# Relational Joins for Data on Tertiary Storage

Jussi Myllymaki
Miron Livny

Technical Report #1331

January 1997

# Relational Joins for Data on Tertiary Storage *

Jussi Myllymaki Miron Livny

Computer Sciences Department
University of Wisconsin–Madison
{jussi,miron}@cs.wisc.edu

**Abstract**

Despite the steady decrease in secondary storage prices and continuous, sometimes drastic, improvements in storage density, the data storage requirements of many organizations cannot be met economically using secondary storage alone. Tertiary storage solutions, especially automated magnetic tape libraries, offer a much lower storage cost at a level of functionality that satisfies the needs of many application types.

Database management systems (DBMS) typically do not incorporate tertiary storage devices such as magnetic tapes and optical disks as a first-class citizen in the storage hierarchy. The typical solution in bringing tertiary-resident data under the control of a DBMS is to use operating system facilities to copy the data to secondary storage, and then to perform query optimization and execution as if the data had been in secondary storage all along. This approach fails to recognize the opportunities for saving execution time and storage space if the data were accessed on tertiary devices directly and in parallel with other I/Os.

In this paper we examine ways of joining two relations stored on magnetic tapes. Our earlier work has shown that when one relation is stored on tape and the other on disk, a parallel I/O variant of Nested Block Join performs quite well, given sufficient main memory space. Work presented in this paper extends by considering the case where both relations are larger than available disk space. To tackle main memory size limitations, we focus on hashing-based solutions. We modified Grace Hash Join to handle a range of tape relation sizes and to exploit parallelism between disk and tape I/Os. We show how disk and main memory space affect the performance of tertiary joins and demonstrate how parallel I/O helps these methods save execution time as well as memory and disk space.

**Keywords:** tertiary storage, join methods, parallel I/O

## 1 Introduction

In many of today's businesses and scientific communities, the quantity of information available for analysis and further refinement is so vast that it is not economical to store all that data on secondary storage. Both cost and limited physical space contribute to the fact that many types of data, especially the kinds used for data analysis and data mining, are stored on tertiary storage

*An abridged version of this paper appears in the Proceedings of the International Conference on Data Engineering, Birmingham, UK, April 1997

devices such as magnetic tapes and optical disks. Only the most frequently used data is stored in secondary storage, for instance, meta-data, indices, small datasets, or subsets of larger datasets.

Database management systems (DBMS) are typically not capable of operating on tertiary-resident data in the same manner as they handle secondary storage [4]. One reason is the wide variety and dissimilarity of tertiary storage devices, another is the difference in how tertiary storage and secondary storage are accessed at the device level. Magnetic tape storage, for instance, is inherently sequential with slow random-access capabilities. It involves lengthy setup and media exchange times and may deliver only a modest data transfer rate. The difficulty in interfacing tertiary storage intelligently with a DBMS arises not only from potential connectivity and compatibility problems with software drivers and hardware interfaces, but also from the requirement that the query optimizer of the DBMS must have a very detailed and enriched model of tertiary storage devices to plan the execution and device accesses appropriately.

The typical solution in bringing tertiary-resident data under the control of a DBMS is to use operating system facilities to copy all tertiary-resident data to secondary storage, and then to optimize and process the query as if the data had been in secondary storage all along. This approach fails to recognize the opportunities for saving execution time and storage space if the data were accessed on tertiary devices directly and in parallel with other I/Os. Also, this approach fails completely if not enough secondary storage space exists to stage the entire dataset from tertiary storage.

The driving force in our studies is to be able to efficiently compute very large joins directly on tertiary storage using workstations, thereby making database applications similar to data mining possible without mainframe-size machinery and investment. We focus on ways to join two relations stored on magnetic tapes, examining both methods that require the smaller relation to fit on disk (*disk–tape joins*) and methods that do not have this restriction (*tape–tape joins*). Our goal is to show how disk and main memory space affect the performance of tertiary joins and to demonstrate how parallel I/O helps save execution time as well as memory and disk space.

In our earlier work [13], we have examined the case where one relation is stored on tape and the other on disk. Using an analytical model, we have shown that a parallel I/O variant of Nested Block Join [11] performs very well when at least half the smaller relation fits in main memory but very poorly when little main memory is available. To join relations much larger than main memory or disk space, one has to employ hashing techniques.

In this paper we present modifications to Grace Hash Join [5] to make it work on tape storage and to exploit parallel I/O and double-buffering techniques described in [14]. We describe and evaluate both disk–tape and tape–tape join methods based on Grace Hash Join. We also review algorithms developed in [13] and apply them in the context of two tape relations. For all tertiary join methods examined, we show what the resource requirements are and analyze the performance

via an experimental implementation. In one of the experiments, a variant of Grace Hash Join utilizing parallel I/O performed the join of a 10 GB tape relation with a 2.5 GB tape relation in 14 hours, just 7 times the bare read time of the two relations. The experiment used a Pentium workstation with 500 MB of disk space and 32 MB of main memory.

This paper is organized as follows. In Section 2 we discuss literature related to the study of tertiary joins. The system model we employ for describing and analyzing tertiary joins is presented in Section 3. We discuss buffering techniques for tertiary join methods using parallel I/O in Section 4. The tertiary join methods are described in Section 5. Section 6 describes the experimentation environment used in the analysis of these algorithms. The results of the experiments are described and analyzed in Sections 7 through 9. Section 10 concludes the paper.

## 2 Related Work

Our investigation of database joins on tertiary storage was prompted by a SIGMOD Database Challenges paper [4] which pointed out that DBMS control of tertiary storage was becoming ever more important because of the large volumes of tertiary-resident data that are available for analysis and manipulation. While multimedia systems such as video-on-demand servers have successfully integrated tertiary storage into their storage hierarchies [6,10,18], relational database systems still have little notion of incorporating tertiary storage as an integral part of the system.

Joining two relations is one of the most common operations in a relational DBMS and one of the most costly if done naively. The database literature contains an extensive collection of work on optimizing joins, started by the seminal paper by Blasgen and Eswaran [2]. Much of the work has been devoted to optimizing *ad hoc* equi-joins, i.e. joins where an exact match of join attributes is requested and which do not rely on the existence of pre-computed access structures such as indices. Most studies consider main memory and secondary storage as the only forms of storage, but few studies include tertiary storage in the system model. Studies on disk-based joins typically employ a transfer-only cost model [2,3,5,11,17] where the number of pages transferred is the cost metric while the latency penalty of small I/O requests is disregarded. The number of multi-page I/O requests was the cost metric in [8], and a detailed cost model combining both cost models was developed in [7].

