


Figure 3.10: Performance of fragment fetch heuristics under varying number of users for LARGE-DATASET.

is almost factor of two worse than the FINAL-POLICY when the percentage of join queries is 90%. For a given join fraction, the difference between the MIN-LATENCY and FINAL-POLICY is higher for the <tertiary-memory, dataset> pairs with larger value of the s-t-ratio.

- When the percentage of join queries is very large the FREE-CACHE policy performs very close to FINAL-POLICY in many cases, e.g., <Metrum,Medium> in Figure 3.12. But the FREE-CACHE is bad when there are fewer join queries. However, the exact value of join fraction for which the FREE-CACHE performs well depends heavily on the dataset and tertiary memory device. For instance, for SMALL-DATASET (Figure 3.11), the FREE-CACHE is almost factor of two worse than the MIN-LATENCY and FINAL-POLICY even when 90% of the queries are joins. The merit of the FINAL-POLICY is that it performs the best irrespective of the join fraction, the tertiary memory device or the dataset.

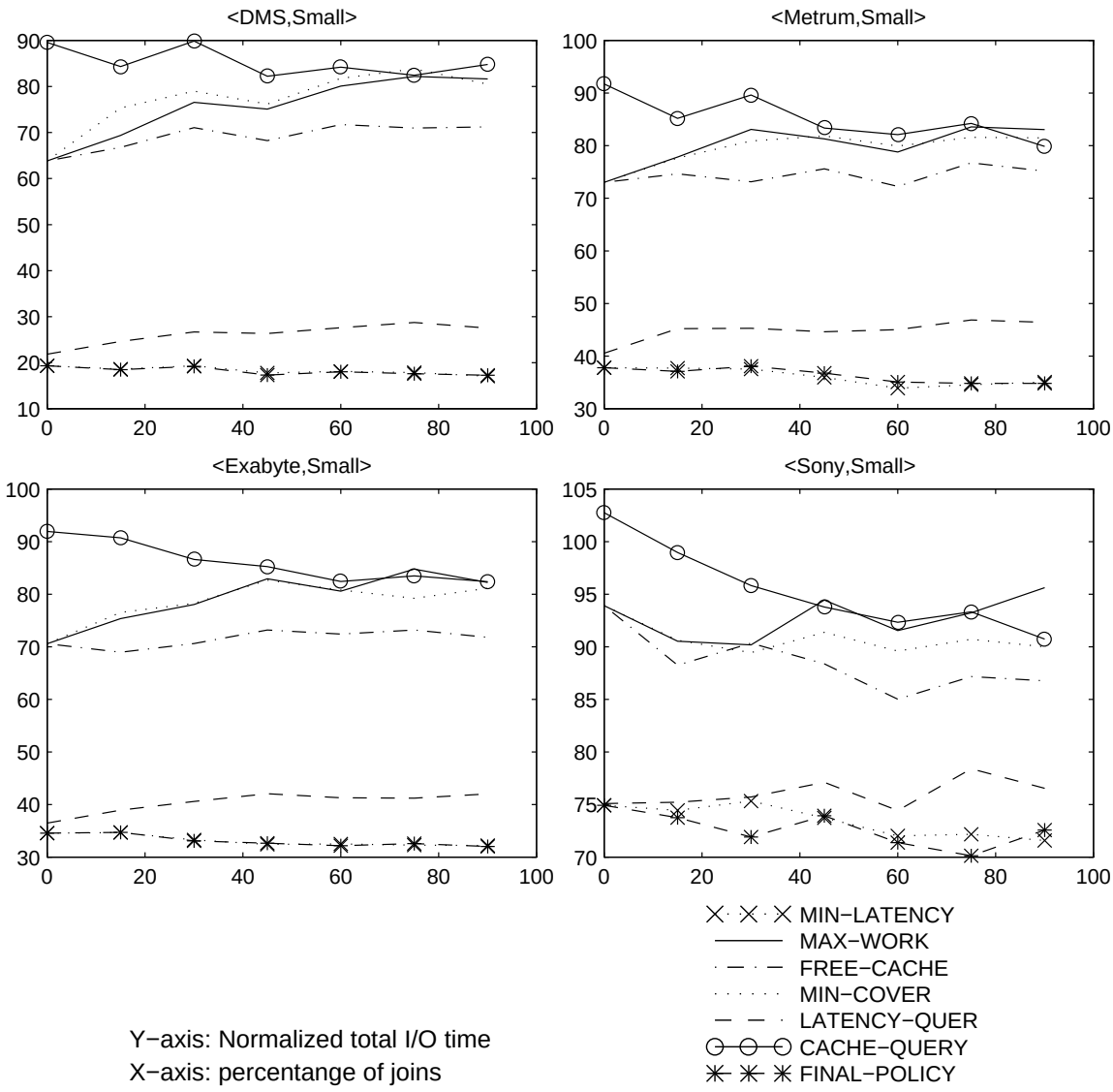


Figure 3.11: Performance of fragment fetch heuristics under varying percentage of join queries for SMALL-DATASET.

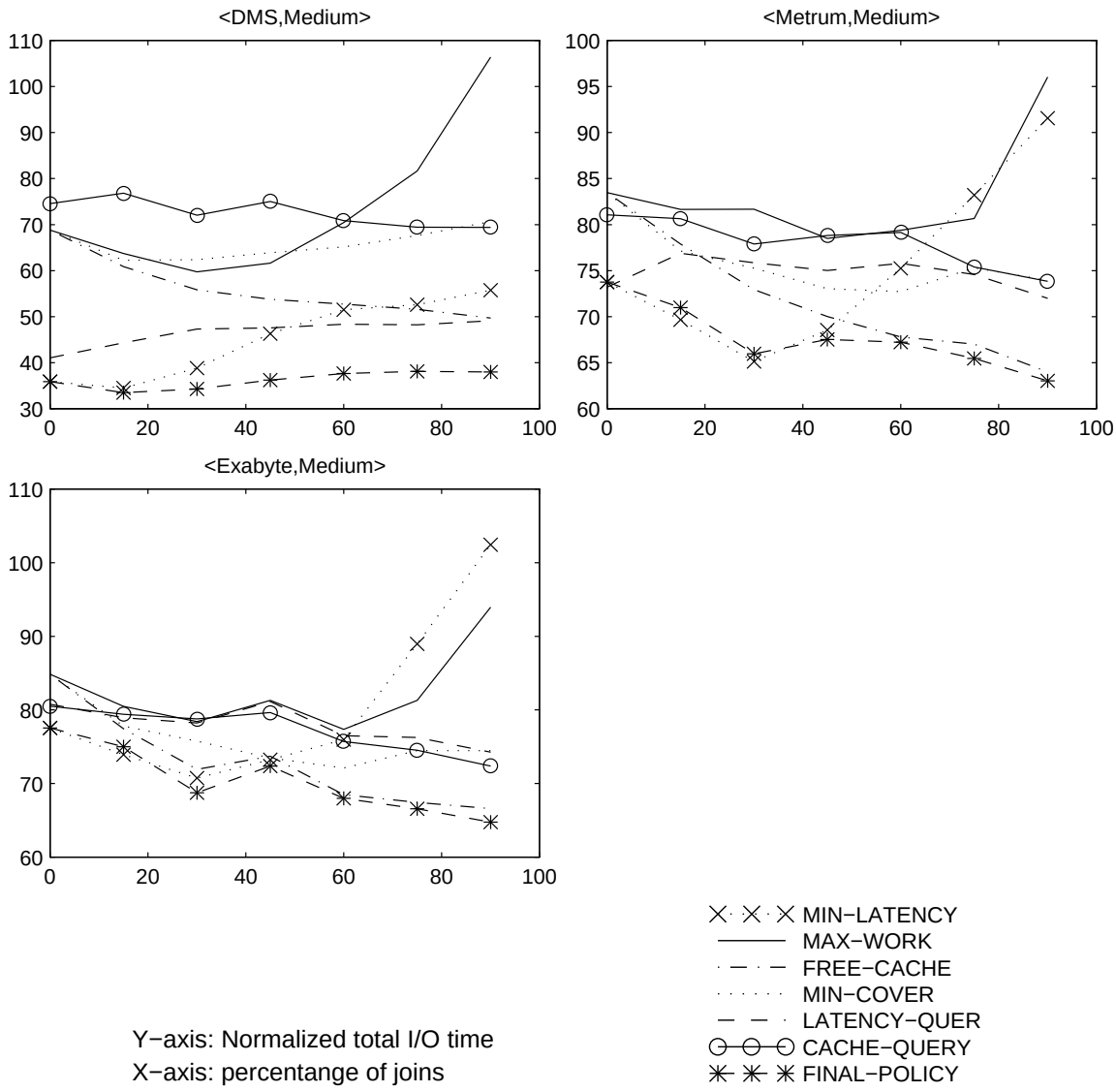


Figure 3.12: Performance of fragment fetch heuristics under varying percentage of join queries for MEDIUM-DATASET.

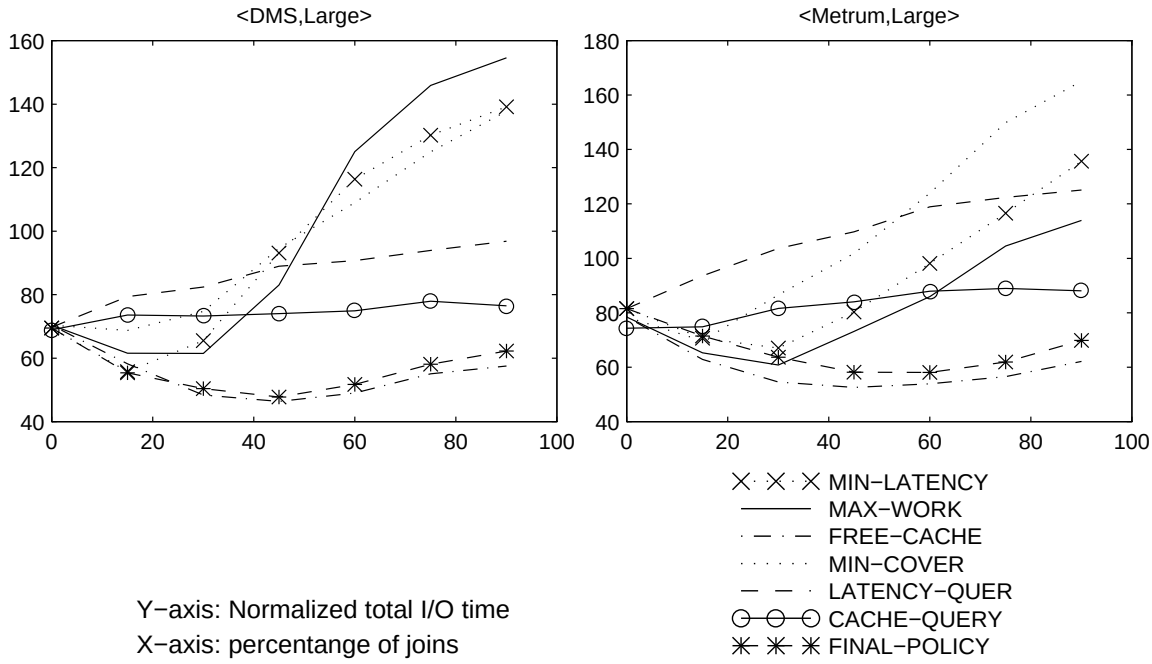


Figure 3.13: Performance of fragment fetch heuristics under varying percentage of join queries for LARGE-DATASET.

3.3.5 Evaluating eviction policy

Like the fetch policy, the eviction policy was also designed by combining the best features of a number of different eviction policies. The different policies that we chose from are:

1. LRU: Replace fragment with largest value for the time of last reference.

2. WEIGHTED-LRU: Replace fragment with largest value of the product of its size and the time of large reference.

3. LEAST-OVERLAP: Replace fragment with the least overlap between fragments that join both with the given fragment and other cached fragments. Resolve ties using LRU.

4. LEAST-QUERIES: Replace fragment with smallest number of pending queries. Resolve ties using LRU.

5. MIN-COST: Replace fragment with the smallest value of the product of the cost of replacement and the probability of future access. For fragments with pending queries

on them, this probability is 1 whereas for others it is one over the time since last access.

Figure 3.14, 3.15, 3.16 shows the performance of the various eviction policies for different cache sizes. The Y-axis is the total I/O time divided by the time taken by the LRU policy for that cache size.

The difference between all the other eviction policies and our final eviction policy is not as significant as in the case of fragment fetch policies. The main reason is that the fetch policy makes good use of cached fragments. Thus we need to rarely evict fragments with pending queries on them. Note that the WEIGHTED-LRU and the LRU policies are bad since they do not take into account the pending queries on fragments. All the other policies evict a fragment with pending queries only when there are no more fragments without any queries on them.

An interesting trend in these graphs is that when the cache pressure is low (for instance, for SMALL-DATASETin Figure 3.14), the LEAST-QUERIES policy is better than the LEAST-OVERLAP policy whereas when the cache pressure is high (for instance, for small cache values in LARGE-DATASETin Figure 3.16) the performance of LEAST-QUERIES is worse. Also, the MIN-COST policy performs badly for LARGE-DATASET (Figure 3.16) because the latency of accessing a fragment is not an significant part of the total I/O cost. Overall, the FINAL-POLICY, in this case too, is better than all the other policies and adopts to changes in the cache size and device and database characteristics.

3.4 Enhancements

In this section we discuss some more features that were added to the scheduler after the initial design. These are extensions made to either further improve the performance of the system or make the scheduler more practical to use.

Maximize drive parallelism When scheduling fragment transfers, it is important to keep as many drives busy as possible to maximize utilization. Hence when choosing a fragment for transfer, the scheduler keeps track of the drives from which data is already being transferred and attempts to schedule the next I/O on a platter from which data is not already being transferred at the time. To enable parallel data transfer from multiple drives, the scheduler maintains as many I/O processes as the the number of drives in the

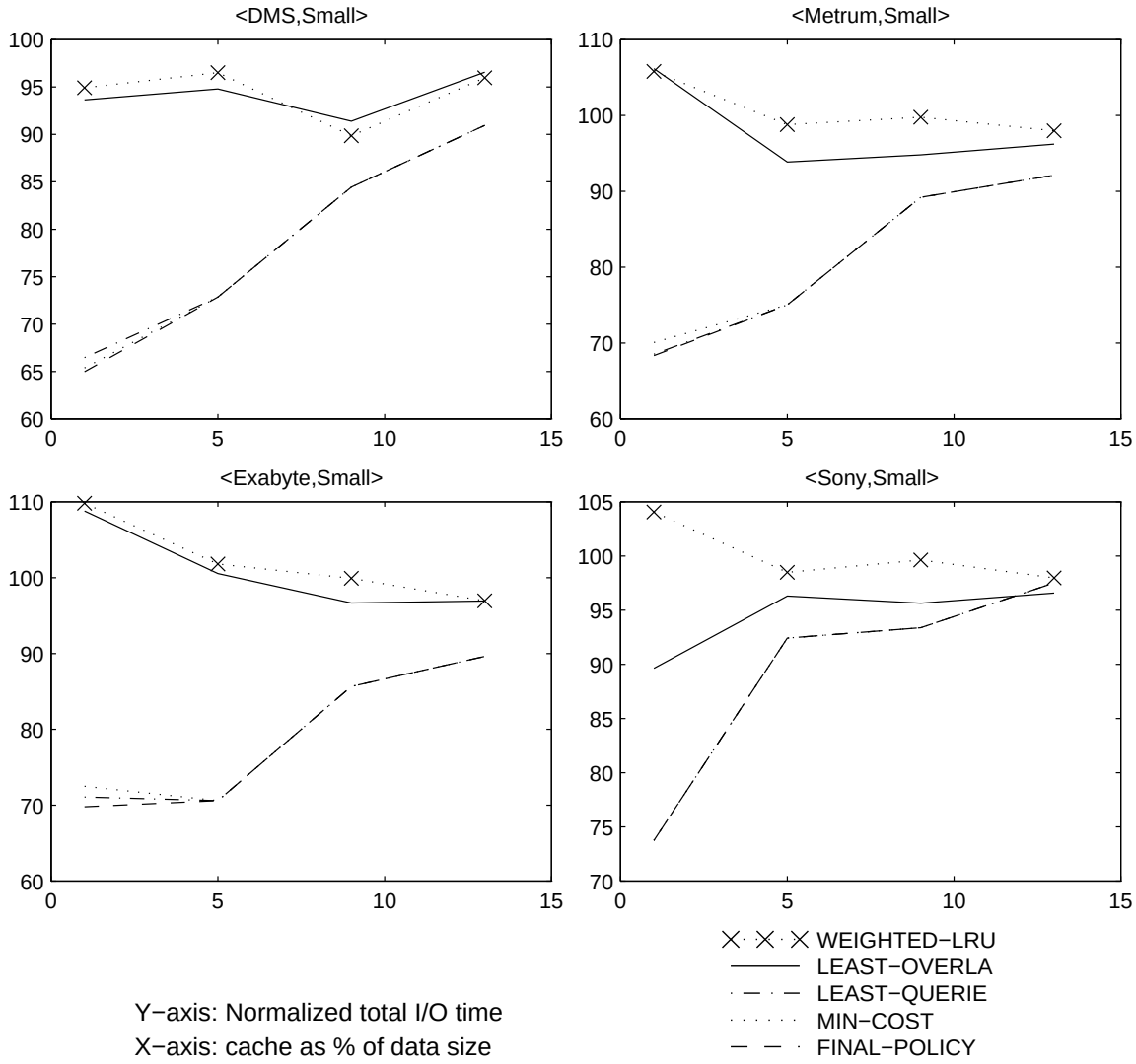


Figure 3.14: Performance of fragment eviction heuristics under varying cache sizes for SMALL-DATASET.

