Single Query Optimization for Tertiary Memory*

Sunita Sarawagi   Michael Stonebraker
Computer Science Division
University of California at Berkeley

Abstract

We present query execution strategies that are optimized for the characteristics of tertiary memory devices. Traditional query execution methods are oriented to magnetic disk or main memory and perform poorly on tertiary memory. Our methods use ordering and batching techniques on the I/O requests to reduce the media switch cost and seek cost on these devices. Some of our methods are provably optimal and others are shown to be superior by simulation and cost formula analysis.

1 Introduction

Large capacity storage systems are essential for an increasing number of scientific and commercial applications. For example, research on global change effects requires the storage and analysis of massive amounts of satellite data [STO91]. The Earth Observation System (EOS) [DOZ91] alone is expected to provide one terabyte per day of raw data to global change scientists. Such volumes of data require high capacity tertiary memory devices [KAT91] for storage and smart data base systems for efficient query support.

Traditional DBMSs have assumed that all data reside on magnetic disk or main memory. Therefore all optimization decisions were oriented towards this technology. Tertiary memory, if used at all, functioned only as an archival storage system to be written once and rarely read. With the inclusion of tertiary memory as an active part of the storage hierarchy it is necessary to rethink the optimization decisions made by a DBMS [STO91a] [CAR93]. In this paper, we propose improvements to existing query execution strategies to adapt them to tertiary storage devices.

Tertiary memory devices have very different performance characteristics than magnetic disks. A typical device consists of a large number of storage *media* (tapes or disk platters) and a few read-write drives. A robot arm switches the media between the shelves and drives, typically in 5–10 seconds. We use the term *extent* to refer to the unit of data transfer from the tertiary memory. An extent is generally much bigger than a magnetic disk page because the high

*This research was sponsored by the National Science Foundation under grant IRI-9107455, the Defense Advanced Research Projects Agency under grant T63-92-C-0007, and the Army Research Office under grant 91-G-0183. Additional support was provided by the University of California and Digital Equipment Corporation under Research Grant 1243. Other industrial and government partners include the State of California Department of Water Resources, United States Geological Survey, Construction Engineering Research Laboratory (CERL) of the U.S. Army Corps of Engineers, the National Aeronautics and Space Administration (NASA), Epoch Systems, Inc., Hewlett-Packard Corp., Hughes Aircraft Company, MCI, Metrum Corporation, PictureTel Corporation, Research Systems Inc., Science Applications International Corporation, Siemens Inc., and TRW Space and Electronics.

Storage device	Exchange time (sec)	Average seek time (sec)	Data transfer rate (KB/sec)	Extent (128 KB) transfer time (sec)	Worst/best access (sec)
Optical disk	8	0.1	500	.256	32.4
Helical scan tape	6	45	4000	.032	4406
Optical tape	>60	30	3000	.043	3488
Magnetic disk	-	.03	4250	.030	4

Table 1: Comparative study of the characteristics of different storage devices

latency of tertiary memory is amortized by transferring data in larger quantities. In Table 1 we summarize the storage characteristics of several tertiary memory devices. The characteristics shown are exchange time (time to unload one medium from the drive and then load a new unit and get it ready for reading), average seek time (one third of the maximum seek time), data transfer rate, transfer time for an extent of size 128 KB, and the ratio between the worst case and best case access times for an extent.

From the last column of Table 1 one can note that magnetic disks are a relatively uniform storage medium. Worst case access times are only a factor of 4 larger than best case times. However, tertiary memory devices have much higher variability. In fact, tape oriented devices have 3 orders of magnitude more variation. This variability makes it crucial to carefully optimize the order in which data blocks are accessed. Furthermore, since a typical tertiary memory device has very high latency it can perform fewer I/Os per unit time than a magnetic disk. The greater speed mismatch between I/O and CPU means that the query optimizer can afford to run more expensive query optimization algorithms since the savings will more than compensate for the additional optimization.