The optimization and execution of queries on tertiary-resident data in the Postgres database system is described in [15,16]. In that architecture, a query on large tertiary relations is broken into smaller, independent subqueries on fragments of the relations. The system caches fragments in secondary storage and improves tape access efficiency by reordering the I/O requests resulting from the subqueries. A similar reordering of tape I/O requests is performed by the tertiary storage interface of the Paradise database system [19]. The system pre-executes queries, collects and

reorders the tape I/O references, and then re-executes the queries. As in the Postgres system, the ordering and batching of tape I/O requests yields improved tape data transfer efficiency.

In our earlier work, we have reported on ways to join a tape relation with a disk relation when direct access to tertiary storage by the join methods is made possible [13]. The results indicated that significant savings in execution time and buffer space (both main memory and disk space) can be achieved by customizing existing relational join algorithms in two ways. First, the customized algorithms take advantage of the apparent mismatch in secondary and tertiary storage device speeds and use only a small amount of disk space for buffering input from the tape relation as opposed to copying the entire relation to disk. Second, some tape I/Os can be executed in parallel with disk I/Os, thereby reducing the response time significantly. Main memory size and the speed ratio between the disk and tape devices were identified as the key components determining the performance of a join of a disk and tape relation.

The buffering techniques discussed in [13] were applied in a more general context, extensible to data mining and data visualization applications, in [14]. The paper also provided an analysis of how the performance characteristics and interaction of main memory, process scheduling, I/O bus and storage devices affect parallel I/O throughput.

In our current studies we focus on joins between two tertiary-resident relations. We modify hashing-based join methods to operate on tapes and to exploit parallel I/O techniques developed in our earlier work. Our goal is to show how disk and main memory space affect the performance of these joins, and demonstrate how parallel I/O helps them save execution time as well as memory and disk space. The distinction between our line of work and that presented in [16] is that our join algorithms access data directly on tertiary devices, using available disk space both as a speed-matching buffer and as a cache. Our performance results are from an experimental implementation rather than a simulation. Whereas [16] considers different types of queries, our focus is on joins without indices.

A study on tape I/O scheduling [9] presents an interesting and detailed model of the random access behavior of the same tape drive model we use in our studies. The tertiary join algorithms we study exhibit mostly sequential tape I/O patterns, complementing the picture drawn by [9].

## 3 System Model for Tertiary Joins

We now present our system model for analyzing relational joins in a tertiary storage environment. As in our earlier work [13], the model is a simplified view of computer system components and their interaction because the goal of our studies is to perform qualitative analysis and to understand the interdependency of system resource requirements and join execution time.

Table 1: Summary of Notation

| | | | |
|---|---|---|---|
| R | smaller relation (tape R) | S | larger relation (tape S) |
| $\|R\|$ | size of R in blocks | $\|S\|$ | size of S in blocks |
| $T_R$ | tape space on R tape | $T_S$ | tape space on S tape |
| $M$ | memory space in blocks | $D$ | disk space in blocks |
| $X_D$ | aggregate speed of disks | $X_T$ | speed of a tape drive |
| $n$ | number of disk drives | | |

### 3.1 System Configuration and Notation

The task of a tertiary join method is to compute the join of two tape relations R and S. The relations are stored on separate tapes and their sizes are $|R|$ and $|S|$, respectively. We define R to be the smaller relation ($|R| < |S|$). Scratch tape space is available on both tapes, $T_R$ blocks on tape R and $T_S$ blocks on tape S. We assume a computer system with two tape drives (one for tape R, the other for tape S) and $n$ disk drives ($n \geq 2$). $D$ blocks of disk space are available for the join, evenly divided on the $n$ disks. A fixed amount of main memory ($M$ where $M < |R|$) is allocated for use in the join. The CPU, the tape drives, the disk drives, and the allocated main memory are reserved exclusively for the join.

The aggregate, sustained data transfer rate of the $n$ disks is denoted by $X_D$ and is assumed constant. The performance of each tape drive is characterized by a constant, sustained transfer rate ($X_T$). We view each tape drive as an abstraction which need not physically correspond to a single tape drive. A tape drive may in fact be an array of tape drives, in which case $X_T$ is the aggregate transfer rate of the array.

The notation we use in the paper is summarized in Table 1.

### 3.2 Cost Model and Assumptions

We employed a transfer-only I/O cost model and made a number of simplifying assumptions in order to present the algorithms and their analysis more clearly. First, we assume that I/O times are the dominating factor in the join and that CPU cost can be ignored. Therefore, the total cost of the join (response time) is roughly the same as I/O cost, like in most other cost models for joins.

Join method analyses typically do not consider query output costs since they are deemed the same for all methods [2, 5, 7, 8, 17]. In our study, however, it seems necessary to include the output costs in the cost model because the input cost may be affected by the output cost. If the query output is simply pipelined to an unrelated process capable of receiving and processing data at the output rate, we can ignore the effect of the output because only a small main memory buffer is needed in the join process. A similar situation arises when the join operator pipelines its output to

an aggregate operator or an operator with high selectivity, both of which reduce the output data volume significantly.

A natural case where the output cost is more likely to affect the input cost is when the join method is required to store the query output locally on disk. The resulting disk writes reduce the bandwidth available for reads on the disk(s) involved. We assume that if the join output is to be stored locally, the effect of writing the output has been taken into account in $X_D$. In other words, $X_D$ represents the reduced, aggregate disk bandwidth available for the join itself.

We assume that all disk accesses are multi-page I/O requests. The cost of a disk access is therefore derived by counting the number of blocks transferred. The seek cost and rotational latency are ignored. As shown in [7], disk seeks and rotational latency play a relatively minor role compared to transfer cost when disk requests are at least moderately large. In our model, the size of all disk requests is assumed to be at least 30 blocks, making seek and latency costs negligible[1]. Such disk accesses incur a fixed per-block transfer cost.

Tape media switch delays which occur in tape robot systems are not included in this cost model. In the join methods we examine, the tapes are read sequentially end-to-end, making tape switch delays (roughly 30 seconds per media exchange) negligible compared to the transfer time of a full tape (several hours). Without loss of generality or accuracy, we assume that each relation fits on a single tape and that the tapes have been inserted and loaded into the tape drives before the join operation begins.

We also ignore tape rewind times for two reasons. First, many medium-to-high performance tape drives store data on serpentine tracks, making the rewind time of large tape files orders of magnitude less than the read time. For instance, a 5 GB tape file might take an hour to read but only 10 seconds to rewind. Second, some tape drives can read in reverse direction[2] which would make rewinds unnecessary in all the algorithms we examine, as the algorithms are independent of the order (direction) in which tuples or buckets of tuples are scanned.