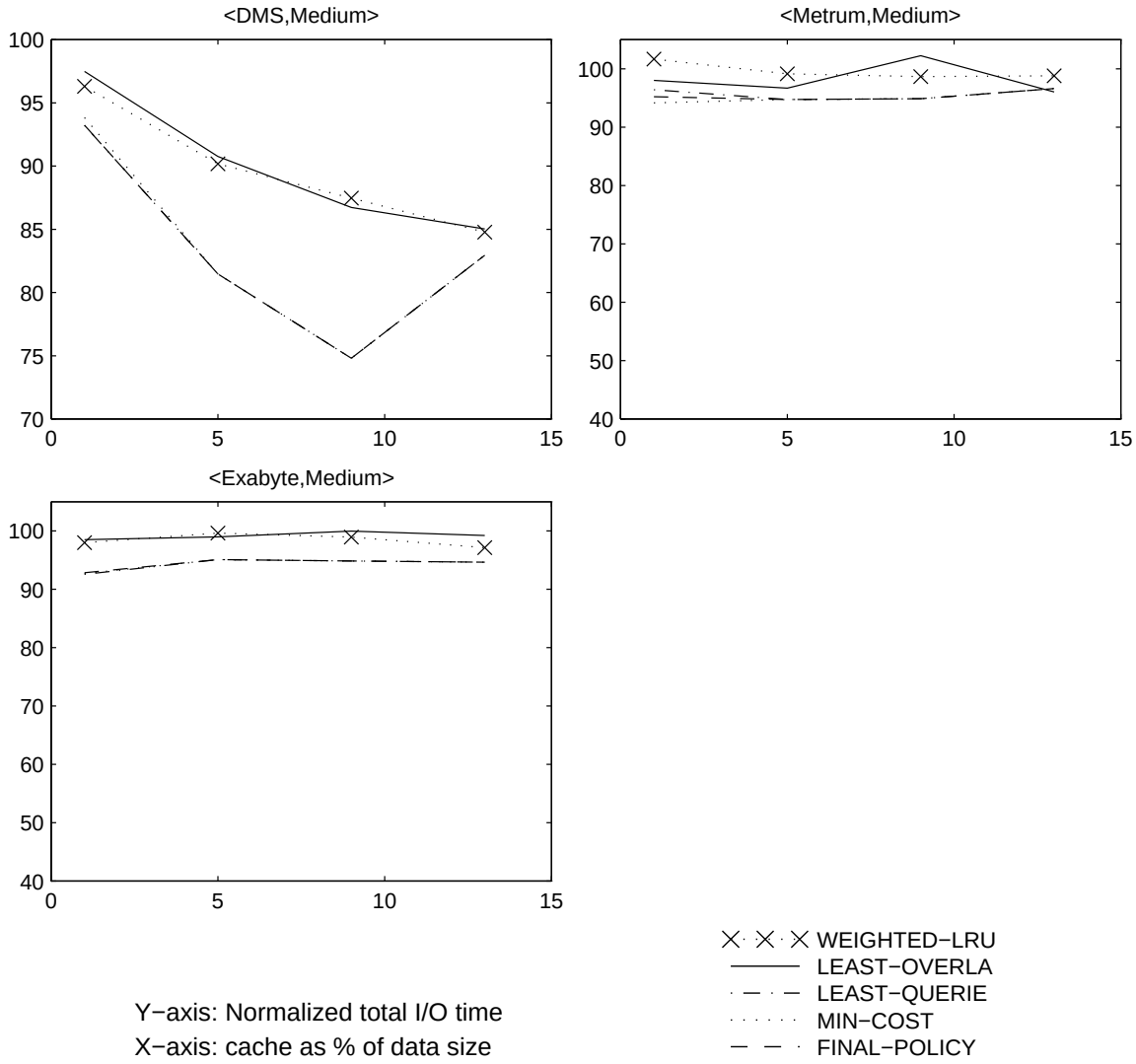


Figure 3.15: Performance of fragment eviction heuristics under varying cache sizes for MEDIUM-DATASET.

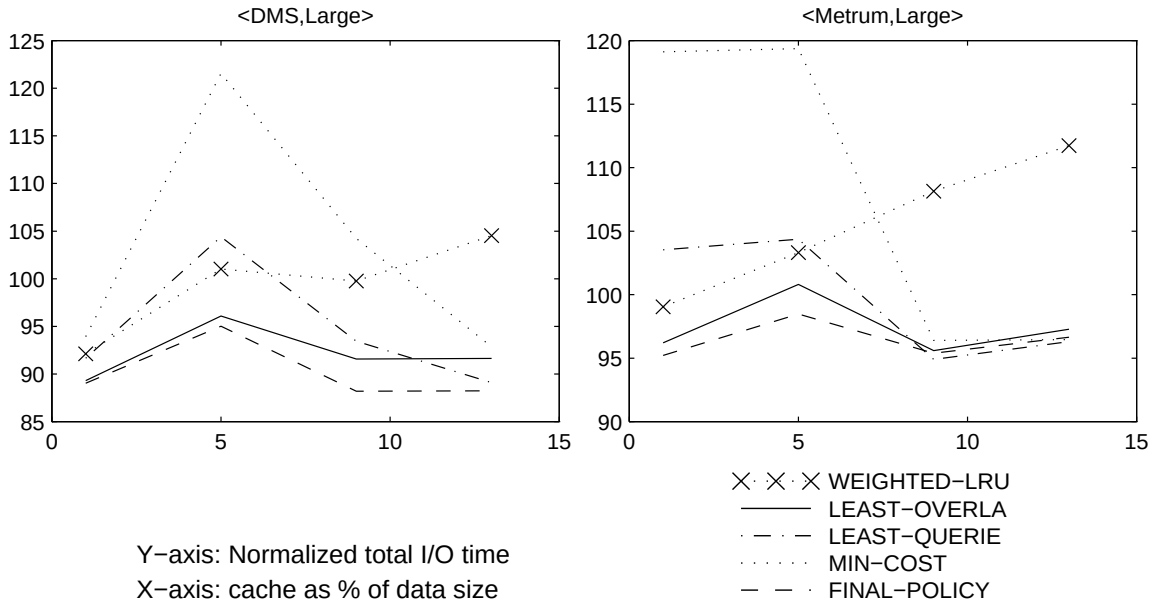


Figure 3.16: Performance of fragment eviction heuristics under varying cache sizes for LARGE-DATASET.

tertiary memory device. This form of parallel I/O scheduling also helps hide some of the latency of platter switch operation — when one drive is transferring data, the robot arm is free and can be employed for switching platters on some other drive.

Handling multi-way joins: Adding multi-way joins to the query workload means that now subqueries can require more than two fragments. The main idea in our fetch policy was balancing the two conflicting objectives of "fetching the fragment that has the lowest I/O latency of access (MIN-LATENCY policy)" and "fetching the fragment that joins with cached fragments and thus relieves the load on the cache (FREE-CACHE policy)". The MIN-LATENCY does not depend on the join relationship between fragments and thus can be applied "as-is" for multi-way joins. However, the FREE-CACHE policy has to be modified to account for queries that need more than two fragments to complete. In particular, we cannot fetch fragments simply based on the what fragments it joins with — some form of subquery-level decisions are required.

The modified FREE-CACHE policy is: First choose a subquery which will make best use of the cached fragments. That is, the remaining uncached fragments of the subquery will join with the largest sum of sizes of the cached fragments. Then, fetch each fragment of the subquery in the MIN-LATENCY order. Note that when there are only two way join

queries, this policy reduces to our original FREE-CACHE policy.

Fetching partial fragments: So far, we have assumed that data transfer always occurs in units of fragments. This could lead to redundant data transfers for high-selectivity index scans especially when the scan is unclustered. Hence, we modify the scheduler to also handle data transfer requests in the form of list of blocks of the fragment instead of whole fragment.

Alternative seek algorithms: For the fragment fetch policy, we assumed a linear model for computing seek costs. That is, the best order of fetching fragments on a tape is by sorting the fragments in their storage order. We can easily extend our algorithm to work for other cases where this assumption does not hold. For example, for DLT tapes the linear cost assumption does not hold and [HS96b] presents an alternative, better algorithm for reducing seek cost on such tapes. We can adapt such alternative algorithms for scheduling fragment fetch cost in our framework. We simply invoke the alternative algorithm to find the best order for fetching the fragments. Then, the the MIN-LATENCY policy would fetch the fragment in that order instead of the storage order.

Fairness: Our goal in the design of the scheduler was to maximize throughput. We believe that throughput is a more desirable metric for optimization than response time because the limitations of tertiary memory devices prevent interactive querying. Most of the queries are expected to be submitted in a batch mode rather than in an interactive manner. However, to prevent indefinite starvation, we have a mechanism whereby a query waiting long enough will be scheduled for execution irrespective of other throughput optimizations.

3.5 Summary

In this chapter we described the working of the scheduler and presented the policies it uses for fetching and evicting fragments from the disk cache. Our policies perform well under a wide range of tertiary memory characteristics, workload types, cache sizes and system load and adapts dynamically to changes in these parameters.

Chapter 4

Performance Evaluation

This chapter presents the implementation details of the proposed query processing engine. Then it presents a performance evaluation of the prototype using the Sequoia 2000 benchmark and several synthetic workloads.

4.1 Implementation

The query processing architecture described in the previous two chapters is implemented and operational on a DEC Alpha AXP workstation running Digital UNIX (OSF/1 V3.2). The current system was built from the original POSTGRES [SK91] database system that was extended by the Mariposa project [S+96] to provide a multi-threaded network communication package using Digital's ONC RPC and Posix threads. In addition, the following new modules were added:

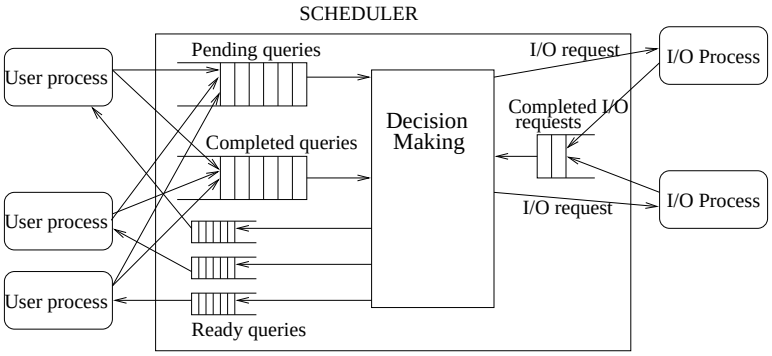


Figure 4.1: Design of the Scheduler.

The Scheduler: The scheduler communicates with the user and I/O processes using RPC calls. It maintains queues of events for query arrival, query completion and I/O transfer completion. The main scheduler waits on these queues until an event triggers some action. A query arrival causes the scheduler to queue the new subqueries and schedule new I/O transfers if possible or mark a subquery "ready" if the data it needs is cached. An I/O transfer is possible if one of the I/O processes is free, the cache has un-pinned space for the selected fragment and the drive on which the I/O is intended is free. In the event of a query completion, the reference count on the cached fragments is decreased. If the reference count becomes zero, the fragment is un-pinned and the pending queries are examined to select new fragments to transfer and schedule queries on fragments that are either being transfered or are already cached. In the event of the completion of an I/O request, the I/O process is marked free and a new fragment is selected for transfer, if available. All ready queries are collected in separate queues (one queue per user). The user-processes wait on this queue to collect the list of subqueries to be executed next. The scheduler maintains information about the state of the tertiary memory device and the cached fragments in local data structures. The scheduler is a total of 5800 lines of C code. It can currently schedule only subqueries containing a maximum of two fragments each.

I/O process: The I/O-processes act as slaves of the scheduler. They communicate with the scheduler process to support the following two RPC calls: (1) CacheFragment() and (2) UnCacheFragment(). The number of I/O processes that can concurrently execute the "CacheFragment" call is limited to the number of drives in the tertiary memory device.

Execution Engine: The user processes are the original POSTGRES backend processes whose execution engine have been modified to support the three new meta-operators: "combine", "schedule" and "resolve" and the additional plan-tree fragmentation and subquery extraction procedures. The total number of additional lines of code for these extensions was around 2000.

Catalogs: To support the concept of fragments and to maintain their layout information the system catalogs were extended with the following new tables:

```
CREATE TABLE pg_fragment     -- provides the fragments to relation mapping
(
     FragmentId     INTEGER
     BaseRelId      INTEGER
);
```

```
CREATE TABLE pg_layout       -- layout information for each fragment
(
     FragmentId     INTEGER
     PlatterId      INTEGER  -- platter on which fragment is stored
     BlockId        INTEGER  -- block number of fragment
     Offset         INTEGER  -- offset on the platter
);

CREATE TABLE pg_device
(
   Type          INTEGER  -- Tape or disk?
   SwitchTime    DECIMAL  -- average time to switch a storage medium
   TransferRate  DECIMAL  -- data transfer rate
   SeekRate      DECIMAL  -- search/rewind rate for tapes
   SeekStart     DECIMAL  -- time to start a search or rewind on tapes
   NumDrives     INTEGER  -- number of drives in the storage device
   PlatCapacity  INTEGER  -- capacity of a platter
   NumPlatters   INTEGER  -- number of platters
);
```

Storage Hierarchy: The original version of POSTGRES provided a storage manager switch [Ols92] for adding new levels of storage hierarchy. This enabled easy incorporation of many different tertiary storage devices and magnetic disk caches. Each relation is simply tagged with the storage manager on which it resides during the creation stage. All I/O requests pass through a switch which in turn passes the request to the appropriate storage manager based on the catalog information. The tertiary memory storage manager stages data to a magnetic disk cache before moving to main memory. The size of the disk cache is fixed in advance during query compilation. For efficiency, data movement from the disk cache to the tertiary memory occurs in units of 256 KB which is 32 normal disk pages.

Many of our experiments were run on real tertiary memory devices. However, to facilitate measurements on robots for which the actual device was unavailable, we implemented a tertiary memory device simulator. The simulated storage manager used a magnetic disk for data storage but serviced I/O requests with the same delay as would an actual tertiary device which received the same request sequence.

4.2 Experiments

In this section we present an experimental evaluation of our prototype. There are two main challenges to doing performance evaluation for tertiary memory databases.

1. **Dealing with largeness:** Tertiary memory devices will be deployed for handling multi-terabyte sized datasets. Running experiments on such massive datasets results in unmanageable blow-ups both in terms of time and space. Loading data can take days and running queries on them can take even longer. Running sub-optimal algorithms for comparison purposes is especially time-consuming. Furthermore, multi-user query processing on such datasets really tests the limits of both the software and hardware, exposing bugs that were undetected so far. Important among these are those related to memory leaks and integer limits. Bugs that corrupt the database are especially heinous.

   To partly get around the problem of largeness, we constructed experiments on scaled-down version of databases, cache sizes, number of users etc. We supplemented these experiments with simulations as described in Chapter 3 during the early design stages when it was necessary to run several experiments in quick succession.

2. **Finding representative workloads:** The use of tertiary memory devices for on-line (or "near-line") database query processing is a relatively recent proposition. Thus, it is not possible to get any real-life workload or widely accepted benchmark. In particular, benchmarks like TPC-A, TPC-B, TPC-C and Wisconsin are not suitable since they were designed primarily for secondary memory databases [Gra93b]. The TPC-A and TPC-B benchmark are meant for testing performance of small update intensive transactions. It is extremely unlikely that tertiary memory devices will be used for frequently updated databases due to the very limitation of the device. Although the TPC-C benchmark includes a few complex queries, it shares the update-intensive nature of the TPC-A and TPC-B benchmarks. The Wisconsin benchmark is more query-intensive than the TPC-A,B and C benchmark but it has a very small number of relations (only 3) and is designed for measuring single user performance whereas most of our scheduling, caching and batching optimizations apply during multi-user processing. Of the existing benchmarks, the two that came closest to our requirements were the SEQUOIA-2000 benchmark and the TPC-D benchmarks. We

	tape-stacker	Magneto-optical (MO)jukebox
switch time (sec)	30	14
transfer rate (MB/sec)	2	0.5
seek rate (MB/sec)	140	-
seek startup (sec)	1.3	0.3
number of drives(varied)	1	2
platter size (GB)	10	1.3 (both sides)
number of platters	10	32

Table 4.1: Tertiary memory parameters.

could not run the TPC-D benchmark because most of the queries in the benchmark involve multi-way joins and the current scheduler can only handle 2-way joins. We provide results of running our experiment against the national version of the SEQUOIA-2000 benchmark. Further, to be able to better tune various workload parameters, we also constructed a synthetic workload and took a series of measurements for various different characteristics of the synthetic dataset.

Both of the above workloads provide us only average case performance. More useful insights can often be obtained by running particular query instances where it is easy to analyze where and why one approach performs better than the other. Therefore, we start with a few anecdotal cases of simple scan queries (Section 4.2.2) and then run measurements on the mixed multi-user workloads (Section 4.2.3).