Based on the above differences we have developed methods of executing queries that are tuned to the tertiary memory characteristics. The main drive of our optimization methods is to preprocess and plan I/O requests so that data is fetched in the order in which it is stored on the tertiary memory instead of in a random order. To convert random I/Os to ordered I/Os, one technique we use is, first find out all data to be fetched in the form of tuple identifiers (TID) (using sequential scans or index structures whenever possible), and then use this list of TIDs to read tuples wisely. We suggest methods of doing I/O in batches that utilize the cache effectively to amortize the high latency of tertiary memory devices. In this paper, we will discuss how these ideas can be applied for executing single relation queries and two-way joins.

The paper is organized as follows. In Section 2 we discuss methods of optimizing index scans on single relation queries. In Section 3 we consider methods of optimizing two way joins for relations that are present on a single medium. We suggest methods for improving the traditional nested loop join, sort merge join and hash join. Finally, we suggest future work and make concluding remarks.

1.1 Notations and Assumptions

We assume that all relations are stored on *tape-based* or *disk-based* tertiary memory and that the optimizer knows how a relation is laid out on this memory. In particular, it knows the media on which each part of a relation is stored and the locations of the extents of a relation if the storage medium is tape-based. Indices for relations are assumed to be stored on magnetic disk or on some medium where the cost of accessing the index is small compared to the cost of accessing the relation. A magnetic disk is used as the cache for tertiary memory. The size of the cache

TID	tuple identifier
M	available cache
E_R	number of extents in relation R
T_R	number of tuples in relation R
E_M	number of extents that can fit in M
T_M	number of tuples of a relation that can fit in M
s	tuple selectivity of a join operation
e	extent selectivity of a join operation

Table 2: Notations

is usually small compared to the capacity of tertiary memory. For example, the Sequoia 2000 project [STO91b] has a 14 gigabyte magnetic disk cache attached to 9 terabyte tape robot. As such, cache size is 0.16% of storage size and few cache "hits" can be expected. Therefore, the optimizer assumes that all data has to be fetched from tertiary memory, and no data is already present in the cache. The optimizer has a good estimate of the amount of free space available in the cache and can control what extents should be kept in the cache and what extents should be evicted. In this paper we will concentrate on optimizing I/O time for tertiary memory. We ignore CPU time and I/O time for magnetic disk because the delay of tertiary memory is so high that we expect it to be the dominant component. Other notations used are summarized in Table 2.

2 Single Relation Queries

The conventional method of executing an index scan on a relation is to search the index tree, fetching the relevant tuples in the order in which their TIDs occur in the index leaf pages. This method is inefficient for tertiary memory because it might fetch an extent for every tuple read and might spend considerable time seeking between them if the order of the extents is random. We propose a modification called the *deferred index scan* method as follows:

We first scan the index and get the list of TIDs of qualifying tuples. The TID list is then ordered so that the following conditions will hold if the TIDs are processed in that order:

- all extents on one medium are fetched before fetching extents on another medium.
- if the storage medium is tape all required extents on the tape are fetched by one forward sweep across the tape rather than seeking back and forth.
- when an extent is fetched all qualifying tuples in that extent are read before fetching the next extent.

The deferred index scan method fetches the minimum number of extents and incurs the least I/O cost. So, according to our proposed cost model of considering only I/O cost, this scheme is obviously optimal.

3 Two-way Joins on Single Medium Relations

In this section we propose modifications to the traditional methods of doing two way joins for efficient execution on tertiary memory when each relation is confined to a single medium.

In traditional DBMSs the disk I/O cost is measured only in terms of the number of pages fetched. This is inadequate for tertiary memory since media switches and seek distance account for a significant fraction of the I/O cost. Hence, in addition to the **number of extents fetched** we use the **number of switches between the two component relations** as a cost metric. The latter quantity refers to the number of times a request changes from one relation to the other. By multiplying this number with the total time of switching from one relation to the other we can measure the seek time involved in moving between the two relations and also the number of media switches since we have assumed that a relation is confined to a single medium. For deriving the cost formulas for different join methods, we assume that the tertiary memory has only one read-write drive, the size of the disk cache is small compared to the size of the relations, the buffer management policy used during query processing for traditional join methods is LRU, and the join attribute is *unclustered*.