A tape drive is said to be in *streaming* mode if its internal read-ahead buffer never fills up and therefore the drive never stops. Otherwise the drive operates in *stop/start* mode and incurs repositioning delays due to mechanical motions. For simplicity, we assume that the tape drive has enough buffer memory to hide these delays.

# 4 Buffering Techniques for Tertiary Joins Using Parallel I/O

The join of two tape relations cannot be computed in a single iteration when the smaller relation is larger than available main memory and the combined size of the relations exceeds available disk

[1]Disk caching would reduce seek and latency costs even if requests were smaller than 30 blocks.

[2]The SCSI standard defines the command READ REVERSE but its implementation by tape drive manufacturers is left optional [1]. On a historical note, Knuth also assumes bi-directional tape drives in his work on tape sorting [12].

space. An iterative way to compute the join is to read relation S in pieces, denoted by $S_i$, and compute a mini-join $S_i \bowtie R$ in each iteration $i$. The iterative phase is preceded by a setup phase where relation R is "prepared" by copying or hashing it from tape to disk, or by hashing it from tape to tape.

The iterative nature of the join raises the possibility of effective parallelism between I/Os made in iteration $i$ and those made in iteration $i+1$. One particular technique join methods can use is *double-buffering*, in which a reader process reads $S_i$ from one memory or disk buffer and joins it with R while a writer process fetches $S_{i+1}$ from tape and stores it in another buffer [14]. The simple approach is to split existing buffer space in two halves, but this makes each $S_i$ half the original size. This in turn doubles the number of iterations needed and doubles the number of times R is scanned. Furthermore, the average utilization of available buffer space is only 50% assuming that the writer and reader processes operate at equal speeds.

A better approach is to interleave the accesses to the two buffers so that as space in the buffer holding $S_i$ is gradually released, the space is immediately reused by storing more of $S_{i+1}$. In effect, there is just one physical buffer shared by two logical buffers. The number of iterations is unchanged despite the use of double-buffering. Also, buffer utilization can stay at 100% during the entire execution of the join.

For *main memory* buffers, a simple circular buffer implementation is sufficient for this purpose. But with double-buffered *disk* space, interleaving the accesses in the prescribed manner requires disk scheduling that prevents the disk writes of iteration $i+1$ from interfering with the reads of iteration $i$, otherwise the benefits of I/O parallelism would be lost. Striping the buffer across $n$ disks using an ordinary RAID would not work because we need finer control over the placement of disk blocks and usage of disk arms. One therefore needs special disk striping routines to balance the consumption of bandwidth and storage space on the disks involved.

# 5 Tertiary Join Methods

We now describe tertiary join methods and highlight their requirements for main memory and disk space. First, we examine disk–tape join methods, that is, methods for joining two tape relations when enough disk space exists to hold the smaller tape relation ($D \geq |R|$). We then discuss tape–tape join methods which do not have this restriction. Finally, we summarize the resource requirements of the algorithms and present an analytical comparison of their performance. The results of an experimental implementation are presented in subsequent sections.

## 5.1 Disk–Tape Join Methods

### 5.1.1 Disk–Tape Nested Block Join (DT-NB)

The sequential Disk–Tape Nested Block Join (DT-NB) computes the disk–tape join as follows. Main memory $M$ is split into $M_R$ blocks of buffer space for R input and $M_S$ blocks of buffer space for S input. In Step I, relation R is copied from tape to disk. In Step II, $M_S$ blocks of relation S are read into memory and then relation R is scanned to produce the join tuples. Step II is iterated for each $M_S$-size chunk of S until it is exhausted. The chunk of S read in iteration $i$ is $|S_i| = M_S$.

### 5.1.2 Disk–Tape Grace Hash Join (DT-GH)

The disk–tape join is computed sequentially by Disk–Tape Grace Hash Join (DT-GH) as follows. In Step I, relation R is read from tape and partitioned into hash buckets on disk. In Step II, $d = D - |R|$ blocks of data are read from S, hashed, and written to buckets on disk. Each R hash bucket is read back into memory in turn and joined with the corresponding S hash bucket by scanning it. Step II is iterated until S is exhausted.

The number of hash buckets is $B = \frac{|R|}{M}$ where $M > \sqrt{|R|}$ (see [5]). We assume that hash values are uniformly distributed, that is, the hash buckets for R are equal-sized. These two conditions guarantee that each R hash bucket fits into memory when read back from disk.

### 5.1.3 Concurrent Disk–Tape Nested Block Join (CDT-NB)

Concurrent Disk–Tape Nested Block Join (CDT-NB) is a parallel I/O variant of DT-NB and was first described in [13]. As in DT-NB, relation R is first copied from tape to disk. CDT-NB performs the join by reading a chunk of S into a main memory or disk buffer and joining the previous chunk simultaneously with R. This method has two variations. CDT-NB with main memory buffering (CDT-NB/MB) requires two main memory buffers to achieve I/O parallelism[3]. CDT-NB with disk buffering (CDT-NB/DB) uses double-buffered disk space to which accesses are interleaved as described in Section 4.

The details of the two variations are as follows. In CDT-NB/MB, main memory space $M$ is divided into one R buffer ($M_R$) and two S buffers ($M_S = \frac{M-M_R}{2}$). Each iteration $i$ of the algorithm consists of reading $M_S$ blocks of data from S into one memory buffer, while joining the previous $M_S$ blocks of data from the other buffer with R. This process is iterated until S is exhausted. The chunk of S read in iteration $i$ has size $|S_i| = M_S$.

[3] Interleaved double-buffering (Section 4) cannot be applied in CDT-NB/MB because the contents of a memory buffer are needed throughout the iteration. The buffer space is released only at the end of the iteration, thus requiring a second buffer in which data for the next iteration is stored.

Substituting disk buffer space for memory space allows larger chunks of data to be read from S in each iteration, thus reducing the number of times relation R is scanned. Instead of splitting main memory space into two halves, CDT-NB/DB uses a disk buffer of size $M_S = M - M_R$ to hold a chunk of S. In each iteration $i$, a full disk buffer is read into memory and joined with R, while the disk buffer is refilled in parallel. Note that $|S_i| = M_S$ holds for both CDT-NB/DB and CDT-NB/MB, but $M_S$ is twice as large in CDT-NB/DB.

### 5.1.4 Concurrent Disk–Tape Grace Hash Join (CDT-GH)

Concurrent Disk–Tape Grace Hash Join (CDT-GH) is a parallel I/O variant of DT-GH that first hashes relation R from tape to disk and then uses the remaining disk space ($d = D - |R|$) to double-buffer the hash buckets of S.