4.2.1 Options compared

Another important issue is what we compare our approach with. Research on query processing for tertiary memory devices is in its initial stages. Hence, at the time we ran our experiments, there were no competing approaches against which we could compare. The version of POSTGRES with which we started supported query processing on tertiary memory using the conventional block-at-a-time paradigm of most secondary memory databases. Therefore, one of our metrics of performance was how much better the new architecture performed compared to the original POSTGRES architecture. However, comparing simply with POSTGRES is not satisfactory since it does not prefetch I/O blocks — a technique that is employed in some secondary memory database systems. Without prefetching, I/O

operations are done in a block-at-a-time manner. In contrast, with prefetching a fixed number n of blocks is fetched asynchronously whenever request is made for a single I/O block. For sequential scans the n blocks are the next n blocks stored contiguously after the requested block. For index scans, prefetching is done by first scanning the entire index tree, collecting the list of blocks required, sorting the blocks in the storage order and fetching n blocks at a time from this list. This corresponds to the list-prefetch technique used in DB2 [O'N94]. We extended POSTGRES to do prefetching. To this end, we added a set of prefetch processes whose sole responsibility was to asynchronously transfer a given collection of n blocks from tertiary memory to the disk cache. The number of such prefetch processes is equal to number of drives on the tertiary memory device. When a request is made for a block, the prefetch process is notified to fetch the next n block asynchronously. The user-process continues with normal execution as soon as the first block is cached. When a prefetch process associated with a drive is busy, all user-processes requesting I/O on that drive wait until the prefetch process is available again.

Summarizing, these are the three variations of query processing we measured:

- NoPrefetch: With this scheme data is fetched in units of a storage block (256 KB) on demand and no prefetching whatsoever is used. This corresponds to the original POSTGRES architecture. Note that data movement from the disk cache to main memory occurs in units of 8 KB but when a miss occurs for an 8 KB page, an entire 256 KB block is fetched from tertiary memory.

- Prefetch: In this scheme, both sequential prefetch (for sequential scans) and list prefetch (for index scans) is used. The size of the prefetch unit was set to 32 storage blocks (8 MB), which is used in some database systems that use prefetching [O'N94].

- Reordered: which is our scheme of query processing as described in this thesis.

4.2.2 Simple scan tests

In this section we demonstrate some of the basic cases where query scheduling is effective. Our objective is to show how conventional query processing techniques, although acceptable for single user queries, perform badly when multiple users interact. In particular, as few as two concurrent users are sufficient to highlight the difference between the above three approaches.

For the experiments in this section we used synthetically generated relations. Each tuple of a relation consisted of ten integer fields that enable selection based on different selectivities (as in the Set Query Benchmark [Nei89]) and a text field that was used to pad each tuple to a total (internal) size of 300 bytes. We used a 512 MB local magnetic disk drive as a cache. We first did a set of experiments on a simulated tape stacker (performance characteristics in Table 4.1).

Sequential scans

We measured the performance of sequential scan queries on a 5 GB relation with one, two and five users. The relation was stored contiguously on a single tape. The fragment size was 100 MB, which is one-fifth of the disk cache size to enable five users to concurrently execute.

The first set of experiments were with a single user. In Figure 4.2(a) we show the total time taken to process the sequential scan with the three schemes: NoPrefetch, Prefetch and Reordered. We also show the part of the total time spent in data transfer, platter switch and seeks on tertiary memory. We note that the Prefetch and Reordered schemes are 20% better than NoPrefetch. This is mainly due to I/O-CPU overlap. The total I/O done is the same in all three schemes but NoPrefetch does not enable effective overlap between I/O and CPU.

We then let two users run the same scan query, the second user submitted the query after the first one had scanned just more than 512 MB of the relation. The total time in all our multi-user experiments is defined as the time between the submission of the first query and the time when the answer to the last query is returned. As shown in Figure 4.2(b), the total time with Reordered is one-fifteen of NoPrefetch and less then one-half of Prefetch. With Reordered, the second user started the scan from the remaining part of the relation instead of the beginning as in the other two schemes. Thus, both users synchronized their processing perfectly, so that they processed the same data blocks at the same time. In contrast, with Prefetch the second user had to re-fetch every data block since the cache can only hold 512 MB. After the first user had scanned 512 MB of the relation, the blocks were evicted from the cache in LRU order for making space for the new blocks fetched by the first user. Thus, the second user had to fetch the evicted blocks again. With Prefetch, we not only have to fetch more data than Reordered but the

time for each block transfer also increases since the I/O requests of the two users interfere on tape resulting in increased seek cost. For PREFETCH, almost 15% of the time is spent is spent in seeks whereas for REORDERED, the seek overhead is negligible. For NOPREFETCH the data blocks are fetched one-at-a-time. Therefore, the access requests of the two users interfere for almost all blocks fetched and the tape has to seek back and forth between the scan positions of the two queries much more frequently than for PREFETCH.

We next repeated the query with five users to measure how these results scale. Each user submitted its query after the first one had scanned somewhere between one-tenth to one-half of the entire relation (selected randomly). In this case, REORDERED takes almost one-fifth the time taken by PREFETCH. By synchronizing the scans of the different users, REORDERED not only makes better use of cached data, it also incurs smaller seek cost. For PREFETCH almost 80% of the total time is spent in seeks whereas for REORDERED the seek cost is negligible. We expect this trend to continue as we increase the number of users and stagger their scans such that simple LRU based cache replacement policies cannot ensure proper reuse of cached data. The separation between the scans of different users is critical in determining the difference between the various schemes. In particular, when all users submit the scans at the same time all three schemes have the same performance. When each user query is perfectly serialized, then REORDERED again performs better since the second user's scan will be reordered to scan the last part of the relation cached on the disk first instead of scanning from the beginning.

This experiment illustrates how our method of reordering execution can enable better caching performance than conventional prefetching schemes. The next experiment illustrates how we can use execution reordering to reduce I/O cost even when two queries are accessing disjoint data.

Index scans

In this experiment, we report the performance of unclustered index scans first with a single user and then with two users.

We used two 25 GB relations spread across 5 different tapes in units of 5 GB each. The first relation was stored on tapes 1 through 5 and the second on tapes 2 through 6. The fragment size was 256 MB. The indices reside on magnetic disk. The selectivity of the index scan was 0.01%. In Table 4.2 we show the performance of a single-user index scan.

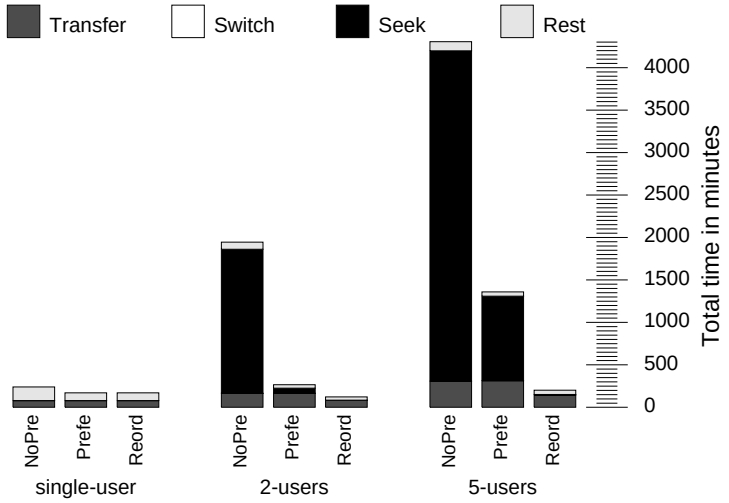


Figure 4.2: Difference in total execution time for three methods (NoPrefetch, Prefetch, Reordered) with sequential scans. "Rest" refers to the part of the total time not spent in tertiary memory I/O. The platter switch cost is negligible since data is on a single platter.

	Total (minutes)	Transfer (minutes)	Switch (minutes)	Seek (minutes)
Single-user				
NoPre	5619	19.4	4010	1527
Pref	297.3	17.5	2.5	276
Reord	297.3	17.5	2.5	276
Two-users				
NoPre	12351	38.9	8035	4215
Pref	1339	35	302.5	1000
Reord	586	35	3	548
5-users				
NoPre	30171	100	20090	9919
Pref	3144.5	87.5	600	2450
Reord	1467	87.5	6.5	1372

Table 4.2: Difference in total execution time with index scans.

NoPrefetch is almost two orders of magnitude worse than the other two schemes because it does too many random I/Os. Since the index scan is unclustered, each block access could result in an I/O request to any of the five tapes of the tertiary memory. This leads to high platter switch and seek overhead. Schemes Prefetch and Reordered convert the unclustered I/O to clustered I/O by pre-scanning the index tree, sorting the qualifying RIDs and fetching the data blocks in their storage order. This results in significant reduction in the number of platter switches and the the seek cost. Note that both Prefetch and Reordered incur the same I/O cost but the total time for Prefetch is slightly more. This is mainly because with Reordered the base relation is fragmented and each fragment has its own index tree whereas with Prefetch there is a single index tree for the entire relation. So, with Prefetch the RID of the entire relation needs to be sorted into a single list whereas with Reordered the RID list of each fragment is sorted separately, resulting in a smaller sorting cost.

Next, two users concurrently submitted the index scan query on the two relations. The first user's scan was on relation 1 that was spread on platters 1 to 5 whereas the second user's scan was on relation 2 that was spread on platter 2 to 6 as described earlier. For this case too, NoPrefetch was much worse than Prefetch and Reordered. In addition, Reordered performed almost factor of 2.5 better than Prefetch. Reordered does much fewer platter switches than Prefetch because the execution of user-1 is modified such that first both users finished processing on the data lying on tapes 2 though 5, then user-1 scans its part of the relation on tape 1, and finally user-2 scans its part of the relation on tape 6. Thus, the total number of platter switches is 6. In contrast, with Prefetch the scans of users 1 and 2 interfered. For instance, in the beginning when user-1 was fetching data from tape 1, user-2 was fetching data from tape 2. Although each user's scan was clustered (because of list prefetch), when the two users executed concurrently, for every prefetch request a tape switch was incurred. Even if we increase the size of the prefetch unit, Prefetch will incur at least four more media switches than Reordered.

We demonstrate how this result for two users scales over multiple users by running concurrently a collection of five index scans queries on five different relations of 25 GB each. Each relation was spread in units of 5 GB each across five different platters chosen randomly from 1 to 13. Each platter could hold a maximum of 10 GB. In this case too, the number of platter switches incurred is almost two orders of magnitude more with Prefetch than with Reordered.

This experiment demonstrates that statically reordering index scans reduces random I/O considerably for single user index scans. But, with multiple users static reordering is not sufficient for reducing random I/O. Summarizing, the sequential example showed how the amount of data transferred can be reduced by doing better scheduling of queries that share data accesses. The index scan example showed how the number of platter switches can be reduced by doing better scheduling of queries that share common platters. Thus, simply reordering execution based on static data storage order is not sufficient. When multiple users interact, dynamic execution reordering can yield significant gains even for simple workloads like sequential and index scans.

4.2.3 Multiuser-mixed workload tests

In this section we report performance of some multi-user mixed workloads of select and join queries. We first present the performance of a synthetically constructed workload and later present the performance of the SEQUOIA-2000 benchmark.

We took measurements under different configurations of cache sizes, number of drives and number of users to identify conditions where reordering pays off and where it does not. We report measurements on the simulated tape tertiary memory of the previous section and a real HP magneto-optical jukebox (performance characteristics summarized in Table 4.1) that is connected to our prototype[1].

Synthetic dataset

Table 4.3 summarizes the details of experimental setup for the synthetic dataset.

In Figure 4.3(a) we plot the total time for this workload on the tape-jukebox and the MO-jukebox with one drive each[2]. On the tape-jukebox, the total time with PREFETCH is about one-fifth of NOPREFETCH while REORDERED is one-seventh of PREFETCH. On the MO-jukebox, the total time with PREFETCH is about one-third of NOPREFETCH and REORDERED is about one-third of PREFETCH. For both NOPREFETCH and PREFETCH, the execution time is dominated by I/O on tertiary memory unlike in REORDERED. As shown in Figure 4.3(a), the main I/O bottleneck is platter switches for both NOPREFETCH and

[1]Magneto-optical jukeboxes offer substantially lower price-performance advantage over tape-jukeboxes, hence they are less popular in mass storage systems. We, therefore, prefer to do most of our experiments on tape jukeboxes.

[2]The one drive MO jukebox also had to be simulated since we only had a two-drive MO jukebox

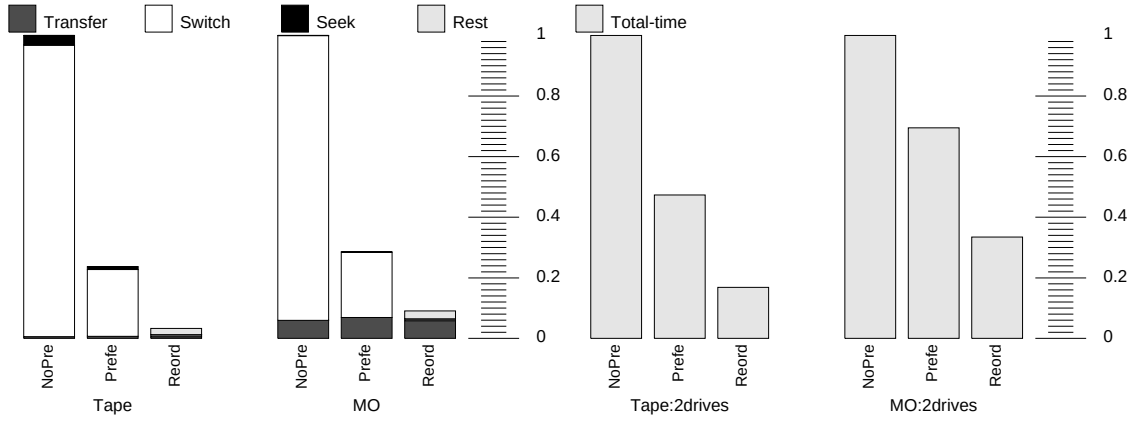
Description	Default
Workload	
Number of queries per user	5
Number of users	3
% of 2-way join queries	50
% of single relation queries	50
% index scans	80
Index selectivity	0.1-10%
Hardware parameters	
Storage device	DLT tape jukebox (simulated)
	Magneto optical jukebox (Table 4.1)
Cache size	512 MB
Database characteristics	
Tuple size	300 bytes
# of relations	10
Relation size (Uniform distribution)	100 MB to 10 GB
Data layout	each relation stored from 1 to 5 platters
Fragment size	$\leq$ 85 MB (one-sixth cache size)

Table 4.3: Experimental setup for experiments on the synthetic workload.

PREFETCH. REORDERED performs better since it greatly reduces the number of platter switches. For the MO-jukebox the platter switch cost is not as high as for the tape-jukebox. Therefore, we observe smaller relative gains with REORDERED for the MO-jukebox.

Increasing the number of drives: Since the main bottleneck is platter switches, increasing the number of drives from 1 to 2 decreases the difference between the reordering and non-reordering based schemes as shown in Figure 4.3(b). For the two-drive case we plot only the total execution time since it is difficult to separately account for the time spent in doing various I/O activities. For instance, data transfer on one drive might be overlapped with seeks on another drive. For REORDERED there was negligible change in execution time when we increased the number of drives from 1 to 2 since the total execution time was not bound by tertiary memory I/O.

In general, if we further increase the number of drives we can expect this trend to continue. At the stage where the number of drives is so large that all required platters are always loaded, the various schemes will differ only in the amount of data transfered and the seek overhead. We observed that in this case, REORDERED performed 25% better than



(a) Single Drive (b) Two Drives

Figure 4.3: Difference in total execution time for three methods (NoPrefetch, Prefetch, Reordered) using the mixed workload. The execution time is normalized by the time taken by scheme NoPrefetch to allow drawing on the same scale.

Prefetch for the tape-jukebox.

Decreasing working set: For all the experiments so far, the transfer cost incurred with all three schemes was not significantly different. One of the merits of our query scheduling policies is better reuse of the cached data. Therefore, we also expected to observe significant reduction in transfer time with Reordered. Closer inspection of the workload revealed that there was very little opportunity for reusing data since the degree of sharing between the three concurrent users was limited. Each of the three users picked at most two of the ten relations in the database with equal likelihood. Hence there was little chance of overlap between the component relations of queries running concurrently. To verify this claim, we repeated the 2-drive experiments, with five users instead of three and skewed the access requests so that 80% of the accesses go to 30% of the data. We observed that the transfer time for Reordered was almost one-half of that with Prefetch for the skewed dataset (Figure 4.4). Note that the amount of data transferred is slightly more for Prefetch because the prefetched data can often replace more useful data and thus adversely affect caching performance [CFKL95].

There experiments demonstrate that scheduling is beneficial for tertiary memory databases either when the platter switch or seek costs are high or when the degree of sharing

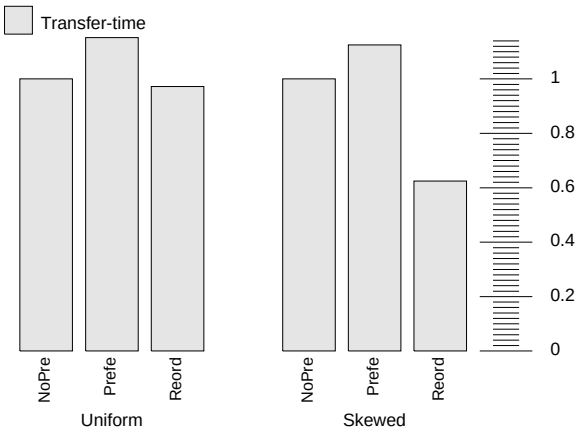


Figure 4.4: Difference in total transfer time for the three methods (NoPrefetch, Prefetch, Reordered) using the mixed workload on the tape jukebox. The time is normalized by the time taken by scheme NoPrefetch to allow drawing on the same scale.