3.1 Nested Loop Method

This method, also called the iterative substitution method, computes the join of two relations by scanning the outer relation (R) and for each tuple of R scanning the inner relation (S) to get the matching tuples. We propose three improved techniques based on the method used for scanning R and S:

1. Fragmented Nested Loop: applies when the join attribute of neither R nor S is indexed.
2. Alternating Nested Loop-1: applies when the join attribute of one of R or S is indexed.
3. Alternating Nested Loop-2: applies when the join attributes of both R and S are indexed.

3.1.1 Fragmented Nested Loop

When there are no indices on the join attribute for either the inner relation or the outer relation the traditional method of executing the join is to sequentially scan the inner relation for each tuple of the outer relation. We propose a modification to this method that requires fewer I/Os:

The cache can hold only E_M extents at a time. Divide the smaller relation (say R) into fragments of size E_M. We will get $n = E_R/E_M$ fragments of R (call them $R_1, R_2 \ldots R_n$). The join will be done in n passes. In the ith pass, cache all extents in fragment R_i and do standard nested loop between R_i and S choosing S as the outer relation. During this join, tuples of S are fetched sequentially and for such tuple of S, a pass is made over R_i that remains in cache.
Cost Analysis
Assume neither R nor S fits in cache. For the sake of comparison assume R is the outer relation in the original nested loop method.
Original nested loop: For each tuple of R we need to fetch all extents of S. So the total number of extents fetched is $T_R E_S + E_R$. The tuples of R are fetched sequentially but once an extent of R is fetched it remains in cache until all tuples in it have formed joins. Therefore, a switch is made from R to S and back to R for every extent of R except the last for which we do not need to switch back to R. So, the total number of switches between the two relations is $2E_R - 1$.
Fragmented nested loop: For each fragment of R, one sequential scan is made on S. Also, an extent of R is fetched only once from tertiary memory. So, the total number of extents fetched is $E_R + nE_S$. For each fragment of R, data requests to tertiary memory change from R to S and then from S back to R. So, the number of switches is 2 for every fragment except the last for which we do not need to switch from S back to R. This adds up to a total of $2n - 1$ switches.
The estimated cost in terms of the number of extents fetched and the number of switches is summarized next.

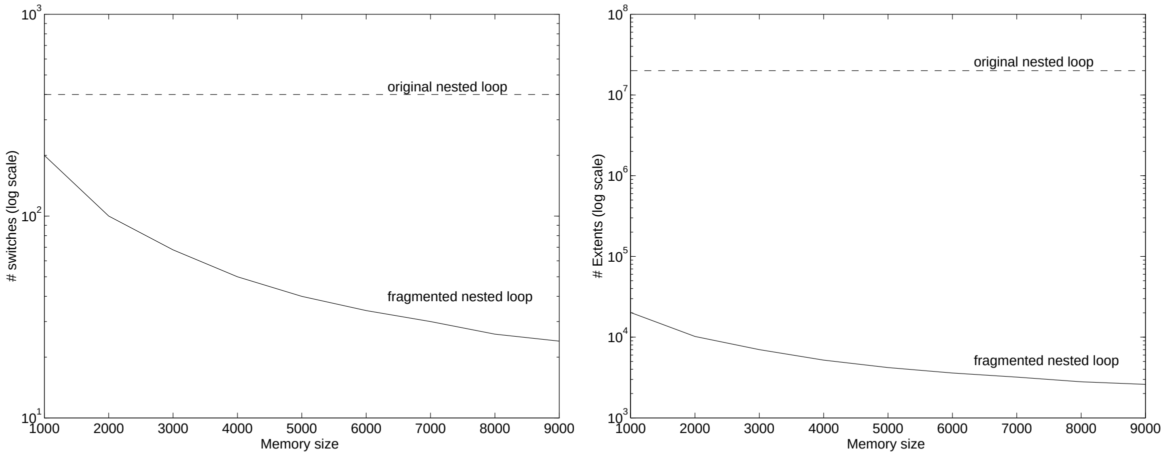

Figure 1: I/O costs as a function of memory size

	fragmented nested loop	original nested loop
# of switches between R & S:	$2n-1$	$2E_R-1$
# of extents fetched:	E_R+nE_S	$T_RE_S+E_R$

The above cost formulas are plotted in Figure 1. The broken line corresponds to the traditional method and the solid line to the modified method. The parameters used for the plots are given in Table 3. The time taken by the modified method decreases when the available cache size increases and levels out when all of the smaller relation fits in the cache.