The details of CDT-GH are as follows. In Step I, relation R is read from tape and partitioned into hash buckets which are written to the disks in stripes. Step II is iterated until S is exhausted. In each iteration $i$, $|S_i| = d$ blocks of S data are read from tape, hashed, and written to hash buckets which are striped on all $n$ disks. A join process then reads each bucket of R into memory and joins it with the corresponding bucket of S. As described in Section 4, a hash process can simultaneously read more data from S and produce the hash buckets needed in the next iteration.

CDT-GH has the same main memory requirement as DT-GH, namely $M > \sqrt{|R|}$.

## 5.2 Tape–Tape Join Methods

### 5.2.1 Concurrent Tape–Tape Grace Hash Join (CTT-GH)

The key difference between Concurrent Tape–Tape Grace Hash Join (CTT-GH) and CDT-GH is that CTT-GH creates a hashed version of R on tape, not on disk. The disk is used as an assembly area and the assembled buckets are appended to the R tape. Once Step I is complete, CTT-GH iteratively produces hash buckets of S on disk (as in CDT-GH) and joins them with the tape-resident buckets of R. The disk space available for double-buffering S is $|S_i| = d = D$.

The details of Step I are as follows. Relation R is scanned $\left\lceil \frac{|R|}{D} \right\rceil$ times. In each scan, a fraction of R buckets are generated, in full, on the disk. The number of buckets completed in each scan is $\frac{D}{M}$, the product of the total number of buckets $B = \frac{|R|}{M}$ and the fraction of R that fits on disk $\frac{D}{|R|}$. The buckets resulting from each scan are appended to the R tape. Once all scans have been performed, the R tape will have a complete, hashed copy of R, and all buckets are stored contiguously.

CTT-GH has the same main memory requirement as both DT-GH and CDT-GH, namely $M > \sqrt{|R|}$.

Table 2: Resource Requirements of Tertiary Join Methods

| method | symbol | $M$ | $D$ | $T_R$ | $T_S$ |
|---|---|---|---|---|---|
| Disk–Tape Nested Block Join | DT-NB | $\|S_i\|$ | $\|R\|$ | 0 | 0 |
| Concurrent Disk–Tape Nested Block Join with Memory Buffering | CDT-NB/MB | $2\|S_i\|$ | $\|R\|$ | 0 | 0 |
| Concurrent Disk–Tape Nested Block Join with Disk Buffering | CDT-NB/DB | $\|S_i\|$ | $\|R\| + \|S_i\|$ | 0 | 0 |
| Disk–Tape Grace Hash Join | DT-GH | $\sqrt{\|R\|}$ | $\|R\| + \|S_i\|$ | 0 | 0 |
| Concurrent Disk–Tape Grace Hash Join | CDT-GH | $\sqrt{\|R\|}$ | $\|R\| + \|S_i\|$ | 0 | 0 |
| Concurrent Tape–Tape Grace Hash Join | CTT-GH | $\sqrt{\|R\|}$ | $\|S_i\|$ | $\|R\|$ | 0 |
| Tape–Tape Grace Hash Join | TT-GH | $\sqrt{\|R\|}$ | any | $\|S\|$ | $\|R\|$ |

### 5.2.2 Tape–Tape Grace Hash Join (TT-GH)

In Step I, Tape–Tape Grace Hash Join (TT-GH) creates a hashed version of R to the *S tape*. The S tape is used as the target in order to eliminate tape seeks between the source and destination locations. As in CTT-GH, hashing R requires $\left\lceil \frac{|R|}{D} \right\rceil$ scans over it. Relation S is then hashed using the same procedure and the buckets are appended to the *R tape*. As a result of this phase, the complete hash buckets of relation R reside contiguously on the S tape, and the hash buckets of relation S reside on the R tape.

In Step II, each bucket of relation R is read into memory, and the corresponding bucket of relation S is scanned. Buckets of both relations are stored in the same order, although not necessarily in any particular order.

## 5.3 Resource Requirements and Expected Response Time

The resource requirements of the tertiary join methods are summarized in Table 2. For all methods using iterative computation, $S_i$ refers to the piece of S consumed in iteration $i$. Note how the type of storage space used (memory, disk, tape) changes diagonally, with DT-NB consuming the most main memory, CDT-GH consuming the most disk space, and TT-GH consuming the most tape space.

The following three charts show the expected response time of the join methods, relative to the tape read time of relation S. We varied $|R|$ relative to $M$, but fixed $|S| = 10|R|$ and $D = 32M$. The aggregate disk speed was assumed to be twice the tape speed ($X_D = 2X_T$). The response times were calculated using cost formulas derived for each join method. The derivation and the resulting

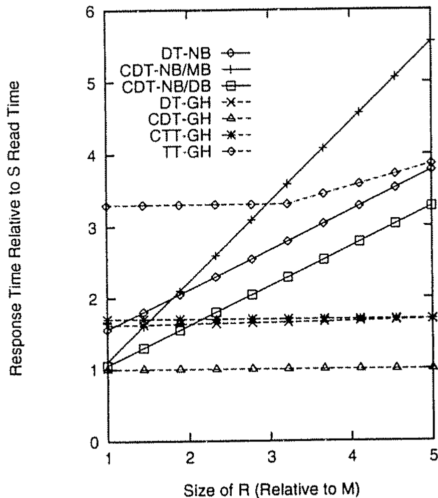


Figure 1: Small $|R|$.

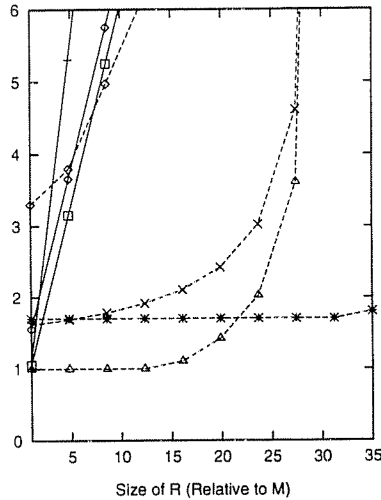


Figure 2: Medium $|R|$.

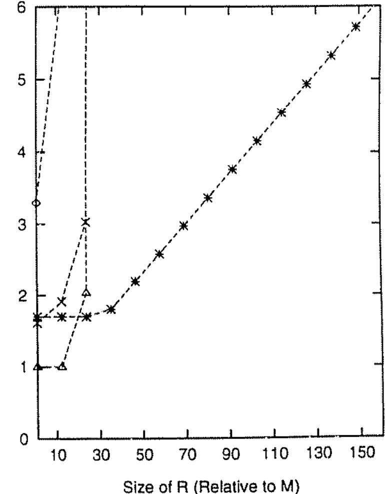


Figure 3: Large $|R|$.

formulas are based on [13] but are beyond the scope of this paper.