Table name	# of tuples	tuple size	total size
RASTER	130	129 MB	16,744 MB
POINT	1,148,760	24 bytes	27.5 MB
POLYGON	1400, 000	204 bytes	286 MB
GRAPH	6500,000	175 bytes	1110 MB

Table 4.4: Sequoia Benchmark relations (national).

between queries is large.

Sequoia Benchmark

We ran the national version of the SEQUOIA-2000 benchmark which is of total size 18 GB. The database consists of four different kinds of relations: RASTER, POINT, POLYGON and GRAPH as summarized in Table 4.4. For the RASTER data, each tuple contains a 2-dimensional array of size 129 MB. The benchmark consists of 10 data retrieval queries which are either two-way joins and select queries on various relations. The last query involves a recursive ("*") operator on the GRAPH table which we could not run on POSTGRES since it does not support recursive queries. Since, the Sequoia benchmark does not have any information about the frequencies of posing individual queries, we let each user choose one of the 9 queries uniformly randomly. In Table 4.5 we summarize the details of the default

Description	Default
Number of queries per user	6
Number of users	5
Storage device	DLT tape jukebox (simulated)
Cache size	512 MB
Fragment size	$\leq$ 50 MB (one-tenth cache size)
Database Layout	
Indices	magnetic disk
RASTER base table	magnetic disk
2-dimensional arrays	spread over 10 platters; each platter has 13 arrays.
POINT	platters 1 and 2
POLYGON	platters 3,4,5 and 6.

Table 4.5: Experimental setup for experiments on the SEQUOIA-2000 benchmark.

experimental setup used for running this benchmark.

Figure 4.5 shows the difference in total execution time for the three schemes for varying number of users. In single user mode, REORDERED is 40% better than the NO-PREFETCH scheme used in the original POSTGRES architecture. This difference is mainly due to the reduction in platter switch and seek cost. Compared to PREFETCH, REORDERED is only 10% better in single user mode. However, as we increase the number of users REORDERED performs almost a factor of 2.5 times faster than PREFETCH. In general, the benefit of reordering is higher for larger numbers of users since there is more opportunity for batching and scheduling subqueries from multiple users. For eight users, REORDERED is almost a factor of 15 faster than NOPREFETCH. This difference arose out of a factor of 2.3 reduction in transfer time, a factor of 14.5 reduction in platter switches and a factor of 11.8 reduction in seek cost.

Figure 4.6 shows the effect of changing the cache size on the performance of various schemes. Notice that, as we increase the cache size from 256 MB to 512 MB, the total time taken with all three schemes decreases much more dramatically than increasing the cache size from 512 MB to 1024 MB. This is because of the special nature of the SEQUOIA-2000 benchmark. The majority of the data volume is due to the raster arrays (16.7 GB out of a total of 18 GB) whereas the rest of tables (POINT and POLYGON tables only, in our case) are a total of 314 MB in size. However, of the nine queries we ran, five queries access the raster arrays and six queries access POINT and POLYGON data. There is little locality on the

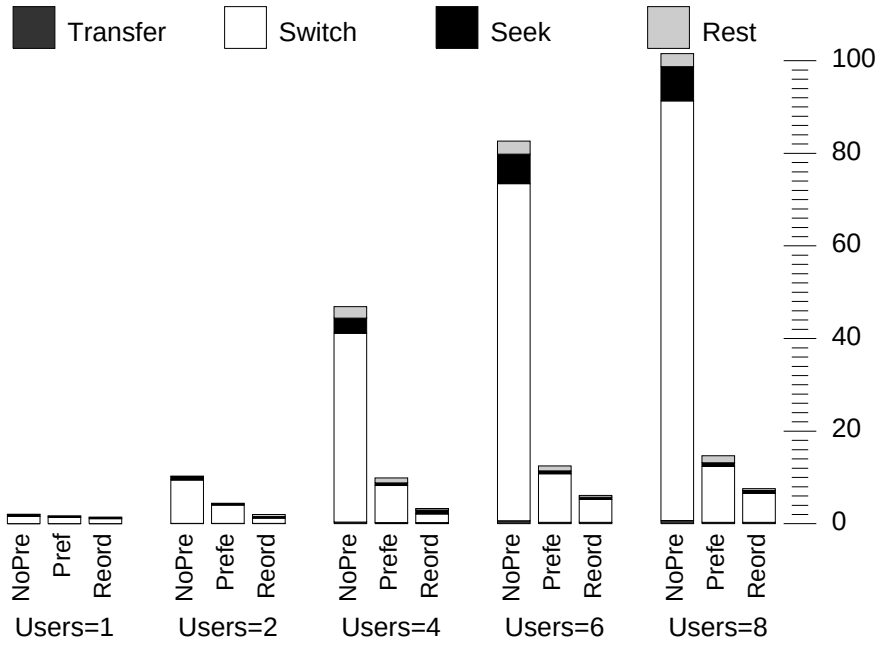


Figure 4.5: Results of running Sequoia benchmark with the three schemes for varying number of users.

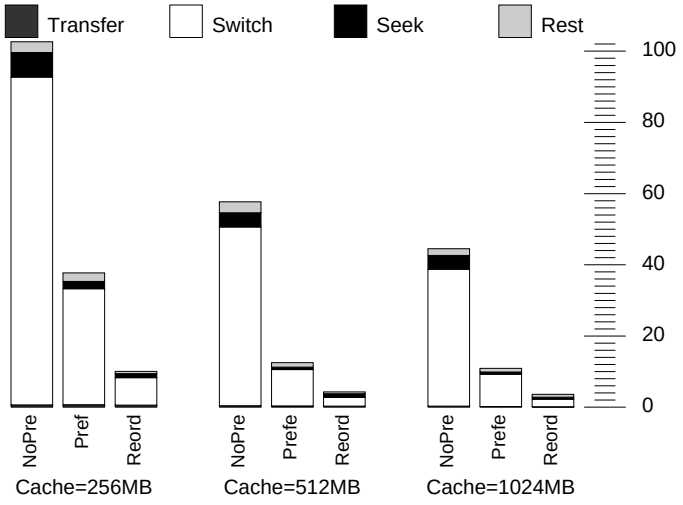


Figure 4.6: Results of running the SEQUOIA-2000 benchmark with the three schemes for different cache sizes.

raster data. Therefore, as soon as we can cache the 314 megabytes of POINT and POLYGON data we get most of the benefit of caching.

Another observation from Figure 4.6 is that, for the smaller value of cache size (256 MB) relative improvement with REORDERED is higher than for larger cache. This is because our algorithms can achieve better reuse of cached data since they base replacement decisions on pending queries whereas the LRU strategy used with NOPREFETCH and PREFETCH bases its decision on just time of last reference. When the cache is large enough to hold the hot set, LRU is good enough.

4.2.4 Scheduling overheads

Finally, we measured the overhead of scheduling in our prototype. For the experiments presented earlier, scheduling has definitely paid off, despite the overhead. But an important question is how well these benefits scale with increasing numbers of users or increasing numbers of fragments. The answer is crucially dependent on the scheduling overhead.

We measure the following overheads using the synthetic dataset described earlier in Table 4.3: (1) The per-fragment overhead that is directly proportional to the number of fragments in the query, e.g., the time to fragment a plan-tree. Measured as a percentage of the time to scan a cached fragment, this overhead was typically 0.06% (1.5 milliseconds). (2) the per-subquery overhead: e.g, the time spent in the extraction phase or in communicating with the scheduler. Measured as a fraction of the time spent in processing a two-way hash-join query on cached data, this overhead was typically 0.15% (5 milliseconds). (3) the per-session overhead e.g., time spent by the scheduler in deciding what subquery to schedule next. Unlike the previous two overheads this overhead depends on factors like the number of users concurrently active and the number of fragments per relation and can only be measured as a function of these factors. We plot this overhead as a function of number of users (1 through 9) and total number of fragments in the database (10 to 100) in Figure 4.7. The overhead per subquery increases only at a rate of 2 millisecond per additional user and less than 1/4th millisecond per additional fragment. The total overhead is thus measured to be typically less than 30 milliseconds per subquery and less than 1% of the total execution time.

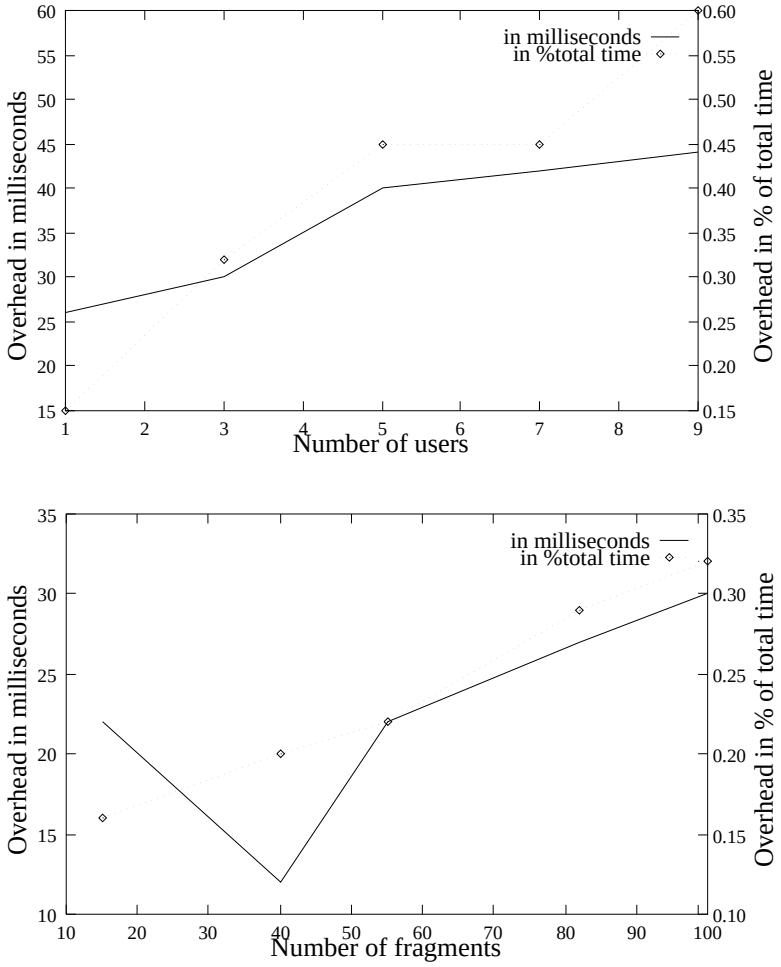


Figure 4.7: The per-session overhead as a function of the number of users and number of fragments. The Y-axes are overhead in milliseconds per subquery (top) and overhead as a percentage of the total execution time (bottom).

4.3 Summary

Our prototype yields almost an order of magnitude improvement over schemes that use prefetching and almost three orders of magnitude improvement over schemes that do not, even for simple index scan queries. Further experiments demonstrate that either (1) when the platter switch and seek costs are high, or (2) when the cache is small and there is overlap between data accesses of concurrent queries, our reordering scheme will enable better scheduling of I/O requests and more effective reuse of cached data than conventional schemes. The overhead of reordering is measured to be small compared to the total query execution time (less than 1%). Thus, at least for tertiary memory databases the penalty of reordering is so negligible that reordering can almost always be used to advantage.

Chapter 5

Array Organization

In the preceding three chapters, we discussed techniques for improving the performance of queries on tertiary memory using better scheduling, caching, prefetching and execution techniques. Another significant means for improving query performance is by doing better data clustering. The problem of clustering relations on one or more keys is well-studied for secondary memory databases and can be extended to tertiary memory databases. One problem that has not received adequate attention from the database community is the storage organization of large multidimensional arrays, even for magnetic disks. This chapter discusses techniques for organizing the storage of large multidimensional arrays for tertiary memory devices, but many of these techniques can also be applied to magnetic disks.

The rest of this chapter is organized as follows. Section 5.1 presents the different schemes we used for organizing arrays, namely chunking, reordering, redundancy and partitioning. Section 5.2 presents the simulation of several earth science arrays used by global change researchers [Sto91b] and shows the results of applying various array organization schemes to this data. Lastly, Section 5.3 presents future work and conclusions.

5.1 Storage of Arrays

We begin this section by presenting the access pattern model used for optimization of array layout. The feasibility and usefulness of data clustering algorithms is inherently linked to the expected access patterns on the data. Data is said to be well-clustered only when data items that are likely to be accessed together are stored together. Therefore,

before making any attempt at finding good storage methods for clustering arrays, we have to devise a model for representing accesses. A good access model should neither be too precise (since it is difficult to precisely specify future accesses), nor should it be too complicated (since it is difficult to find good storage methods for complicated patterns). Based on these considerations, we designed the following for representing array accesses.

Our model of access pattern is a set containing typical "shapes" of subarrays accessed from the array. We restrict each shape to be rectangular — therefore individual shapes differ only in the aspect ratio of the rectangle. In addition, each shape is associated with the probability p of accessing a rectangle of that shape. A more rigorous definition of access pattern is as follows. Consider an n dimensional array. Each user access request is an n-multidimensional rectangle (or hypercube) located somewhere within the array. User accesses are then grouped into collections of classes $L_1, \ldots L_K$ such that each L_i contains all rectangles of a specific size $[A_{i1}, \ldots A_{in}]$ located anywhere within the array. Each class i is then associated with a probability P_i of accessing a rectangle of that class. Therefore, the access pattern for an array can be described by the set:

$$\{(P_i, L_i)\text{such that} 1 \leq i \leq K, 0 \leq P_i \leq 1, L_i = [A_{i1}, A_{i2}, \ldots, A_{in}]\}$$

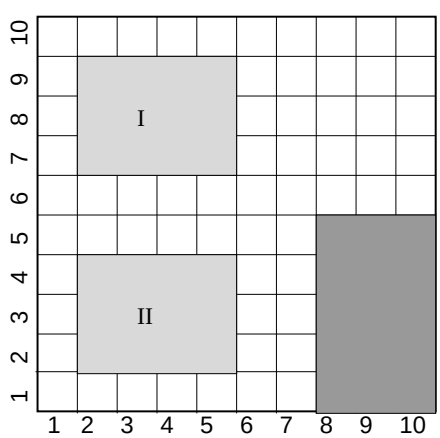


Figure 5.1: An Example Array

Figure 5.1 illustrates an example on a 10×10 array. The three shaded rectangles are each accessed with probability $\frac{1}{3}$ and represent accesses in two classes. Rectangles I and II belong to the first class since they have the same aspect ratio of [3,4] and rectangle III

belongs to a second class of aspect ratio [5,3]. The probability of accessing a rectangle in the first class is thus $\frac{2}{3}$ $(\frac{1}{3} + \frac{1}{3})$ and the probability of accessing a rectangle in the second class is $\frac{1}{3}$. This corresponds to the following access pattern:

$$\{(\tfrac{2}{3}, [3,4]), (\tfrac{1}{3}, [5,3])\}$$

The access pattern can either be provided by an end user at array organization time or can be determined by statistically sampling array accesses in a database management system.

5.1.1 Chunking

Instead of using FORTRAN style linear allocation, we can decompose the array into multidimensional chunks, each the size of one storage *block*. A block is the unit of transfer used by the file system for data movement to and from the storage device. The shape of the chunk is chosen to minimize the average number of block fetches for a given access pattern. To illustrate the significance of chunking we consider the example shown in Figure 5.2. Figure 5.2(a) shows a 3-dimensional array of size X_1=100, X_2=2000 and X_3=8000 stored using linear allocation and Figure 5.2(b) illustrates the same array stored using a chunked representation.

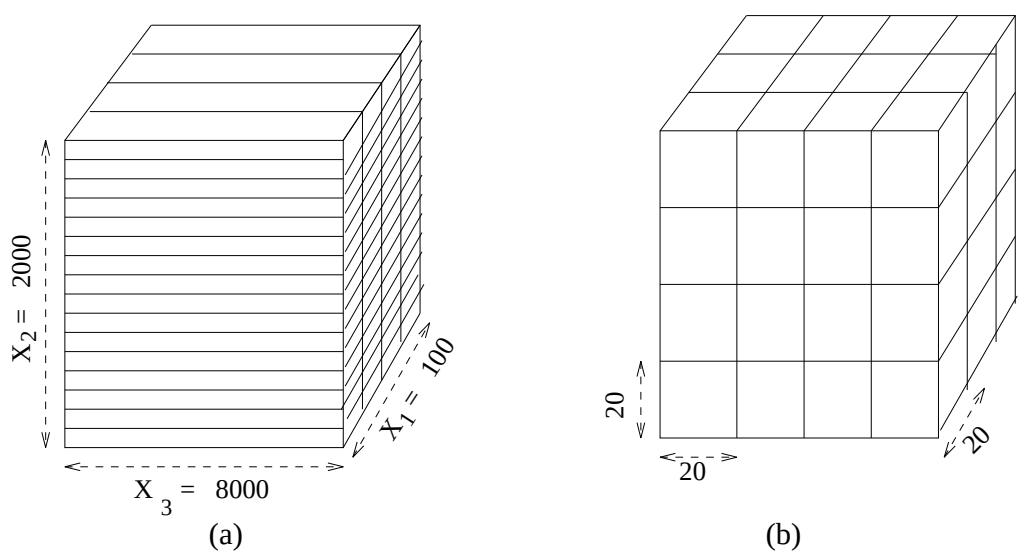


Figure 5.2: An example of array chunking

Assume the array is stored on a magnetic disk and data transfer between main

memory and disk occurs in 8000 byte pages. Let the access pattern for this array be

$$\{(0.5,\ [10,\ 400,\ 10]),\ (0.5,\ [20,\ 5,\ 400])\}.$$

The array is stored linearly with X_3 as the innermost axis followed by X_2 and then X_1, as shown in Figure 5.2(a). The innermost axis corresponds to the axis along which data is stored contiguously. For this method, each disk block will hold just one row of values along X_3. We will now estimate the average number of block fetches required for accessing a rectangle from the above access pattern. For these examples, assume for the sake of simplicity that the lowest point of each access request aligns exactly with the lowest point of the first block it touches. Thus, a request for a rectangle of shape [10, 400, 10] (first element of the access pattern) will span a total of $10 \times 400 = 4000$ blocks. Similarly a request for a rectangle of the second type [20, 5, 400] will span $20 \times 5 = 100$ blocks. Hence on average, this access pattern needs to fetch $0.5 \times 4000 + 0.5 \times 100 = 2050$ blocks per request. The average amount of data requested is 40,000 bytes which can fit in 5 blocks. Hence, the amount of data fetched is 410 times the amount of useful data.