Claim: *With the above method the number of extents fetched and the number of switches is the minimum possible for executing a nested loop join.*

Proof. Assume S consists of m partitions of size E_M and R consists of n partitions ($n \leq m$). To complete the nested loop, each partition of R has to join with m other partitions and each partition of S has to join with n other partitions. There must be a total of nm joins between partitions. The cache can hold only 1 partition at a time. For every partition in the cache, a join with the partition will require fetching one other partition. But a cached partition can participate in at most m other joins. Hence for every m joins between partitions a minimum of $m+1$ partitions have to fetched. So the number of partitions to be fetched for nm joins must be at least $n(m+1) = n+nm$ and since each partition consists of E_M extents, the number of extents fetched must be at least $nE_M + n(mE_M) = E_R + nE_S$. Similarly, it can be argued that a minimum of 2 switches is required for every m joins between partitions. However, for the last partition one switch is enough because it is not necessary to switch back to R for fetching the next partition. So, for nm joins between partitions the number of switches has to be at least $2n-1$. Hence, the modified method achieves the lower bound and is therefore optimal.

3.1.2 Alternating Nest Loop-1

This method can be applied when there is an index on the join attribute of one of the two relations. Let us assume the indexed relation is S and the inner relation used for the join is S. The traditional method is to sequentially scan the outer relation R, and for each of its tuples do an index scan on S and fetch the qualifying tuples (if any). Hence, for each extent of R we need to switch to S, complete the join between all tuples in the extent and S, then switch back to R to fetch the next extent. We propose a modification to this method that avoids the random I/O

t	tuple size	200 bytes
E	extent size	100 KB
T_R	number of tuples in R	100000 tuples
T_S	number of tuples in S	100000 tuples
T_M	number of tuples that can fit in the cache	1000 tuples – 10000 tuples
s	selectivity of join	0.2

Table 3: Graph Parameters

on S and the frequent switches between the two relations by ordering the I/O requests using the TID list obtained from the index on S and a sequential scan on R.

We proceed in two phases. In the first phase, called the *initialization phase*, we find the list of TIDs of tuples of R and S that participate in a join using the index of S and a sequential scan on R. In the second phase, called the *join build phase*, the TID list is used to fetch relevant extents and process the join between R and S. During the join phase, the tuples are read in a manner that reduces the number of times an extent is fetched, and the number of times a switch is made. Details of the algorithm follow:

Initialization phase:
To get the TID list, sequentially scan R, and for each tuple of R search the index of S. This will tell if the R tuple participates in a join and the TIDs of the S tuples it joins with (if any).

Join build phase:
We use the term *qualifying* tuples for tuples that participate in the join. The TID list is used to get the TIDs of qualifying tuples. Here are the steps in this phase:

(1) Cache as many qualifying tuples of R as possible. There can be at most T_M of them. Call these cached tuples R'. This step can be combined with the initialization phase above to avoid fetching the tuples of R' again from tertiary memory.

(2) Next switch to S and process the join between R' and S, caching as many qualifying tuples of S as possible in the space freed by tuples of R' once they are done with their joins. We suggest the following method of doing this task efficiently.

    For the tuples in R', consult the TID list data structure to get the TIDs of S tuples it joins with. Fetch the extents that hold these S tuples in the order prescribed in Section 2. When an extent E of relation S is fetched, process the join for all tuples of E with R'. Tuples of S and R' that finish their joins are removed from the qualifying TID list and tuples of R' that are done with their joins are removed from the cache. Before E is evicted from the cache, all qualifying tuples in E that are not done with their joins are cached if space is available.

    At the end of this operation, all tuples in R' have been removed from the cache since they are finished with their joins. Some qualifying tuples of S that were cached in the above step are still in the cache. If space is available, cache as many more qualifying tuples of S as possible. Call the set of S tuples in the cache S'. These are the tuples of S that need to form joins with R. Switch to relation R and do the join between S' and R repeating the process described above.