In Figures 1 through 3 we examine three interesting ranges of $|R|$. Figure 1 shows the case where $|R|$ is comparable to $M$ ($M$ appears at value 1). The chart illustrates how the response time of join methods based on Nested Block Join depends on the memory size because that determines the number of iterations required. The number of iterations made by hashing-based join methods, on the other hand, depends on the available disk space. Therefore the response times in this $|R|$ range are fairly constant. These analytical results are validated by the experimental results shown in Section 9.

Figure 2 shows the case where $|R|$ ranges up to $D$ ($D$ appears at value 32). When $|R|$ approaches $D$, less and less space is left to buffer S, which in turn increases the number of iterations required in DT-GH and CDT-GH. The high setup cost of TT-GH (which requires both relations to be hashed from tape to tape) rules it out of the competition for very large $|R|$. The response time of CTT-GH, on the other hand, appears largely unaffected by the increased size of R. In Section 8 we compare these analytical results with the results of an experimental implementation.

The next chart (Figure 3) examines the case where $|R|$ is far beyond $M$ and $D$. We see that CTT-GH appears to perform quite well and scales up gracefully. This result is validated via an experimental implementation in Section 7.

## 6 Implementation and Measurement

We implemented the tertiary join methods on a 90 MHz Pentium system running Solaris 2.4. The computer has 32 MB of main memory and two Fast SCSI-2 buses. One SCSI bus has two disks (Quantum Lightning 540 and Quantum Fireball 1080) and one tape drive (Quantum DLT-4000)

Table 3: Parameters and Execution Time of Concurrent Tape–Tape Grace Hash Join

| | $\|S\|$ (MB) | $\|R\|$ (MB) | $D$ (MB) | Read S + R | Step I | Steps I + II | Rel. Cost |
|---|---|---|---|---|---|---|---|
| Join I | 1,000 | 500 | 100 | 895 sec. | 2765 sec. | 7112 sec. | 7.9 |
| Join II | 2,500 | 1,250 | 250 | 2237 sec. | 5598 sec. | 16227 sec. | 7.3 |
| Join III | 5,000 | 2,500 | 500 | 4475 sec. | 10260 sec. | 30783 sec. | 6.9 |
| Join IV | 10,000 | 2,500 | 500 | 7468 sec. | 10260 sec. | 50565 sec. | 6.8 |

attached to it. The other bus has one disk (Quantum Fireball 1080) and one tape drive (Quantum DLT-4000). The tape drives were used in the 20 GB density mode with compression enabled.

We ran three kinds of experiments, all with synthetic data stored in relations S and R. In Experiment 1, we measured the response time of a join of two large tape relations. Four measurements were performed, with the size of relation S ranging from 1,000 to 10,000 MB, relation R ranging from 500 to 2,500 MB, and disk space ranging from 100 to 500 MB. Main memory was fixed at 16 MB.

In Experiment 2, we examined the case where $|R|$ is comparable to $D$ and focused on how a variation in disk space affects the response time of hashing-based join methods. Experiment 3 measured the effect of main memory size on join methods when $|R|$ was comparable to $M$.

In join methods based on Nested Block Join, we allocated 10% of $M$ for scanning relation R from disk and 90% of $M$ for buffering S. In CDT-NB with disk buffering, a main memory buffer is needed for transferring data from tape to a disk buffer. This buffer is very small compared to $M$ and its effect is ignored in the analysis.

In hashing-based join methods, a main memory buffer is needed for reading S, computing the hash values for the tuples, and then writing the tuples to hash buckets on disk. The buffer allows for larger disk writes which help reduce the seek penalty, as appending data to hash buckets on disk involves random I/O. When the number of buckets is large, the size of this main memory buffer becomes significant and is therefore included in $M$.

# 7 Results of Experiment 1: Large S, Large R

In this experiment, we examined the case where two large tape relations are joined using limited main memory and disk space. The join method used in this experiment is Concurrent Tape–Tape Grace Hash Join. In Joins I through III, $|S|$ was increased from 1,000 MB to 5,000 MB, while keeping $|R|$ at one-half of $|S|$. In Join IV, $|S|$ was 10,000 MB and $|R|$ was 2,500 MB. For all joins, the total disk space was one-fifth of $|R|$ and was evenly distributed on two disks. Memory size was fixed at 16 MB.

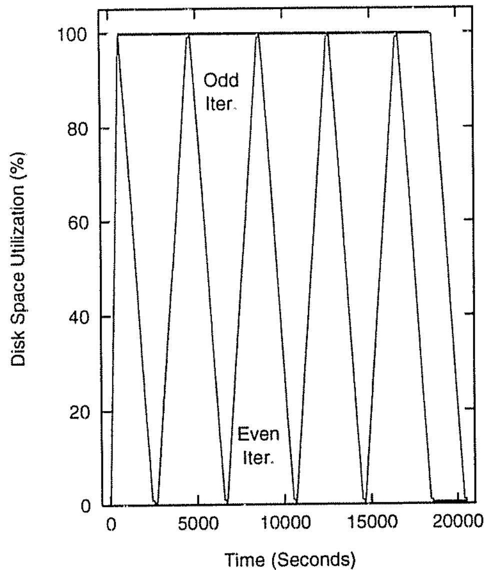


Figure 4: Disk Space Utilization in CTT-GH (Step II of Join III).

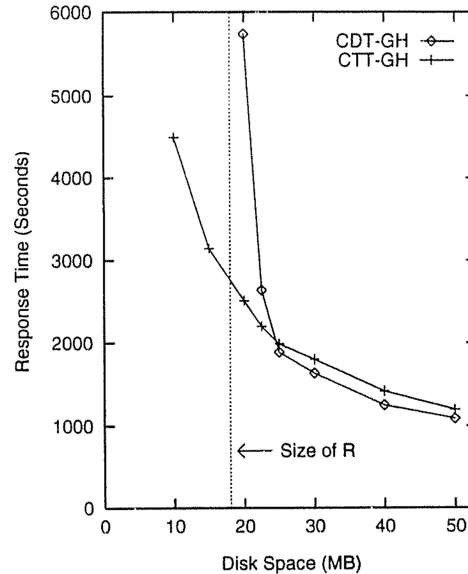


Figure 5: Impact of Disk Space on CDT-GH and CTT-GH.

Table 3 summarizes the parameters of this experiment and shows the total response time (Steps I and II) of the joins. The table also shows the bare read time of the two tape relations, the time required to hash relation R (Step I of CTT-GH), and the relative cost of the joins (response time divided by bare read time). In Join III, for example, just reading S and R once, with no processing of the data, took almost 4,500 seconds (1.25 hours). Relation R was hashed to tape in a little under 3 hours, and the join was completed in 8.5 hours. The relative cost (6.9) agrees with the expected value shown in Section 5.3. Note that the relative cost is more representative of the join method's speed than the absolute response time, since the ratio would remain unchanged even if newer, faster tape technology were used in the join. Also note that increasing $|S|$ without changing the other parameters reduces the relative cost of a join because the setup cost (Step I) is amortized over a longer overall execution time.