Suppose we divide the array into 8000-byte chunks. The shape of each chunk is a (20, 20, 20) cube as shown in Figure 5.2(b). For the same access pattern, the number of blocks fetched is 20 for the first access and 20 for the second access, assuming that the start of the access rectangle aligns perfectly with the start of a chunk. The average number of blocks fetched is 20 as compared to 2050 for the unchunked array. Thus, chunking results in more than a factor of 100 reduction in the number of blocks fetched.

In order to realize these improvements, we need a way to optimize the shape of a chunk. Although, in this example all the sides of the chunk are the same, in general the sides of the optimal chunk can be of varying lengths based on the patterns of access on the array. Although, the idea of chunking arrays has been proposed earlier in other contexts [MC69, FP79], there has not been any work reported on finding good chunk shapes. This topic is discussed next.

We first present a formal definition of the problem. Given an n-dimensional array $[X_1, X_2 \ldots X_n]$ where X_i is the length of the i-th axis of the array, block size C and an access pattern $\{(P_i, [A_{i1}, A_{i2}, \ldots, A_{in}]) : 1 \leq i \leq K\}$, the objective is to find the shape of the chunk into which the array should be decomposed such that the average number of blocks fetched is minimized. The shape of the chunk is specified by a tuple $(c_1, c_2, \ldots, c_n)$ where c_i is the length of the ith axis of the multidimensional chunk. The size of the chunk

puts the following additional constraints on each c_i:

$$\prod_{i=1}^{n} c_i \leq C$$

The average number of blocks fetched for a specified access pattern and chunk shape is given by:

$$\sum_{i=1}^{K} \left(\prod_{j=1}^{n} \left\lceil \frac{A_{ij}}{c_j} \right\rceil \right) P_i \tag{5.1}$$

In the expression above, the quantity within the parenthesis ($\prod_{j=1}^{n} \lceil \frac{A_{ij}}{c_j} \rceil$) is the minimum number of blocks to be fetched for a query rectangle of the ith type in the access pattern. Thus, formula 5.1 is the number of blocks fetched averaged over all classes in the access pattern. The goal is to choose the chunk shape, satisfying the constraints, that minimizes the above expression.

The presence of the "ceiling" function in (5.1) makes a closed form solution difficult. One can always find the optimal solution by exhaustive search of all possible shapes that satisfy the size constraint. In this case, the number of shapes generated is exponential in the dimensionality of the array. Various techniques can be used to prune the search space. For example:

1. Instead of considering all possible shapes, we only generate the ones which are *maximal*. A shape is maximal when increasing the length of any one of the sides of the shapes will violate the size constraint. For example, if $C = 15$ and $n = 2$, then shape (5,3) is maximal whereas (4,3) and (5,2) are not. A shape that is not maximal cannot be the optimal solution because the storage block will contain fewer useful bits for any query than another storage block that contains the corresponding maximal shape.

2. The maximum length of a side of the chunk need not be more than the maximum length of the corresponding side over all classes in the access pattern. For example, for the access pattern {(0.5, 10, 400), (0.5, 20, 5)} on a 2-dimensional array, we need not consider shapes with the first side greater than 20 and the second side greater than 400.

3. Instead of considering all possible shapes, we first generate an approximate solution by only considering shapes for which the length of each side is a power of 2. This

solution is then refined by considering the shapes that are in the "neighborhood" of this shape. The neighborhood consists of sides varying between double and half of the corresponding side in the approximate solution. The optimality of the solution is not guaranteed with this pruning step. However for all cases we considered, the shape generated using this method was equal to the optimal solution that was found using a fully exhaustive search.

5.1.2 Reordering

Once the array is chunked, we require a good method of laying out the chunks on disk or tape. The natural way is to lay out the chunks by traversing the chunked array in the axis order. Hence, different axis orders will result in different chunk layouts. The time to fetch the blocks for a requested rectangle can be greatly reduced by choosing the right axis order. We now derive a simple formula for finding a good ordering of the array axes so that the average seek distance to retrieve a rectangle from the access pattern set is minimized. For this analysis, we assume that the blocks of the array are laid out contiguously on the platter and that the platter is used exclusively for retrievals on the array data.

To minimize seek cost, we minimize the average distance between the first and last block of a query rectangle in a 1-dimensional layout of the array. Consider an n dimensional array $[X_1, X_2 \ldots X_n]$ divided into chunks of shape $[c_1, c_2 \ldots c_n]$. We will first consider the seek cost when the array is stored in a one-dimensional storage medium.

Lemma 5.1.1 The number of blocks between the first and last byte of an access request $(y_1, y_2 \ldots y_n)$ is at least

$$(z_1 - 1)(d_2 d_3 \ldots d_n) + \ldots + (z_i - 1)(d_{i+1} \ldots d_{n-1} d_n) + \ldots + (z_{n-1} - 1)d_n + z_n \qquad (5.2)$$

where $z_i = \lceil y_i / c_i \rceil$, $d_i = X_i / c_i$ (assuming c_i divides X_i exactly).

PROOF. Transform all indices to a new coordinate system where chunk $[c_1, c_2, \ldots c_n]$ is the basis element. In the new coordinate system the array dimension is $[X_1/c_1, X_2/c_2, \ldots X_n/c_n]$ which is equal to $[d_1, d_2, \ldots d_n]$ and the access request is $(\lceil y_1/c_1 \rceil, \lceil y_2/c_2 \rceil, \ldots \lceil y_n/c_n \rceil)$ which is equal to $(z_1, z_2, \ldots z_n)$. We now have an array $[d_1, d_2, \ldots d_n]$ with an access request $(z_1, z_2, \ldots z_n)$ on it. If the array is laid out linearly in the axis order $1, 2, \ldots n$, with n as the innermost axis, the number of blocks between the start block and the end block of the access rectangle is given by formula (5.2). ∎

Lemma 5.1.2 Given an access pattern, the value of expression (5.2) averaged over all elements of the access pattern is minimized for the order $1, 2, \ldots n$ (with n as the innermost axis) if

$$\frac{a_1 - 1}{d_1 - 1} \leq \frac{a_2 - 1}{d_2 - 1} \leq \ldots \frac{a_n - 1}{d_n - 1}, \qquad d_i \neq 1$$

where $a_j = \sum_{i=1}^{K} A'_{ij} P_i$, $A'_{ij} = \lceil A_{ij}/c_j \rceil$ and $d_i = X_i/c_i$.

PROOF. Substituting a_i for z_i in (5.2) gives the expression to be minimized. Rewriting it with $a_i - 1$ replaced by x_i, $\forall i$ we get,

$$(((\ldots(x_1 d_2 + x_2)d_3 + \ldots)d_i + x_i)d_{i+1} + \ldots)d_j + x_j) \ldots + x_{n-1})d_n + x_n. \tag{5.3}$$

Interchanging positions of dimensions d_i and d_j $(i < j)$ gives,

$$((\ldots(x_1 d_2 + x_2)d_3 + \ldots)d_j + x_j)d_{i+1} + \ldots)d_i + x_i) \ldots + x_{n-1})d_n + x_n. \tag{5.4}$$

If (5.3) is minimal then (5.3) $\leq$ (5.4) which is true iff

$$((x_i d_{i+1} + x_{i+1}) \ldots)d_j + x_j \leq ((x_j d_{i+1} + x_{i+1}) \ldots)d_i + x_i \tag{5.5}$$

We next prove that (5.5) holds if,

$$\frac{x_i}{d_i - 1} \leq \frac{x_{i+1}}{d_{i+1} - 1} \leq \ldots \frac{x_j}{d_j - 1} \tag{5.6}$$

Let $P(i, j)$ denote the statement : *if* (5.6) holds then inequality (5.5) holds. The proof is done using induction over $k = j - i$. For $k = 1$, $P(i, i+1)$ is clearly true. Assume $P(i, j)$ is true for all i, j st $j - i \leq$ some k. For $j - 1 = i + k$, therefore,

$$\begin{aligned}
&((x_i d_{i+1} + x_{i+1}) \ldots)d_{j-1} + x_{j-1})d_j + x_j \\
&\leq (((x_{j-1} d_{i+1} + x_{i+1}) \ldots)d_i + x_i)d_j + x_j \\
&\leq (((x_{j-1} d_{i+1} + x_{i+1}) \ldots)d_j + x_j)d_i + x_i \quad (since\ (5) \Rightarrow \tfrac{x_i}{d_i - 1} \leq \tfrac{a_j}{d_j - 1}) \\
&\leq (((x_j d_{i+1} + x_{i+1}) \ldots)d_{j-1} + x_{j-1})d_i + x_i \quad (since\ P(i+1, j)\ is\ true)
\end{aligned}$$

Extending this for any pair (i, j) such that $i < j$ and substituting $a_i - 1$ for x_i completes the proof for Lemma (2). ■

To illustrate the advantage of re-ordering the array axes reconsider the example in Figure 5.2(b). Then for the access pattern assumed for Figure 5.2, the average distance

between the first and last byte of an access request (from Lemma 1) is 4020 for the array axis order (X_1, X_2, X_3). Using Lemma 2, if we reorder the axis as (X_1, X_3, X_2) the distance is reduced to 1020.

Even if the assumptions stated in the beginning of this subsection do not hold strictly and the storage medium is not one-dimensional, it is worthwhile to reorder the array axes. Intuitively, the ordering of Lemma (2) increases the sequentiality of array accesses and hence reduces seek time.

5.1.3 Redundancy

Data layout using one chunk size minimizes *average* access cost, meaning it is efficient for some rectangles but inefficient for others. We propose maintaining redundant copies of the array which are organized differently to optimize for the various classes in the access pattern. Specifically, we divide the classes in the access pattern into as many partitions as there are proposed copies and optimize each copy for its associated partition. Hence, the first step is to find R partitions, where R is the number of copies, such that the cumulative access time for the queries in the classes of the access pattern is minimized. We can do this using one of the following two approaches:

- Use brute force to try all possible partitions and choose the best. In the worst scenario, the number of partitions to be considered is exponential in the number of elements in the access pattern.
- Use vector clustering techniques [LBG80] to group classes into clusters. We have a starting set of K classes and wish to divide them into R clusters. Initially, each class belongs to a different cluster and we progressively merge pairs of clusters with the minimal weighted distance between them until R clusters remain. Algorithms for computing minimal distance are given in [Equ89].

When a read request arrives for a replicated array, the runtime system first finds the replica with the smallest estimated access cost. The estimated cost is a weighted sum of the number of block fetches, seek distance and media switches (in case of tertiary devices). The least cost replica is then used to answer the query.

5.1.4 Partitioning

So far we have presented optimizations for reducing the number of blocks and the seek cost. The third important cost component for tertiary memory devices is the platter switch cost and we now present a method of partitioning a large array across multiple platters to minimize the number of switches. Intuitively, the array should be partitioned such that the parts of the array accessed together frequently lie on the same media. We can extend the chunking methodology to deal with platter switches by:

- modeling the size of the chunk as platter instead of a disk block.
- minimizing the number of platter switches instead of number of page fetches.

Each partition is therefore a chunk of size equal to the amount of data stored in a platter and whose shape is found using the method discussed in Section 5.1.1.

Summary

In this section, we presented the different schemes to reduce the access cost on arrays. Access costs are comprised mainly of data transfer time, seek time and media switch time (for tertiary memory devices). We proposed a step by step procedure for optimizing the storage of an array as summarized in Figure 5.3. We first apply chunking to minimize the number of blocks fetched and hence the transfer time. The chunked array is reordered with the aim of reducing the seek time. For arrays larger than the platter size, we use partitioning to reduce the number of media switches. If R level redundancy is to be used, the access pattern is divided into R clusters and chunking, reordering and partitioning is applied to each cluster.

5.2 Performance

In this section we present the performance improvement provided by our organization techniques. Our experiments were done on a DECstation 5000/200 running Ultrix 4.2. Measurements were made on two different tertiary memory devices: (1) the Sony WORM optical jukebox [Son89b, Son89a] — the tertiary storage device supported by POSTGRES [Ols92] and (2) the Exabyte tape library — this device had to be simulated (as discussed

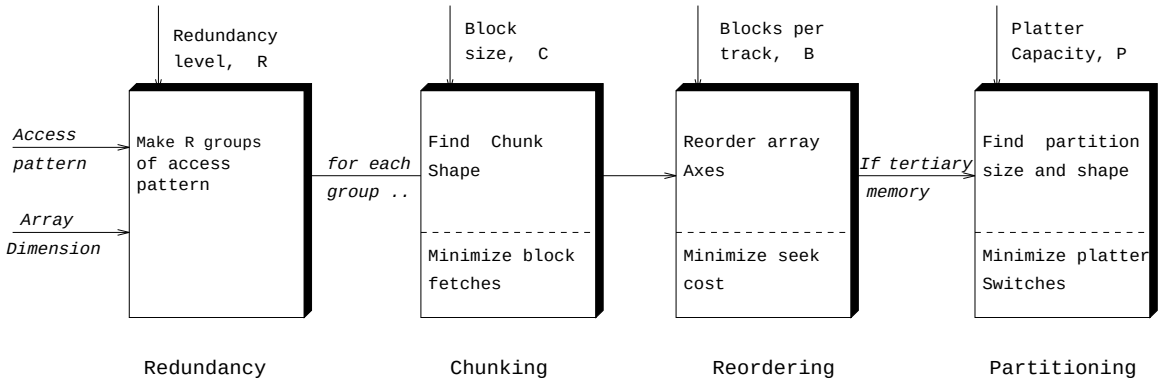


Figure 5.3: Array organization schemes

in Section 4.1 of Chapter 4) since the real device was unavailable. The performance characteristics of these devices appear in Table 3.1. The unit of transfer between the disk and tertiary memory was 256 KB and hence the block or chunk size was set to 256 KB.

Our measurements were on arrays actually used by global change scientists in the Sequoia project [Sto91b]. The first data source was atmospheric output from the General Circulation Model (GCM) experiments done at UCLA. In this model, the entire earth (180° latitude by 360° longitude) is divided into regular grids with resolution varying from 1.25° to 5° for 9 to 57 horizontal layers of the atmosphere. For each point in the three dimensional grid, a collection of 38 variables are recorded at regularly spaced time steps. Thus, the output is another five dimensional array of time, elevation, latitude, longitude and variables. The UCLA scientists currently store the array by a nested traversal of the array axes in the order time, latitude, variables, longitude and elevation with time as the least rapidly varying dimension.

The second source of data was ocean model output from the General Circulation Model (GCM) simulations done at UCLA [M+92, Wei]. The arrays consist of three-dimensional snapshots of the ocean (covering the world or a region of it) taken at regular intervals of time with horizontal grid resolution varying from $\frac{1}{3}^{\circ}$ to 1°. For each point in the three dimensional space (of latitude, longitude and depth) there are 5 model variables namely, temperature, salinity and three velocity components along the x, y and z direction in space. Hence the arrays have five dimensions: time, latitude, longitude, depth and the variables. The UCLA scientists currently store the array by a nested traversal of the array axes in the order time, latitude, longitude, depth and variables with time as the outermost

benchmark #	array size	dimension	element size	storage media
data set 1	4.255 GB	[072 090 038 144 30]	4 bytes	tertiary memory
data set 2	4.255 GB	[114 360 180 024 06]	4 bytes	tertiary memory

Table 5.1: Benchmarks

axis.

We selected two benchmark arrays from the two sources described above as summarized in Table 5.1. The third column indicates the number of values along each of the five array dimensions. Data set 2 is chosen from the ocean GCM and data set 1 from the atmosphere GCM.

For each of the data sets, we obtained a collection of queries (10 to 20 in number) composed after consulting UCLA scientists. Some sample queries include:

- making surface plots of some variables over some portion of the total surface
- finding the mean or variance of a variable over time or elevation
- making cross-section plots of some variable over some region.

The access pattern was derived from the sample queries that were run to evaluate various array organization schemes. To study the performance improvement with the array organization techniques we performed the following measurements for each of the four data sets:

We first determined a good chunk shape for the user-provided access pattern using the method discussed in Section 5.1.1. The time to find the chunk shape for all the four data sets took less than a minute. We organized the array into chunks and ran the benchmark queries on the chunked array. The total execution time and the number of blocks fetched for executing the queries were recorded. Next, we reorganized the chunked array using the axis order specified by Lemma 2 and repeated the measurements using the same query set. Finally, we made two copies of the array as described in Section 5.1.3 and measured performance by executing each query on the array copy that has the smaller estimated cost.