(3) This process of caching tuples from one relation, switching to the other relation, and doing

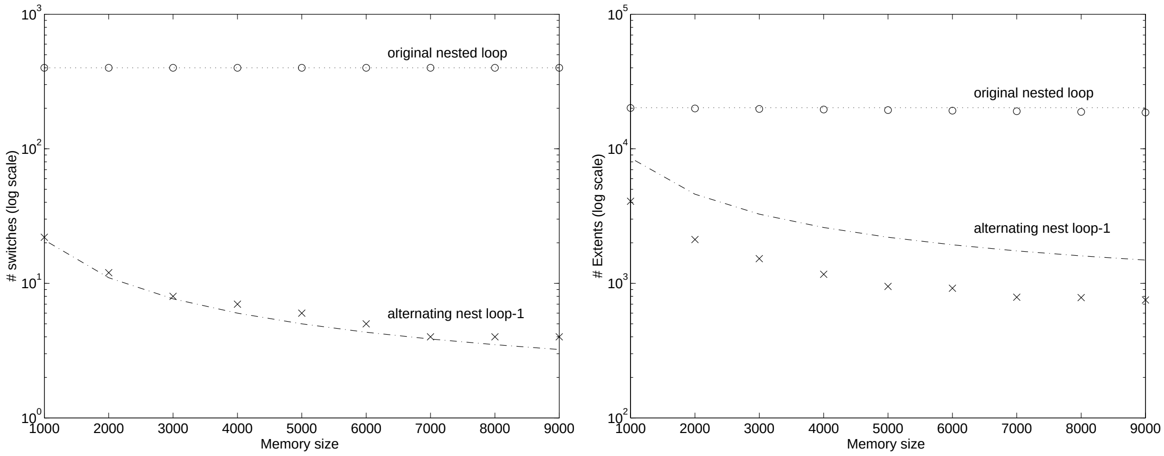

Figure 2: I/O cost as a function of memory size

the joins in batches is repeated until the TID list becomes empty.

Cost Analysis

The cost formula of the above method as compared to the original method follows. For this analysis we assume a tuple of R joins with only one tuple of S and vice versa. Let X_R be the expected number of extents of R over which T_M tuples are spread and X_S is the corresponding value for S.

	alternating nested loop-1	original nested loop
# of switches between R & S:	$n = sT_R/T_M$	$2eE_R$
# of extents fetched:	$E_R + n(X_R + X_S)$	$sT_S + E_R$

Original nested loop: The number of extents of R and S fetched is E_R (since R is the outer relation and is scanned sequentially) and sT_S (since S is the inner relation and the sT_S tuples are fetched randomly) respectively. The number of switches is $2eE_R$ since for each extent of R that contains any qualifying tuple, we need to switch to S and back again to R.

Alternating nested loop-1: Define a *pass* of the algorithm as the interval between two successive switches from R to S. In each pass $2T_M$ tuples of R can be joined. Hence the total number of passes required to join all (sT_R) tuples of R is $sT_R/(2T_M)$. We make two switches in each pass, so the total number of switches is equal to sT_R/T_M. For the initial TID list computation we fetch E_R extents of R. For each pass of the algorithm, we fetch $2(X_R + X_S)$ extents on an average as explained next. To process the join with the T_M tuples of R that are cached, we fetch T_M tuples of S. These T_M tuples of S require fetching of X_S extents because an extent that contains any of those T_M tuples is fetched only once. Next, to cache T_M tuples of S (the set we called S') we fetch another X_S extents of S. For ease of analysis, we assume no tuples of S' are cached during the join with cached R. Similarly after switching to R we need to fetch $2X_R$ extents of R. Hence the total number of extents fetched during the join build phase is the number of passes ($= \frac{n}{2}$) times the total number of extents fetched per pass ($= 2(X_R + X_S)$) which is equal to $n(X_R + X_S)$. An approximate value for X_R is $E_R(1 - (1 - \frac{1}{E_R})^{T_P})$.