Figure 4 demonstrates the efficiency of disk space usage in an interleaved double-buffering scheme. The figure shows the disk space utilization during Step II of CTT-GH (Join III). The lower, shark-toothed line represents the usage of disk space by the even-numbered iterations, while the area between the two lines represents disk utilization by the odd-numbered iterations. The upper line, at or near 100%, is the total disk space utilization.

# 8 Results of Experiment 2: Large S, Medium R

The goal of this experiment is to join a large tape relation with a smaller tape relation whose size is comparable to disk space. The experiment illustrates how variations in available disk space

affect Concurrent Disk–Tape and Tape–Tape Grace Hash Join. In this experiment, we chose a very small R (18 MB) to speed up experimentation. We note that the outcome of this experiment is determined by the relative values of $M$, $D$ and $|R|$, not the absolute values used. Larger absolute sizes could have been used, without affecting the results, at the expense of slower experimentation.

We fixed main memory size at $M = 0.1|R|$ and decreased $D$ gradually from $3|R|$ to $0.5|R|$. The size of S was 1,000 MB.

Figure 5 shows the response time of CDT-GH and CTT-GH as a function of $D$. As predicted in Section 5.3, CDT-GH performs very poorly when $D$ approaches $|R|$ as it has less and less disk space left to buffer S, increasing the number of scans of relation R. For instance, with $D = 20$ MB, CDT-GH is left with only 2 MB in which to buffer S, and relation R must be read 500 times. CTT-GH, on the other hand, has all 20 MB for buffering S and needs to read R just 50 times. As the chart shows, a tape–tape join method such as CTT-GH is a better alternative when $D \approx |R|$.

The experiment demonstrates that even if there was just enough disk space to store R and it would seem obvious to proceed as a disk–tape join, it may be better to use the disk space for other purposes, such as buffering S in larger chunks which in turn reduces the number of times R has to be scanned. The reduction in the number of R scans may well offset the extra cost of scanning R from tape instead of disk, and in situations where tape drives are faster than disks, this would indeed be a more attractive approach.

## 9 Results of Experiment 3: Large S, Small R

This experiment deals with the case where $|R|$ is roughly comparable to $M$. We examine how a variation in main memory size affects disk–tape joins. We will see that the effect of memory size on the response time of DT-NB and CDT-NB is quite analogous to the results of Experiment 2, where variations in *disk space* were shown to have a significant impact on CDT-GH and CTT-GH. In this experiment, $|S|$ was 1,000 MB while $|R|$ was chosen to be 18 MB so that we could vary $M$ freely up to $|R|$. As in Experiment 2, the outcome of the experiment depends on the relative values of $M$ and $|R|$ and not on the absolute values. Disk space was fixed at 50 MB.

When a significant fraction of R fits in memory, an interesting possibility arises. Depending on the relative speed ratio of the disk and tape drives, a disk–tape join method may be able to perform the join in only the time it takes to read S once! The bare transfer time of S from tape, which we define as the *optimum join time*, is therefore the target of the join methods we compared in this experiment. For each join method tested, we computed the relative difference between the response time and the optimum join time. We define the result as *join overhead.* A 30% overhead, for instance, means that a join takes 30% longer than simply reading S from tape would take.

To study the join methods' sensitivity to variations in the disk/tape speed ratio, we repeated

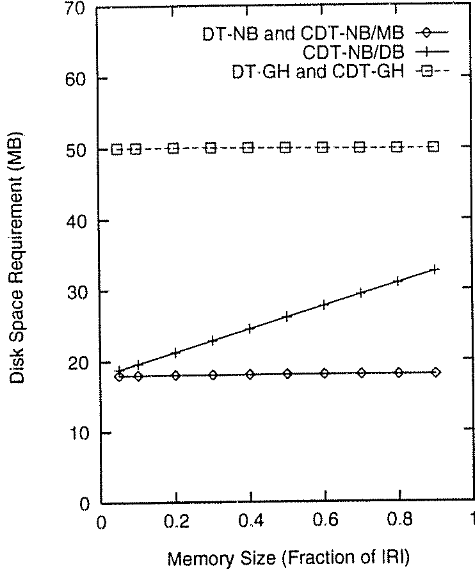


Figure 6: Disk Space Requirement.

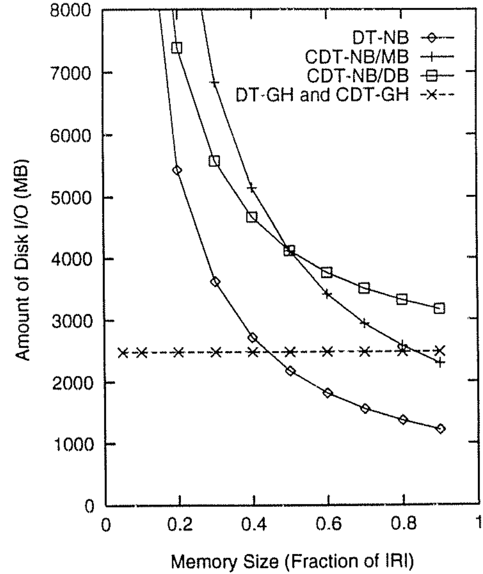


Figure 7: Disk I/O Traffic.

Experiment 3 with tape relations containing data with different compression ratios. As discussed in [14], a higher compression ratio yields a higher data transfer speed of the tape drive, and a lower compression ratio yields a lower data transfer speed. Each compression ratio also yields a different optimum join time. The relative join overhead measure normalizes the results so that runs of Experiment 3 with different tape/disk speed ratios can be compared directly. The normalized metric also makes the results independent of the tape relation size; relative join overhead expresses the performance of a join method for tape relations of arbitrary size.

Note that increasing the speed of the tape drive has the same effect on join overhead as having slower disks, and reducing the tape speed has the same effect as having faster disks.

We begin the analysis by comparing the join methods' requirements for disk space and the total amount of disk traffic. Figure 6 shows the disk space requirement of the join methods as a function of memory size. In this and subsequent charts, memory size is expressed relative to $|R|$, that is, from zero to one.

In CDT-NB/DB, the disk space requirement depends on the main memory size. The disk space requirement of DT-GH and CDT-GH is fixed but higher than that of CDT-NB/DB. DT-NB and CDT-NB/MB also have a fixed disk space requirement $|R|$.