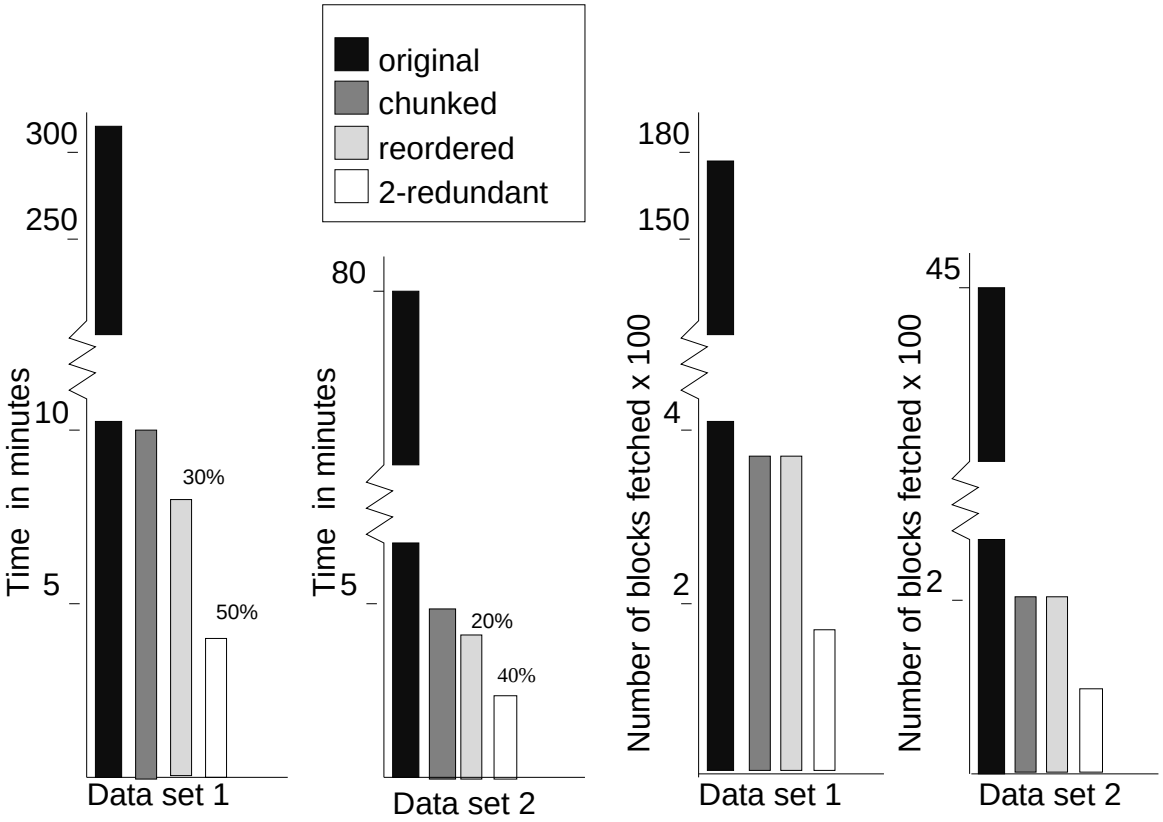


Figure 5.4: Performance measurements on the Sony WORM

5.2.1 Measurements on Sony WORM

For the Sony WORM, the total capacity of each platter is 3.27 GB, which is less than the total size of the array. We divided each array over two platters each containing approximately 2 GB. The Sony WORM has two drives. Thus, the two platters containing the array were always loaded into the reader during the course of running all queries on a particular array.

Figure 5.4 shows the results of applying various organization schemes on our data sets. Comparison of bars 1 and 2 for data set 1 shows that queries on the unorganized data take 5.2 hours to complete compared to 10.2 minutes on the chunked array. Similarly for data set 2 we observe a factor of 12 reduction in elapsed time. Reordering also works well and a 20% and 12% reduction in access times is achieved for data set 1 and 2 respectively. With 2-level redundancy the number of blocks fetched is lowered by another 60% and the access time by 50% as compared to the best single copy version for both data sets. Note that 2-redundancy, is not always twice better, the actual benefit depends on the query workload. For instance, when all queries are the same, 2-redundancy will provide no additional benefit.

5.2.2 Measurements on the tape jukebox

For the Exabyte tape jukebox, the entire array was stored on a single platter. In Figure 5.5 we show the total time and the seek cost for the four reorganization schemes. Note that the number of blocks transfered is the same as in the case of Sony WORM since we used the same access pattern and the same chunk size and shape. For this case, we notice almost a factor of 70 reduction in total time for dataset 1 and a factor of two reduction for dataset 2 when moving from the original layout to the chunked layout. The reason, we get lesser improvement with dataset 2 then with dataset 1 is because the original layout is good to start with for dataset 2. With reordering we observe a 5% reduction in total time with dataset 1 and a 25% reduction with dataset 2. Redundancy of level 2 reduces the total time further by another factor of two for both cases. An important difference between the results on the tape jukebox as compared with the Sony WORM is the percentage of time spent in seeks. In the right side of Figure 5.5 we show the seek time for each run. For most runs more than 80 to 90% of the total time is spent in seeks. The predominant seek cost also explains why we get lesser improvement with chunking for the Sony WORM than for the Exabyte tape library. Chunking, not only reduces the number of blocks transfered, it also

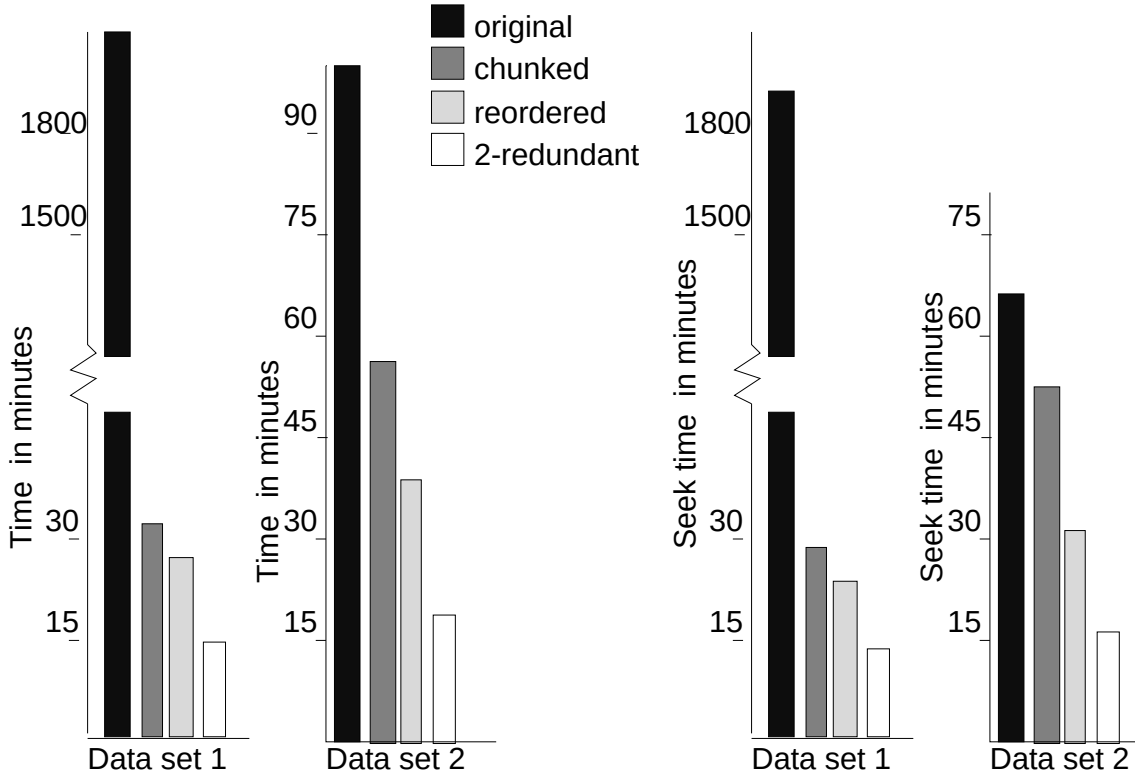


Figure 5.5: Performance measurements on Exabyte tape jukebox.

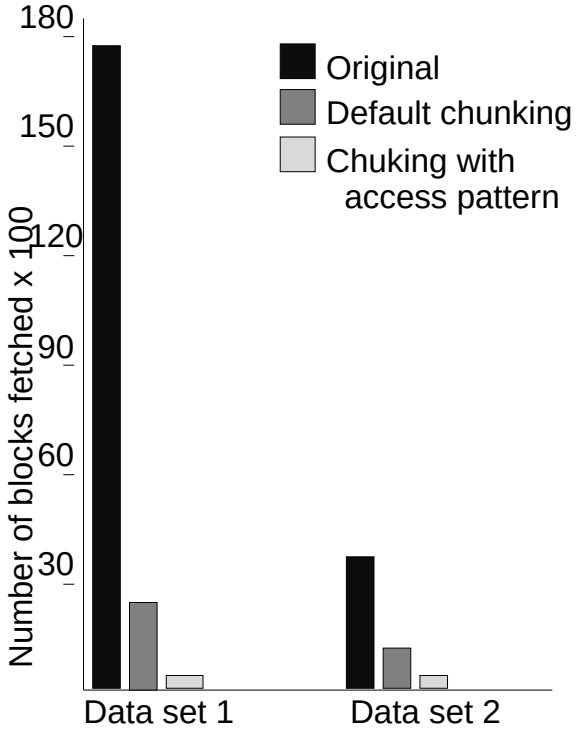


Figure 5.6: Performance of default chunking

reduces the seek distance between adjacent blocks. Reducing seek cost has higher payoffs on the tape jukebox than the Sony WORM. This explains the higher payoff on dataset 1. On the other hand, for dataset 2 the payoffs are smaller for tape jukebox than for Sony WORM because the array storage order is good to start with and therefore there is not significant reduction in the total seek cost. Reducing the number of blocks transfered does not reduce total cost as much for tape jukebox since the transfer cost is not dominant.

5.2.3 Effect of Access Pattern

In all of the optimization strategies discussed, the input access pattern has played a crucial role. To evaluate the role of the access pattern, we measured the performance on arrays that are chunked without regard to a particular access pattern. Instead, each array is organized using a default chunk, each side of which is chosen to be proportional to the side in the original array. Figure 5.6 shows the difference in total number of blocks accessed between an array chunked using a perfect access pattern and an array chunked using the default strategy. From the figure we notice that even with default chunking we can get

significant improvments but chunking with an access pattern can improve the performance further. For instance, for dataset 1, default chunking reduced the number of blocks fetched by one order of magnitude and using better access pattern reduces this further by almost another order of magnitude. Hence even when no knowledge of the access pattern is available linear layout is inferior to chunking for most real-world workloads.

5.3 Summary

In this chapter, we presented a number of strategies for optimizing layout of large multidimensional arrays on tertiary memory devices. Based on a suitably captured access pattern, we used chunking of arrays to reduce the number of blocks fetched, and reordering of array axes to reduce seek distance between accessed blocks. In cases where it was affordable, we suggested the use of redundancy to organize multiple copies of the same array based on different access patterns. Finally, we suggested partitioning as a method to reduce the media switch costs.

These optimization techniques were tested for their effectiveness in reducing the enormous access time on large arrays. Towards this end, we collected data from real users of large multidimensional arrays. Measurements based on their usage patterns showed significant reduction of access times with our optimization strategies.

Chapter 6

Related Work

This chapter places the thesis research in perspective with related research work. Section 6.1 discusses some of the early methods of deploying tertiary memory devices in mass storage systems. Then, Section 6.2 discusses tertiary memory database systems. Although there is limited work on query processing for tertiary memory databases per-se, a number of areas in secondary memory databases are relevant, particularly, query scheduling, centralized resource allocation, device scheduling, buffer management and multiple query optimization. Section 6.3 discusses these topics. Finally Section 6.4 covers related work in the area of data placement.

6.1 Mass storage systems

Although tertiary memory devices have only recently attracted the interest of database researchers, they have long been used in file-oriented mass storage systems like the National Center for Atmospheric Research (NCAR)'s MSS [N+87], Lawrence Livermore Laboratory's LSS [Hog90] and Los Alamos National Laboratory's CFS [C+82]. These are typically centralized supercomputing systems with multiple clients storing user files in huge tape libraries (e.g. Storage Tek Silos). Disk caches are used for staging data from the tape libraries in units of files. Files are brought from tape on user requests that are normally submitted long before the file is actually needed, to accommodate the huge latency of accessing the tape library. When space needs to be freed from the disk cache, techniques like LRU and weighted LRU [Smi81] are used to select files to be replaced.

6.2 Tertiary memory database systems

Tertiary memory devices are being increasingly deployed in a large number of new data intensive applications like digital libraries [W+], video on demand systems [FR94], image archiving systems [SB+93, OS95] and document repositories [FMW90]. We will concentrate our discussion on the use of tertiary memory devices with a DBMS.

As a result of the increasing demand for handling larger and larger datasets, the database community started realizing the need for handling tertiary memory devices and many researchers proposed doing so [SSU95, Sto91a, CHL93, Sel93, Moh93]. As a result some DBMSs started providing support for storing and accessing data on tertiary memory devices. Such DBMSs can be classified into two categories. The first and the more common category consists of DBMSs that do not exercise direct control of the tertiary memory device [Isa93, CK92]. Instead, they rely on the file system or a hierarchical storage manager (HSM) [CWCH93] to get transparent access to tertiary memory data, and store only metadata information in the database. The second category and the one of greater interest to us consist of DBMSs that directly control the tertiary memory device, e.g., POSTGRES [Ols92] and Digital's relational database product `DEC Rdb` [RFJ+93]. POSTGRES is a pioneer system in this regard. It included a Sony optical jukebox as an additional level of the storage hierarchy by 1992. The `DEC Rdb` supported storage of large multimedia objects on a write-once optical jukebox. Although these systems provide the enabling technology for supporting tertiary memory devices, they do very little tertiary memory specific performance optimization. More recently, a few research projects have started work on building a more "tertiary memory aware" system. We discuss them next.

The Paradise project (a database system for GIS applications) at Wisconsin [DKL+94] has recently integrated support for storing and accessing data on tertiary memory devices. Yu et al in [YD96] discuss the mechanism used by Paradise for storing satellite images on tape tertiary memory, and for optimizing accesses to these images. In their model, only satellite images reside on tape, and the rest of the data — including the base relations that refer the images — reside on magnetic disk. For optimizing accesses to these images, they first pre-execute each query to collect a batch of images and the list of blocks within each image that the query needs. Then, they optimize the retrieval of these blocks from tape by reordering the blocks of this batch along with those of other concurrent queries. However, unlike in our system, they cannot reorder the processing of tuples.

The Strata project [HSb] at Bell-labs is in the early stages of building a tertiary memory storage manager. In the context of the Strata project, [HS96a, HS96b] presents algorithms for scheduling small random I/O requests on a DLT tape device.

6.2.1 Single query execution on tertiary memory

[ML95] reports evaluation of various approaches for processing a single two-way join query where one relation is stored on magnetic disk and the second is streamed directly from a tape to main memory. This is different from our approach of staging data to the disk cache before processing any query on it. While their approach could lead to higher throughput when a *single* two-way join query is being processed, it is not clear how this approach scales when multiple users each executing multi-way joins are running concurrently. Since memory caches are typically much smaller than disk caches, the amount of asynchronous I/O and hence the degree of concurrency might be limited if we require relations to be streamed directly from tape to memory. Ideally, one should support a mix of the two approaches, choosing dynamically based on the number of user queries concurrently active, the number of drives and the sizes of the relations joined.

6.3 Related topics in secondary memory systems

6.3.1 Query scheduling

Parallel and distributed database systems Query scheduling is common in parallel [AC88, BSCD91, HS93, Y.W95, Gra90, CYW92] and distributed database systems [S+96, CP84]. Processing a plan-tree accessing multiple relations each of which could be horizontally fragmented across many different sites raises many interesting scheduling issues, and a variety of algorithms have been proposed. The details of the scheduling algorithms are not relevant to us since our goals are different to start with. Our goal is to maximize cached data reuse and to minimize platter and seek cost, whereas the goal in parallel and distributed DBMSs is to maximize load balance and minimize communication between processors. In spite of the difference in goal, there is similarity in the mechanism of scheduling plan-trees between these systems and our system. For parallel scheduling, plan trees have to be analyzed for meeting pipelining and ordering dependencies in a manner somewhat analogous to our subquery extraction step. However, our mechanism is different because,

for efficiency reasons discussed earlier (Chapter 2,Section 2.3.1), we execute all subqueries from a single plan-tree together. In contrast, most parallel and distributed systems use different plan-trees for subqueries to be scheduled on different processors. For example, suppose a nested loop join between relation, R consisting of 3 fragments and S consisting of 2 fragments is broken into 6 nested subqueries. In a parallel DBMS, typically, we would construct 6 different plan-trees, each of which are then executed independently. In our case, we construct only one plan-tree. If the scheduler makes more than one subquery ready together, say (R_1, S_1) and (R_1, S_2), we can execute them together since they are part of the same plan-tree. If we had two different plan-trees for (R_1, S_1) and (R_1, S_2), we would have to execute them one after the other, causing R_1 to be scanned twice — once for S_1 and once for S_2. Since the number of subqueries that the scheduler will mark "ready" is not known in advance, we cannot break a plan-tree into smaller parts that will be executed as a unit. Our execution engine gives the flexibility to decide dynamically what fragments pairs will be joined simultaneously.