Figure 2 shows the number of extents fetched and the number of switches for the two different join methods. The two cost metrices are plotted (on a log scale) against varying cache size. The broken lines represent the estimated cost formulas tabulated earlier and the points show the values obtained from a simulation of the two join methods for the parameters in Table 3. The observed number of switches between the two relations agrees very closely with the predicted cost formula. The number of extents fetched for the modified method is overestimated slightly by the cost formula because these are derived under the assumption that no caching is done during the join processing stage. In reality, since the caching and the join stages are combined, fewer extents are fetched in the caching stage. Even so, the improvement achieved by the modified method is significant. For instance, for a cache size of 5000 tuples, our modification results in the number of switches dropping from 400 to 6 and the number of extents fetched dropping from 19,500 to 750. The relative performance of the modified method improves when the available cache increases because cache utilization is better than in the traditional method.

3.1.3 Alternating Nest Loop-2

This method applies when there is an index on the join attribute of both R and S. An index scan on the outer relation is better than a sequential scan only when there is a restriction on the join field of R. In the traditional scheme, the index for R is scanned to get a qualified tuple of R. This tuple is used to do an index scan of S, and if there is a match the matching tuples of S are fetched to complete the join. This means that, in the worst case, every join tuple requires a switch between R and S and the fetching of at least two extents. A more efficient method is now suggested.

This method is the same as the previous case (Section 3.1.2) except that the method used to get the initial qualifying TID list is different. Instead of sequentially scanning tuples of R to get the list of qualified TIDs we use the indices of R and S. The leaf nodes of the index trees store the (*key*, TID) pair for the join attribute. The index tree can then be used to create a *sorted* list of the (*key*, TID) pair for each relation. These lists are joined using sort merge on the *key* field to get a list of qualifying tuples as above.

Cost Analysis

The cost formula of the above method as compared to the original method is as follows:

	alternating nested loop-2	original nested loop
# of switches between R & S:	$n = sT_R/T_M$	$2sT_R$
# of extents fetched:	$n(X_R + X_S)$	$sT_S + T_R$

Original nested loop: The number of switches between R and S is $2sT_R$, since for each tuple of R that participates in a join, it might be necessary to fetch a new extent of S and then switch back to R to fetch the next tuple of R. Since tuples of R are not accessed sequentially, for each tuple of R it might be necessary to fetch an extent from tertiary memory. So, the number of extents of R fetched is T_R. The number of extents of S fetched is sT_S since only qualifying tuples of S are fetched.

Alternating nested loop-2: The number of switches is the same as in alternating nested loop-1. The number of extents fetched differs only in the extra E_R extents fetched during initial TID list formation.

The graph in Figure 3 shows a comparison of the number of extents fetched and the number of switches for the two methods. The parameter settings are the same as in the previous case. From the graph it is clear that the original nested loop method requires almost a thousand times more switches than the modified method. For instance, for cache size of 5000 tuples, the number