The amount of disk traffic produced by the join methods is shown in Figure 7. DT-NB, CDT-NB/MB and CDT-NB/DB require a very large number of disk I/Os when very little main memory is available. Except for very large memory sizes, CDT-NB/MB requires the most disk I/Os because it is forced to make twice the number of iterations over relation R compared to DT-NB and CDT-NB/DB. On the other hand, CDT-NB/DB has to buffer relation S through the disks, thereby increasing its disk traffic above CDT-NB/MB in the upper memory size range.

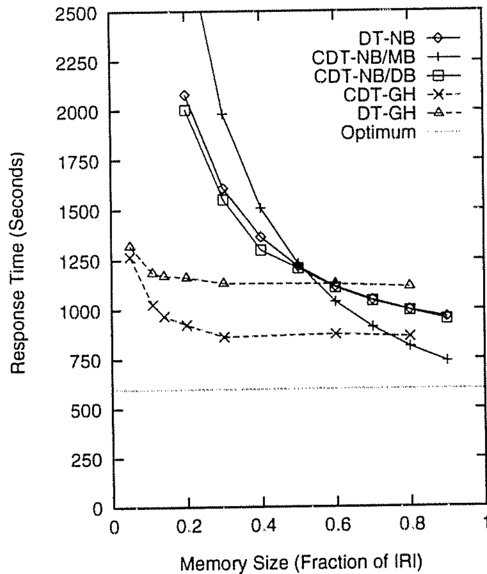


Figure 8: Response Time of Joins.

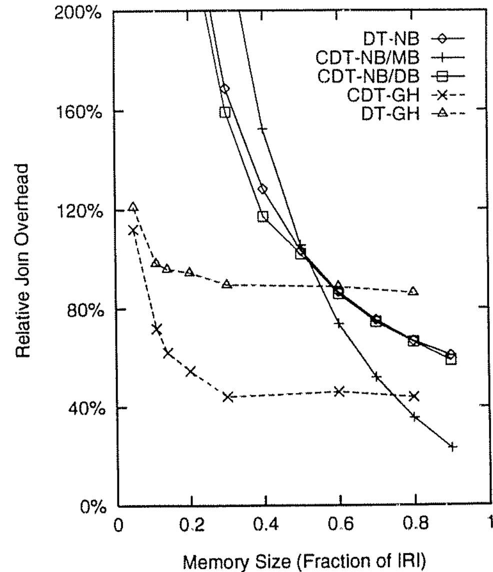


Figure 9: Relative Join Overhead.

The number of disk I/Os made by DT-GH and CDT-GH is identical and independent of memory size. With very small memory sizes these two algorithms stand out from the others. DT-GH and CDT-GH make the same number of scans over R. For both methods, the largest fraction of the disk traffic comes from buffering all of S through the disk buffers.

The two charts shown highlight the key difference between join methods based on Nested Block Join and those based on hashing: storage space can be traded in for disk traffic and vice versa. This is especially clear when very little main memory is available. Hashing-based methods reduce the amount of disk traffic by storing temporary results (hash buckets), while methods based on Nested Block Join consume less disk space, but they may have to perform far more disk I/Os to retrieve the same data over and over again. As is shown in subsequent charts, the excessive number of disk I/Os required by DT-NB, CDT-NB/MB and CDT-NB/DB has a detrimental effect on their response times.

We now move on to compare the response time of the join methods. In the first run of Experiment 3, the data on the tape was 25% compressible. Figure 8 shows the response time of the join methods as a function of memory size. The baseline shows the optimum join time for this dataset. The relative join overhead of the join methods is shown in Figure 9.

We first focus on the join methods based on Nested Block Join. The performance spectrum of DT-NB, CDT-NB/MB and CDT-NB/DB can be divided into three memory size ranges. When a large fraction of R fits in memory, CDT-NB/MB performs the best of all methods and is close to reaching the optimum join time. As the memory size decreases, the response time of CDT-NB/MB increases much more rapidly than that of DT-NB and CDT-NB/DB because it has to perform twice as many iterations over R. Therefore, in the middle memory size range, DT-NB and CDT-NB/DB

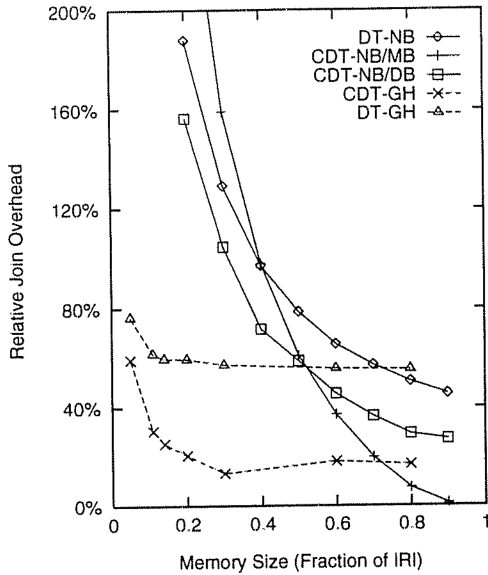


Figure 10: Relative Join Overhead (Slower Tape Drive).

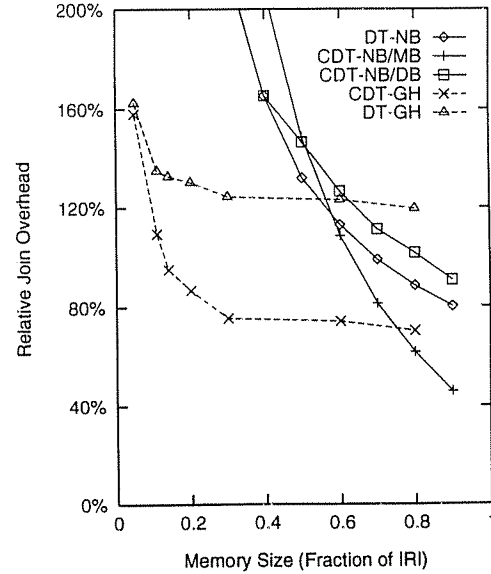


Figure 11: Relative Join Overhead (Faster Tape Drive).

provide equal or better performance compared to CDT-NB/MB. With small memory sizes, the response time of all three methods increases dramatically, making them very poor candidates for a tertiary join.

Focusing on the hashing-based algorithms next, we observe that the response times of DT-GH and CDT-GH are fairly steady except for the smallest memory sizes where writing data to buckets and reading the data back becomes more like random I/O rather than sequential I/O. In the small to medium memory size range, CDT-GH clearly dominates all other join methods. Across the memory size range, the wide margin between CDT-GH and DT-GH demonstrates the advantage of parallel I/O.