Page join graphs: In centralized database systems, intra-query scheduling has been proposed to reduce the I/O cost when processing two-way nest-loop joins queries. Merrett et. al. in [MKY81] address the problem of minimizing the number of pages fetched from disk to a limited main memory when executing a two-way join query. They formulate the two-way join query as a graph (called the page join graph) where the nodes are the pages of the two relations and an edge between two nodes indicates that there is a matching tuple between the two pages. This problem is a special case of our formulation for fetching and evicting fragments from tertiary memory to disk cache as discussed in detail in Section 3.2 of Chapter 3. Similar page join graphs are also discussed in [MR93] and [LC93] for scheduling a two-way join query with the goal of maximizing resource utilization in a parallel database system. However, all these scheduling techniques apply for the special case of a two-way join query whereas we consider scheduling of more general query plans in a multi-user environment. Also, scheduling on tertiary memory devices is more difficult since we are also trying to optimize for platter switches and seeks and not just transfers.

6.3.2 Query optimization

Our technique is reminiscent of the way multiple query optimizers combine queries with common subexpressions [Hal76, GM81, RC88, RC88, SP89, AR92]. The scheduler in

our framework optimizes processing of multiple queries by scheduling together the execution of subqueries that refer the same data item. Most multiple query optimizers also attempt to combine together the execution of queries that access the same relation, but the key difference is that they combine queries at a whole relation or subexpression level whereas we combine queries at a finer granularity. This enables us to do several optimizations that are not possible with conventional multiple query optimizers. For instance, we can combine a new scan query on a relation with another query that is part way through scanning the relation. A limitation of our approach is that we do not further combine queries to be executed together based on common subexpressions. The scheduler represents each subquery simply as the list of fragments it needs and does not have any information about the predicates or expressions on the subquery. Thus, it can combine subqueries only at the level of fragment accesses from tertiary memory. Further, modifying the execution architecture to combine queries of separate user processes would require extensive changes to our process architecture. This is an interesting topic for future work.

Dynamic query optimization [Ant93, CG94] is relevant to our work since it also involves plan tree modification at runtime. However, in contrast to our method of plan-tree modification for dynamically reordering subqueries, the emphasis in dynamic optimization is on choosing dynamically from some fixed set of methods for performing an operation in the plan-tree.

6.3.3 Device scheduling

The scheduler, in our framework, integrates the tasks of device scheduling and cache management. Hence, many device scheduling algorithms are relevant in our context. One class of algorithms [BK79, Wie87, Hof83, SLM93] deals with reordering a set of pending I/O requests to reduce seek and rotational latency on magnetic disks. There is also work on scheduling a collection of I/O requests on tape so as to reduce seek cost. For instance, [KMP90] addresses the problem of finding the optimum execution order of queries on a file stored on a tape assuming a linear seek cost model, and [HS96b] studies the scheduling problem on a DLT tape assuming a non-linear seek cost. Scheduling in robotic devices is more challenging because tertiary memory devices have three cost components, all of which are significant in some situation.

6.3.4 Buffer management

Previous work on buffer management falls into three categories based on the kind of information used for making decisions:

1. those based on tuning to an expected reference pattern (e.g. LRU and the more adaptive variations like LRU/k [OOW93] and 2Q [JS94]),

2. those based on exact date needs of each query as inferred by the optimizer. (e.g., DBMIN [CD85], Hot set model [SS86], MG-x-y (buffer allocation based on marginal gains) [RN91, YC91])

3. those based on hints from the optimizer instead of exact and complete data reference pattern (e.g.,[HC+90, JCL90]).

Our approach falls in the second category, but is different from previous algorithms in that category for three reasons:

1. we modify execution order based on contents of buffers,

2. we integrate data replacement decisions with the state of the I/O device (e.g., by replacing fragments that are on currently loaded platters in preference to fragments on unloaded platters as suggested in [Yu95]) and

3. we make buffer management decisions on the basis of parts of a query instead of whole queries.

6.3.5 Prefetching

Prefetching is useful both in operating systems [CFKL95, Kot94, PGGo95] and database systems [TG84, CKSV93, AFZ96, GK94] especially when accompanied by execution reordering, e.g., list prefetch [MHWC90, Ant93, CHH+91] used with index scans. Our system extends prefetching to entire plan trees and not simply to index scans. A significant difference is that we can reorder based on dynamic conditions like cached data, the state of the I/O device, and the data needs of other queries whereas existing prefetching techniques reorder execution based on static storage layout.

Prefetching in OODBs: In object oriented databases, the navigational nature of queries can lead to bad I/O performance making it important to do prefetching [GK94] and batching [KGM91]. Keller et al. in [KGM91] present ways of modifying the plan-tree to replace object-at-a-time references with an assembly operator that collects multiple object references first and then reorders them to optimize I/O accesses. However the main difference between their scheme and ours (apart from the obvious difference in the architecture — relational vs object oriented) is that they cannot handle reordering across different operators of a plan-tree or across data references of different users. They can only reorder I/O accesses within a single operator and thus cannot handle interaction between the I/O requests of different operators.

6.4 Array Organization

The idea of array chunking is not new. McKellar and Coffman [MC69] explored the benefits of chunking in improving the performance of various matrix algorithms in a paging environment. They concluded that the use of sub-arrays can improve paging performance by orders of magnitude. Similar work on the use of chunking to make matrix computations efficient is discussed in [FP79]. Blinn [Bli90] reports a 10-fold reduction in the number of page fetches with chunking in a graphics rendering application. Chunking in the context of image processing has been used to build tiled virtual memory systems [Wad84] [RCM80] [Fra92]. Whereas all these systems deal only with two dimensional arrays and assume magnetic disk as the storage device, our interest is in arrays of higher dimensions with tertiary memory as storage devices. A more theoretical approach to organizing multidimensional arrays is presented in [Ros75]. Jagadish in [Jag90] discusses issue of organizing multidimensional arrays by reducing it to the general problem of mapping a multidimensional space on to a one dimensional space. All these approaches organize data without regard to any user specified access pattern, whereas our work considers access patterns to optimize layout.

Array organization is related to the general problem of data clustering. Most clustering algorithms [JD88] work on a collection of records that are not structured in any way. Arrays have a regular structure that facilitates a different approach to storage organization. This is also the reason why indexing structures like grid files [NHS84] or KDB trees used for indexing multi-attribute data are not relevant to array organization.

Subsequent to our work in 1992-93, a few mass storage systems have incorporated

similar techniques and enhanced them in various ways. Notable examples are: [DHL+93], [Bau93] and [SW94]. [SW94] deals with layout of arrays on disks of a multiprocessor and [DHL+93] discusses use of mass storage file systems to efficiently store and retrieve arrays on tertiary memory.

Chapter 7

Conclusion

7.1 Summary

In this thesis we discussed query processing and data placement techniques for optimizing retrieval of data stored on tertiary memory devices. Our basic goal in designing these techniques was to reduce and reorder I/O on tertiary memory.

Towards this end, we extended conventional query processing architecture with a centralized scheduler and modified the execution engine to allow arbitrary reordering of queries. The scheduler keeps system-wide information about the state of the tertiary storage device, the size and the contents of the disk cache and the data requirements of the pending queries. It uses this global information for making query scheduling, I/O scheduling and cache management decisions with the aim of maximizing over all system throughput.

We designed an execution engine that can work in cooperation with the scheduler to process queries in the order in which data arrives rather than demand data in a fixed order as most conventional systems do. For building a reorderable execution engine, we extended the plan tree data structure with three new meta-nodes that are added in an extra phase between optimization and execution of the plan tree. These operators enable the executor to communicate and synchronize with the scheduler for ordering the execution of subqueries. Our changes are restricted only to these new operators and the extra phase and thus enable modular extension of existing execution engines.

We then designed policies for deciding on the order of fetching and evicting data from tertiary memory. The policies are a function of the device parameters, the querying pattern of the application, the current load on the system and the size of the disk cache.

By identifying a few crucial device and workload parameters that are used to drive our optimization process, we make our system responsive to changes in these parameters.

We extended the POSTGRES execution engine and used it for building a prototype of a tertiary memory database. Our prototype yields almost an order of magnitude improvement over schemes that use prefetching and almost three orders of magnitude improvement over schemes that do not, even for simple index scan queries. Further experiments demonstrate that either (1) when the platter switch and seek costs are high, or (2) when the cache is small and there is overlap between data accesses of concurrent queries, our scheme will enable better scheduling of I/O requests and more effective reuse of cached data than conventional schemes.

Another approach to optimizing query processing on tertiary memory databases is through better data placement. We explored techniques for optimizing the storage of large multidimensional arrays. We presented a number of strategies for optimizing layout of large multidimensional arrays on tertiary memory devices. Based on a suitably captured access pattern, we used chunking of arrays to reduce the number of blocks fetched and reordering of array axes to reduce seek distance between accessed blocks. In cases where it is affordable, we suggested the use of redundancy to organize multiple copies of the same array based on different access patterns. Finally, we suggested partitioning as a method to reduce the media switch costs. Our measurements on several real-life datasets showed almost an order of magnitude reduction in access times.

7.2 Contribution

In the area of query processing, our main contribution is integrating device scheduling, cache management and query execution functionalities. Normally, these functions are handled independently but we show that by integrating these decisions in a centralized unit better optimizations are possible. For instance, our device scheduling decisions can be influenced by the contents of the cache — if a fragment joins with one of the cached fragments it will be fetched in preference to another fragment that does not. Similarly, our cache replacement decision can be influenced by the device state — a fragment on a loaded platter could be replaced in preference to those on unloaded platters. This is a departure from conventional cache management algorithms like LRU or WEIGHTED-LRU. The order in which different parts of a query plan-tree execute can also be influenced by the contents of

the cache and the state of the storage device. In contrast, in conventional query processing engines, once a plan is optimized, it is processed in a fixed order.

Our method of query scheduling and execution reordering can be packaged as a general framework applicable to other cases where the data access latency in various parts of the plan-tree varies widely and dynamically. The key features of our framework for reordering execution are:

1. Relations are comprised of **fragments** that are available together. This corresponds to part of a relation stored contiguously for tertiary memory databases.

2. Each query plan tree is divided into parts (called **subqueries**) that can be executed independently in arbitrary order.

3. A **scheduling unit** collects subqueries from many users and decides at runtime the order in which they are executed.

4. A **reorderable executor** communicates with the scheduler to process the query plan in the order dictated by the scheduler.

We can apply this framework in a number of alternative scenarios as follows:

- *Cache systems:* In all caching environments, the cached data is "nearer" than the uncached data and it might help to use our method of reordering to process the cached data before fetching more data that might replace it. Database cache replacement algorithms [CD85, Gra93a, TG84] are extensively researched but none of these algorithms have considered applying execution reordering to adapt to the cached data. Using this framework, we can do optimizations like scan a relation from the middle instead of the beginning if a prior scan query on the relation is already mid-way through. Such optimizations might be specifically useful in data warehousing environments where users scan a few large relations and the chances of finding multiple user-queries on the same relation is high.

- *Broadcast disks for mobile computing:* Broadcast disks [AAFZ95, IVB94, HGLW87] are gaining importance in mobile and asymmetric environments where the data bandwidth in the server to the client direction is much higher than in the reverse direction. With broadcast disks, data is periodically transmitted by base stations or servers to multiple clients instead of clients explicitly requesting data from the servers. This

reduces the number of messages from the clients to the data servers. In such an environment, it will help to reorder execution of queries at the client size so that they process the plan tree in the order in which data is broadcast by the server instead of following a fixed order of processing. For applying our framework to this environment, first one has to identify collections of data pages that are broadcast together. The scheduling unit would be responsible for watching the broadcast data stream, caching relevant data when appropriate and scheduling ready subqueries for execution.

In all these cases, the benefits have to be compared against the overhead of scheduling to evaluate overall payoffs. For instance, scheduling on fragments as small as a disk page is probably not a viable option since the measured overhead of scheduling per subquery in our prototype is around 25 milliseconds which is more than the time required to fetch a page from magnetic disk. For scheduling to pay off it is important to divide a relation into larger fragments, but too large a fragment can limit the degree of concurrency between queries. Also, the difference between the best case and worst-case time for accessing data must be high, and/or there should be significant overlap between data needed by multiple queries so that they can be batched together.

In the area of data placement, our main contribution was employing user-defined access patterns to optimize storage of multidimensional arrays. Normally, large arrays in database systems are stored as binary large objects without any inherent understanding of their structure. The ideas of chunking and reordering array axes have been in existence for a long time; our contribution was to provide means for choosing the shape of the chunks and to determine the optimal ordering of the axes based on expected access patterns.

7.3 Future Work

7.3.1 Query Optimization

The subquery-based execution paradigm introduced in this thesis raises many query optimization issues. Our method of executing queries in parts invalidates some of the assumptions and cost functions used by the optimizer. One such case arose when using index scans to get data in sorted order. When each fragment has its own index tree, an extra merge step is necessary to scan the whole relation in sorted fashion; we handled this case by explicitly taking into account the merge cost when index scans are used for getting data

in sorted fashion. For the case of nest-loop joins, the number of times each outer fragment is scanned cannot be known during optimization, and hence it is not possible to correctly estimate the cost of nest-loop joins. Other cases where optimizer changes are necessary are (1) estimating the access cost when some relations are stored permanently on disk and others on tertiary memory; (2) including the size of the disk cache in optimizing queries. When the disk cache is smaller than the relation, sorting may not be attractive since there is not enough space to hold intermediate runs.

Our approach of dynamically determining what part of a query gets executed together would benefit from dynamic query optimization [CG94]. For instance, the optimizer might have chosen a nest-loop query because of memory limitation but when a smaller fragment of the relation is scheduled, it might be more profitable to execute the join using hash-join instead of nest-loop join. Also, it might be useful to consider plans where the index scan is used only during query processing, but the relation itself is fetched sequentially from tape.

Another area of complexity arises with respect to storage of intermediate relations. We have assumed in this thesis that there is enough space on disk for storing intermediate relations. Since predicates on large datasets can be expected to be highly selective, this is a reasonable assumption in many cases. When this assumption does not hold, we either need to be able to modify the query plan to execute in parts as discussed in [SS94, ML95], or be able to store intermediate results on tertiary memory. This opens avenues for interesting research, especially on a sequential medium like tape. Where should the intermediate results be stored? How can we store the hash buckets to enable both fast writes during bucket creation stage and efficient reads during the probe phase? Similar issues arise when storing sorted runs during sort-merge joins.

Another assumption we have made is that the intermediate results are stored in an area different from the disk cache. This is a reasonable assumption in many installations where the disk cache comes as part of the tertiary memory device and can only be used for staging data in and out of tertiary memory. Removing this assumption greatly complicates the fragment transfer decisions because now the intermediate space will need to be managed as another resource. Scheduling of a subquery depends not only on the cached fragments but also on the amount of intermediate space that it will consume or free. The scheduling problem becomes more complex and interesting if we are willing to dynamically change execution methods inorder to trade off time for space. For example, changing a hash-join

method to nest-loop join, can enable us to schedule the join on cached fragments when scratch space is limited.

7.3.2 Caching directly to main memory

In this thesis we have assumed that all data is staged first to the disk cache before processing any queries on it. Sometimes, it might be more beneficial to cache data directly from tertiary memory to main memory. Bypassing the disk cache in certain cases, could reduce contention on disk and increase I/O parallelism. Myllymaki and Livny [ML95], for instance, propose methods for executing single two-way join queries where one of the relations is cached to disk and the other is streamed from a tape. However, this method of join processing may not be desirable in all cases. Since main memory sizes are a factor of 10 to 100 times smaller than disk sizes the amount of data that can be prefetched will be smaller. Most of our gain was obtained by scheduling large amounts of data to be transferred asynchronously. Also, streaming data from tertiary memory to main memory during join processing means that we are reserving drives for the entire duration of execution of the query. This might limit the number of concurrent user queries. Ideally, the scheduler should also manage main memory caching and exploit cases where it is more profitable to cache data directly to main memory.

7.3.3 Alternative storage configurations

So far we have assumed that data is stored only on a single tertiary memory device. An interesting extension is when data resides on more than one tertiary memory device, possibly of differing performance characteristics. We can extend our fragment fetch and eviction heuristics to handle multiple devices. First, we apply the same technique we used for increasing I/O parallelism between multiple drives of the same tertiary memory to increase I/O parallelism between multiple tertiary memory devices. Next, we modify the fetch policy so that we calculate the threshold values for each tertiary memory device and when the cache pressure mounts we fetch the next fragment from the device with the smaller value of the threshold first. Also, when evicting a fragment we replace the one which has lower cost of replacement.

7.3.4 Handling update queries

In this thesis, we optimized for read-only queries. Our belief is that in most cases data on tertiary memory will be loaded periodically in batch mode rather than updated interactively along with query processing. In cases where update queries are allowed concurrently with read-only queries, the job of the scheduler becomes more complicated. For instance, the decision of which fragment to fetch will be affected by the location (on tertiary memory) of the dirty blocks to be evicted from the cache to make space for this fragment. Similarly, the decision of where to place a dirty (new) block can be influenced by the pending read queries when there is no restriction on the location of the new block.