The response times of the two best-performing join methods, CDT-GH and CDT-NB/MB, cross at memory size $M = 0.7|R|$, at which point it is important to look back at the disk space and disk I/O requirements of these two methods. Recall that CDT-GH requires 50 MB of disk space whereas CDT-NB/MB consumes just 18 MB (Figure 6). The amount of disk I/O consumed by these two methods is very similar at around 3,000 MB (Figure 7). The extra disk space used by CDT-GH helps it to dominate in the small memory size range, but in the large memory size range, CDT-NB/MB is clearly the method of choice because it meets or even beats the performance of CDT-GH and requires far less disk space.

The next two charts show how the performance of the join methods changes as the disk/tape speed ratio is varied. Figure 10 shows the relative join overhead with a slower tape drive (0% compressible tape data), and Figure 11 shows the relative join overhead with a faster tape drive (50% compressible tape data).

Using a faster or slower tape drive has a slightly different effect on concurrent join methods than the sequential ones. In this experiment, concurrent join methods are bound by disk I/O, not by tape I/O, whereas the sequential methods are alternatingly bound by disk I/O and tape I/O. Changing the speed of the tape drive has no effect on the disk speeds, therefore it also has no effect on the response time of the concurrent join methods. However, increasing the tape speed reduces the optimum join time, resulting in an increased join overhead for the concurrent join methods. Similarly, lowering the tape speed increases the optimum join time and reduces the join overhead. For example, in the base case (medium tape speed shown in Figure 9), CDT-GH at its best performed the join with a 40% overhead. With a slower tape drive the overhead is reduced to just 10% and with a faster tape drive the overhead is 70%.

For the sequential join methods, an increased tape speed reduces the tape I/O portion and thus reduces the overall response time. As the tape I/O cost is reduced, the disk I/O cost becomes more dominant in the overall cost, making the sequential methods more disk I/O bound. Increasing the tape speed causes the join overhead to grow, but the increase in overhead is not as high as with the concurrent join methods. For instance, DT-NB has a minimum join overhead of 60% in the base case which reduces to 45% with a slower tape drive and increases to 80% with a faster one. The change in the overhead is significant but not as dramatic as with the concurrent join methods.

## 10 Conclusion

In this paper, we have discussed ways to compute an *ad hoc* equi-join of two relations stored on magnetic tape. We have examined join methods that require disk space at least the size of the smaller tape relation (disk–tape joins) and join methods that do not have this restriction (tape–tape joins). We have presented modifications to Grace Hash Join to make it operate on tape-resident relations, and also reviewed Nested Block Join-based methods developed in our earlier work [13]. An experimental implementation and performance analysis with relation sizes ranging up to 10,000 MB allowed us to validate the strengths and weaknesses of these methods.

The tape join methods we have examined present a range of alternatives in terms of their resource requirements (memory, disk and tape space), execution time and implementation complexity. Some of the methods exploit parallelism between disk and tape I/Os to reduce response time. Some methods also stripe data on multiple disks to balance the consumption of disk bandwidth and storage space.

Of the join methods analyzed in this paper, Concurrent Tape–Tape Grace Hash Join is the sole candidate for very large tape joins as it requires very little main memory and disk space. The join method delivers quite acceptable performance despite being tape I/O intensive and scales up gracefully when relation sizes increase.

When ample disk space but little main memory is available, Concurrent Disk–Tape Grace Hash Join is the preferred join method, computing the join in less time than the tape–tape variant. At the other end of the relation size spectrum, Concurrent Disk–Tape Nested Block Join yields very good performance when a large fraction of the smaller relation fits in memory, making it possible to effectively overlap disk and tape I/Os.

## References


[1] ANSI X3T9.2 Committee. Small Computer System Interface 2 (SCSI-2), Nov. 1993. Working Draft, Project 375D.

[2] M. Blasgen and K. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):363–377, 1977.

[3] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. Conf. Very Large Databases*, pages 323–333, Singapore, Aug. 1984.

[4] M. J. Carey, L. M. Haas, and M. Livny. Tapes hold data too: Challenges of tuples on tertiary store. In *Proc. ACM SIGMOD*, pages 413–417, Washington, D.C., May 1993.

[5] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD*, pages 1–8, Boston, MA, June 1984.

[6] S. Ghandeharizadeh, A. Dashti, and C. Shahabi. Pipelining mechanism to minimize the latency time in hierarchical multimedia storage managers. *Computer Communications*, 18(3):170–184, Mar. 1995. Also available as Technical Report 94-584, Computer Science Department, University of Southern California.

[7] L. M. Haas, M. J. Carey, and M. Livny. SEEKing the truth about *ad hoc* join costs. Technical Report 1148, Department of Computer Science, University of Wisconsin at Madison, May 1993.

[8] R. B. Hagmann. An observation on database buffering performance metrics. In *Proc. Conf. Very Large Databases*, pages 289–293, Kyoto, Japan, Aug. 1986.

[9] B. K. Hillyer and A. Silberschatz. Random I/O scheduling in online tertiary storage systems. In *Proc. ACM SIGMOD*, Montreal, Canada, June 1996.

[10] M. G. Kienzle, A. Dan, D. Sitaram, and W. Tetzlaff. Using tertiary storage in video-on-demand servers. In *Proc. CompCon*, pages 225–233, San Francisco, CA, Feb. 1995.

[11] W. Kim. A new way to compute the product and join of relation. In *Proc. ACM SIGMOD*, pages 179–187, Santa Monica, CA, May 1980.

[12] D. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching.* Addison-Wesley Publishing Co., Redwood City, CA, 1973.

[13] J. Myllymaki and M. Livny. Disk–tape joins: Synchronizing disk and tape access. In *Proc. ACM SIGMETRICS*, pages 279–290, Ottawa, Canada, May 1995.

[14] J. Myllymaki and M. Livny. Efficient buffering for concurrent disk and tape I/O. In *Proc. Performance '96*, pages 453–471, Lausanne, Switzerland, Oct. 1996.

[15] S. Sarawagi. Database systems for efficient access to tertiary memory. In *Proc. IEEE Symposium on Mass Storage Systems*, pages 120–126, Monterey, CA, Sept. 1995.

[16] S. Sarawagi. Query processing in tertiary memory databases. In *Proc. Conf. Very Large Databases*, Zurich, Switzerland, Sept. 1995.

[17] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, Sept. 1986.

[18] H. Suzuki et al. Storage hierarchy for video-on-demand systems. In *Proc. Storage and Retrieval for Image and Video Databases II*, pages 198–207, San Jose, CA, Feb. 1994.

[19] J.-B. Yu and D. J. DeWitt. Query pre-execution and batching in Paradise: A two-pronged approach to the efficient processing of queries on tape-resident data sets. Document available at http://www.cs.wisc.edu/~jiebing/tape.ps.