7.3.5 Data placement

In this thesis we concentrated only on placement of large multidimensional arrays. Placement of relations and other data types also presents some interesting problems, like, How is a relation partitioned across multiple platters? It intuitively seems profitable to place fragments that are queried together frequently on the same platter. But, this means that I/O requests might get sequentialized on the platter that holds all the frequently accessed data. Hence, it might help to spread requests across multiple platters to exploit drive parallelism. This raises many of the same kind of issues as declustering data across parallel disks. When seek cost is high, it might help to put the most frequently accessed data towards the beginning of the tape even if this requires spreading a relation across multiple platters. Sometimes it might be useful to store multiple copies of a relation organized in different ways since storage is cheap and updates are often infrequent. However, allowing for duplicates makes the query scheduling decision even more complicated.

7.4 Closing

In closing, it can be summarized that this thesis is a significant step towards improving performance of a tertiary memory database system. We concentrated on better caching, prefetching and query scheduling techniques to hide the access latency of data on tertiary memory devices. For the special class of multidimensional arrays we also studied methods of array organization to improve performance. However, there are a large number of topics that could still benefit from intense study by the database research community.

Bibliography

[AAFZ95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for assymetric communication environments. *Proc. ACM SIGMOD International Conference on Management of Data*, 24(2):199–210, 1995.

[ABC+76] M.M. Astrahan, M.W. Blasgen, D.D Chamberlin, et al. System r: A relational approach to database management. *acm Transactions on Database Systems*, 1(2):97–137, 1976.

[AC88] W. Alexandar and G. Copeland. Process and dataflow control in distributed data-intensive systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 90–98, 1988.

[AFZ96] S. Acharya, M. Franklin, and S. Zdonik. Prefetching from broadcast disks. In *Proc. International Conference on Data Engineering*, 1996.

[Ant93] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proc. International Conference on Data Engineering*, pages 538–547, 1993.

[AR92] J.R. Alsabbagh and V.V. Raghavan. A framework for multiple-query optimization. In *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 157–162, 1992.

[Bau93] P. Baumann. Database support for multidimensional discrete data. In *Proc. of Symposium on Large Spatial Databases*, pages 191–206, 1993.

[BK79] F.W. Burton and J. Kollias. Optimizing disk head movements in secondary key retrievals. *Computer Journal*, 22(3):206–8, Aug 1979.

[Bli90] J F Blinn. The truth about texture mapping. *IEEE computer graphics and applications*, 10:78–83, March 1990.

[BSCD91] P. Borla-Salamet, C. Chachaty, and B. Dageville. Compiling control into database queries for parallel execution management. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 271–279, Dec 1991.

[C+82] B. Collins et al. A network file storage system. In *Digest of Papers, Fifth IEEE Symposium on Mass Storage Systems*, pages 99–102, Oct 1982.

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

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Application-controlled caching, prefetching and disk scheduling. Technical Report TR-493-95, Princeton University, 1995.

[CG94] R.L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. *Proc. ACM SIGMOD International Conference on Management of Data*, 23(2):150–160, 1994.

[Che94] Ann Louise Chervenak. *Tertiary Storage: An Evaluation of New Applications*. PhD thesis, U. C. Berkeley, December 1994.

[CHH+91] J. Cheng, D. Haderle, R. Hedges, B.R. Iyer, et al. An efficient hybrid join algorithm: a DB2 prototype. In *Proc. International Conference on Data Engineering*, pages 171–80, Apr 1991.

[CHL93] M.J. Carey, L.M. Haas, and M. Livny. Tapes hold data, too: challenges of tuples on tertiary store. *Proc. ACM SIGMOD International Conference on Management of Data*, 22(2):413–417, 1993.

[CK92] Jr. Carino, F. and P. Kostamaa. Exegesis of DBC/1012 and P-90: industrial supercomputer database machines. In *Proceedings of the fourth International PARLE Conference*, pages 877–92, Jun 1992.

[CKSV93] K.M. Curewitz, P. Krishnan, and J. Scott Vitter. Practical prefetching via data compression. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 257–66, 1993.

[Cora] Exabyte Corporation. Exb-8505xl features. http://www.Exabyte.COM:80/Products/8mm/8505XL/Rfeatures.html.

[Corb] Quantam Corporation. Digital linear tape meets critical need for data backup. http://www.quantum.com/products/whitepapers/dlttips.html.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*, chapter 5,6. McGraw-Hill Book Company, 1984.

[CWCH93] S. Coleman, R. Watson, R. Coyne, and H. Hulen. The emerging storage management paradigm. In *Digest of Papers, Twelfth IEEE Symposium on Mass Storage Systems*, pages 101–110, Apr 1993.

[CYW92] M. Chen, P.S. Yu, and K. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *Proc. International Conference on Data Engineering*, pages 58–66, 1992.

[DHL+93] R. Drach, S. Hyer, S. Louis, et al. Optimizing mass storage organization and access for multi-dimensional scientific data. In *Proceedings Twelfth IEEE Symposium on Mass Storage Systems.*, pages 215–219, Apr 1993.

[DKL+94] D. Dewitt, N. Kabra, J. Luo, J. Patel, and J.Yu. Client-server paradise. In *Proc. International Conference on Very Large Databases*, 1994.

[DR91a] J. Dozier and H.K. Ramapriyan. Planning for the EOS data and information system. In *Global Environment Change*, volume 1. Springer-Verlag, Berlin, 1991.

[DR91b] J. Dozier and H.K. Ramapriyan. Planning for the EOS data and information system. In *Global Environment Change*, volume 1. Springer-Verlag, Berlin, 1991.

[Equ89] William H. Equitz. A new vector quantization clustering algorithm. *IEEE transactions on Accoustics, speech and signal processing*, 37(10), 1989.

[FMW90] J. Fahnestock, T. Myers, and E. Williams. Summary of the intelligence community's mass storage requirements. Technical Report SRC-TR-90-026, Supercomputing Research Center Institute for Defense Analyses, 1990.

[FP79] P C Fisher and R L Prower. Storage reorganization techniques for matrix computation in paging environments. *Communications of the ACM*, 22(7), 1979.

[FR94] C. Federighi and L. Rowe. A distributed hierarchical storage manager for a video on demand system. In *Storage and Retrieval for Image and Video Databases II, SPIE*, pages 185–97, Feb 1994.

[Fra92] James Franklin. Tiled virtual memory for UNIX. In *Proceedings of USENIX, San Antonio, TX*, pages 99–106, 1992.

[GK94] C.A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In *Advances in database technology*, pages 351–364, March 1994.

[GM81] J. Grant and J. Minker. Optimization in deductive and conventional relational database systems. *Advances in Data Base Theory*, 1, 1981.

[Gra90] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *Proc. ACM SIGMOD International Conference on Management of Data*, 19(2):102–11, 1990.

[Gra93a] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun 1993.

[Gra93b] Jim Gray. *The Benchmark handbook : for database and transaction processing systems*. Morgan Kaufmann series in data management systems. M. Kaufmann Publishers, San Mateo, Calif. ;, 2nd edition, c1993.

[Hal76] P.V. Hall. Optimizaton of a single relational expression in a relational database system. *IBM Journal of Research and Development*, 20(3):262–266, 1976.

[HC+90] L.M. Haas, W. Chang, et al. Starburst midflight: As the dust clears. *Transactions on Knowledge and Data Engineering*, 2(1):143–60, 1990.

[HCF+88] L.M. Haas, W.F. Cody, J.C. Freytag, et al. An entensible processor for an extended relational query language. Technical Report RJ 6182 (60892), IBM ALmaden Research Center, 1988.

[HGLW87] G. Herman, G. Gopal, K. Lee, and Weinrib. The datacycle architecture for very high throughput database systems. *Proc. ACM SIGMOD International Conference on Management of Data*, 16(3), 1987.

[Hof83] M. Hofri. Should the two headed disk be greedy?-yes, it should. *Information Processing Letters*, 16(2):83–5, Feb 1983.

[Hog90] C. Hogan. The Livermore distributed storage system: requirements and overview. In *Digest of Papers, Tenth IEEE Symposium on Mass Storage Systems*, pages 6–17, May 1990.

[HSa] B. Hillyer and A. Silberschatz. Storage technology: Status, issues and opportunities. Submitted for publication, available at http://cm.bell-labs.com/cm/is/what/strata/.

[HSb] B. Hillyer and A. Silberschatz. The strata tertiary storage system. http://cm.bell-labs.com/cm/is/what/strata/.

[HS93] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, Jan 1993.

[HS96a] B. Hillyer and A. Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *Proc. 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1996.

[HS96b] B. Hillyer and A. Silberschatz. Random i/o scheduling in online tertiary storage systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 195–204, 1996.

[Isa93] D. Isaac. Hierarchical storage management for relational databases. In *Proceedings Twelfth IEEE Symposium on Mass Storage Systems.*, pages 139–44, Apr 1993.

[IVB94] T. Imielinski, S. Viswanathan, and B. Badrinath. Energy efficient indexing on air. *Proc. ACM SIGMOD International Conference on Management of Data*, 23(2):25–36, 1994.

[Jag90] H V Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[JCL90] R. Juahari, M. Carey, and M. Linvy. Priority hints: An algorithm for priority based buffer management. In *Proc. International Conference on Very Large Databases*, pages 708–21, 1990.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[JS94] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Databases*, 1994.

[KAOP91] Randy H. Katz, T. Anderson, J. Ousterhout, and D. Patterson. Robo-line storage: High capacity storage systems over geographically distributed networks. Technical Report Sequoia 2000, 91/3, University of California at Berkeley, 1991.

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. *Proc. ACM SIGMOD International Conference on Management of Data*, 20(2):148–57, 1991.

[KMP90] J.G. Kollias, Y. Manolopoulos, and C.H. Papadimitriou. The optimum execution order of queries in linear storage. *Information Processing Letters*, 36(3):141–5, Nov 1990.

[Kot94] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. first USENIX Symposium on OS Design and Implementation*, 1994.

[LBG80] Yoseph Linde, Andres Buzo, and Robert Gray. An algorithm for vector quantizer design. *IEEE Transcations on Communications*, 28(1), 1980.

[LC93] C. Lee and Zue-An Chang. Workload balance and page access scheduling for parallel joins in shared-nothing systems. In *Proc. International Conference on Data Engineering*, pages 411–8, Apr 1993.

[Lor95] R. D. Lorentz, 1995. Presentation at Mass Storage Symposium.

[LY77] J. H. Liou and S. B. Yao. Multidimensional clustering for database organizations. *Information Systems*, 2:187–198, 1977.

[M+92] C. Mechoso et al. Parallelization and distribution of a coupled atmosphere-ocean general circulation model, 1992. sumitted to Monthly Weather Review, Aug 4 1992.

[MC69] A C McKellar and E G Coffman. Organizing matrices and matrix operations for paged virtual memory. *Communications of the ACM*, 12(3):153–165, 1969.

[MHWC90] C. Mohan, D Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: optimization, execution, and concurrency control techniques. In *Proc. International Conference on Extending Database Technology*, pages 29–43, 1990.

[MKY81] T. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling page-fetches in join operations. In *Proc. International Conference on Very Large Databases*, pages 488–98, Sep 1981.

[ML95] J. Myllymaki and M. Livny. Disk tape joins: Synchronizing disk and tape access. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling on Computer Systems*, May 1995.

[Moh93] C. Mohan. A survey of DBMS research issues in supporting very large tables. In *Proc. 4th International Conference on Foundations of Data Organization and Algorithms*, pages 279–300. Springer-Verlag, October 1993.

[MR93] M.C. Murphy and D. Rotem. Multiprocessor join scheduling. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):322–38, Apr 1993.

[N+87] M. Nelson et al. The National Center for Atmospheric Research Mass Storage System. In *Proc. Eighth IEEE Symposium on Mass Storage Systems*, pages 12–20, May 1987.

[Nei89] Patrick O' Neil. A set query benchmark for large databases. *Technical Report*, 22(2):2–11, 1989.

[NHS84] J Nievergelt, H Hinterberger, and K C Sevcik. The grid file: An adaptable symmetric multikey file structure. *ACM Transactions on Database systems*, 9(1), 1984.

[Ols92] Michael Allen Olson. Extending the POSTGRES database system to manage tertiary storage. Master's thesis, University of California, Berkeley, 1992.

[Ome92] R. Omerza. United parcel service DIALS overview. In *Proceedings Fourth Annual International DB2 User Group Conference*, May 1992.

[O'N94] Patrick O'Neil. *Database Principles, Programming, Performance*, chapter 8. ISBN 1-55860-219-4. Morgan Kaufmann, 1994.

[OOW93] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-k apge replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 297–306, 1993.

[OS95] Virginia Ogle and Michael Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9), Sep 1995.

[PGGo95] R.H. Patterson, G.A. Gibson, E. Ginting, and others. Informed prefetching and caching. In *Proc. Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[PI85] S. Pramanik and D. Ittner. Use of graph-theoretic models for optimal relational database accesses to perform join. *acm Transactions on Database Systems*, 10(1):57–74, Mar 1985.

[Ran91] Sanjay Ranade. *Jukebox and Robotic Libraries for Computer Mass Storage*. Meckler Publishing, 1991.

[RC87] J.E. Richardson and M.J. Carey. Programming constructs for database system implementation in EXODUS. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 208–19, 1987.

[RC88] A. Rosenthal and Upen S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proc. International Conference on Very Large Databases*, pages 230–239, 1988.

[RCM80] J L Reuss, S K Chang, and B H McCormick. Picture paging for efficient image processing. In S K Chang and K S Fu, editors, *Pictorial Information Systems*, pages 228–243. Spriger-Verlag, 1980.

[RFJ+93] M.F. Riley, J.J. Feenan Jr., et al. The design of multimedia object support in DEC Rdb. *Digital Technical Journal*, 5(2):50–64, 1993.

[RN91] Timos Sellis Raymond Ng, Christos Faloutsos. Flexible buffer allocation based on marginal gains. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 387–396, 1991.

[Ros75] Arnold L. Rosenberg. Preserving proximity in arrays. *SIAM journal on Computing*, 4:443–460, 1975.

[S+96] Michael Stonebraker et al. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1), Jan 1996.

[SB+93] T. Stephenson, R. Braudes, et al. Mass storage systems for image management and distribution. In *Digest of Papers, Twelfth IEEE Symposium on Mass Storage Systems*, pages 233–240, Apr 1993.

[SD91] Michael Stonebraker and Jeff Dozier. Large capacity object servers to support global change research. Technical Report 91/1, University of California at Berkeley, 1991.

[Sel93] P. Selinger. Predictions and challenges for database systems in the year 2000. In *Proc. International Conference on Very Large Databases*, pages 667–675, 1993.

[SK91] M. R. Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10), 1991.

[SLM93] B. Seeger, P. Larson, and R. McFadyen. Reading a set of disk pages. In *Proc. International Conference on Very Large Databases*, pages 592–603, 1993.

[Smi81] A.J. Smith. Long term file migration: development and evaluation of algorithms. *Communications of the ACM*, 24(8):521–32, Aug 1981.

[Son89a] Sony Corporation, Japan. *Writable Disk Auto Changer WDA-610 Specifications and Operating Instructions*, 1989. 3-751-106-21(1).

[Son89b] Sony Corporation, Japan. *Writable Disk Drive WDD-600 and Writable Disk WDM-6DL0 Operating Instructions*, 1989. 3-751-047-21(1).

[SP89] A. Segev and J. Park. Identifying common tasks in multiple access paths. Technical Report LBL-27877, Lawrence Berkeley Laboratory, 1989.

[SS86] G.M. Sacco and M. Schkolnick. Buffer management in relational database systems. *acm Transactions on Database Systems*, 11(4):473–98, 1986.

[SS94] S. Sarawagi and M. Stonebraker. Single query optimization in tertiary memory. Technical Report Sequoia 2000, S2k-94-45, University of California at Berkeley, 1994.

[SSU95] Avi Silberschatz, Mike Stonebraker, and Jeff Ullman. Database research: Achievements and opportunities into the 21st century. Report of an NSF Workshop on the Future of Database Systems Research, http://db.stanford.edu/pub/ullman/1995/lagii.ps, May 1995.

[Sto91a] M. Stonebraker. Managing persistent objects in a multi-level store. *Proc. ACM SIGMOD International Conference on Management of Data*, 20(2):2–11, 1991.

[Sto91b] Michael Stonebraker. An overview of the Sequoia 2000 project. Technical Report 91/5, University of California at Berkeley, 1991.

[SW94] K.E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, Sep 1994.

[TG84] J.Z. Teng and R.A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–18, 1984.

[W+] Robert Wilensky et al. Uc berkeley digital library project. http://elib.cs.berkeley.edu/.

[Wad84] B T Wada. A virtual memory system for picture processing. *Communications of the ACM*, 27:444–454, 1984.

[Wei] William Weibel. Personal Communication.

[Wie87] G. Wiederhold. *File organization for database design*. McGraw-Hill, New York, 1987.

[YC91] Philip S. Yu and Douglas W. Cornell. Optimal buffer allocation in a multi-query environment. In *Proc. International Conference on Data Engineering*, pages 622–631, 1991.

[YD96] Y.Yu and D. Dewitt. Query pre-execution and batching in paradise. In *Proc. International Conference on Very Large Databases*, 1996.

[YSLM85] C.T. Yu, Cheing-Mei Suen, K. Lam, and M.K.Siu. Adaptive record clustering. *ACM Transactions on Database Systems*, 10:180–204, 1985.

[Yu95] Andrew Yu. Buffer management for tertiary storage devices. Master's thesis, University of California, Berkeley, 1995.

[Y.W95] Y.Wang. DB2 query parallelism: Staging and implementation. In *Proc. International Conference on Very Large Databases*, pages 686–91, 1995.