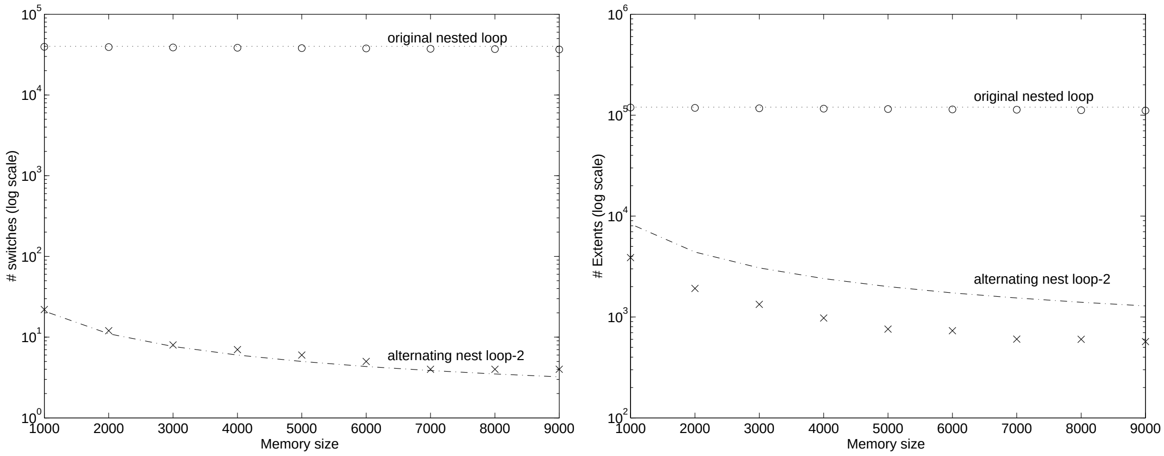

Figure 3: I/O cost as a function of memory size

of switches drops from 40,000 to 10 implying a 55 hour difference in execution time assuming a media switch cost of 5 seconds.

TID List Organization

The success of both of the above methods depends on the efficient manipulation of the TID list structure to do the following operations:

- Given the TID of a tuple of R, retrieve the TID of all tuples of S it joins with and vice versa.
- Find all qualifying tuples in a given extent.
- Remove a TID from the TID list.

All these operations can be done relatively cheaply when the TID structure is in main memory. This will be true when the number of qualifying join tuples is small. When the number of entries in the qualifying TID list gets large it is necessary to design structures that scale well with the list length. We suggest the following scalable method for handling large TID lists.

Let N denote the number of TIDs of R that can fit in the main memory TID data structure. Instead of constructing the initial TID list for the entire relation, R is divided into subparts $R_1, R_2, \ldots R_m$ such that each partition except possibly the last one contains N qualifying tuples. For each partition R_i, we use the methods described earlier for joining tuples in R_i with the corresponding tuples in S. The partitions of R can be easily determined by modifying the initial TID list formation method as follows:

If the initial method of constructing a TID list involves a sequential scan of R, we simply need to remember the ending point of the previous partition and start forming the new TID list from that tuple. When N TIDs are inserted in the list we stop. If the initial method of constructing the TID list involves an index tree search, we need to remember the index key of the last tuple in the previous partition and start the new TID list formation from that key.

3.2 Two-level Hash Join

If the smaller relation, R, fits in the disk cache, any conventional hash join method [SHA86] will work well for tertiary memory databases. Otherwise, the conventional methods of partitioning the two relations into buckets that fit in the cache and doing the join between the partitioned buckets can be inefficient for some tertiary memory devices. In the partitioning phase, this method requires reading from a source file and writing N hash partitions. On a tape, this might require random seeks between $N+1$ different data clusters during the partitioning phase. Also, some tertiary memory devices are write-once and hence writing out temporary partitions may not be feasible. We propose a variation of the hybrid hash join that is adopted for the tertiary memory environment.

The main steps of the algorithm are:

Partitioning phase:

- Scan R and form a hash table organized as follows. Store only the TIDs of the hashed tuples for each of the buckets except the first n. For the first n buckets (call them B_0) store the entire tuples. The value of n is chosen so that all tuples of B_0 can fit in the cache. Unlike the traditional method we do not require all tuples of B_0 to fit in main memory – they can spill over to the disk cache.
- Switch to S and partition tuples of S the same way. For tuples hashing into the first n buckets, directly process the join.
- Flush buckets that contain TIDs of only one of the component relations since they do not participate in any join. All the tuples in buckets of B_0 are finished with their joins and can be evicted.

Join build phase:

At the end of the partitioning phase we have a list of the TIDs of the tuples of R and S in each bucket. A multi-pass algorithm is used to form joins between tuples in these buckets. A pass of this algorithm consists of the following steps:

- Estimate the next x buckets (call them B_i) that can cache tuples of S by counting the TIDs of S hashed into each bucket. The TID list of the S tuples in B_i will yield a list of S tuples (S') to be fetched. The TID list of S' can be ordered as described in Section 2 to read the tuples of S' efficiently from tertiary memory. With these tuples build a hash table for S'.
- Switch to R and process the join between tuples in the bucket set B_i. The tuples of R required for processing this join are fetched in the ordered prescribed in Section 2.
- Build a hash table for the next y buckets that can hold tuples of R, switch to S, and process the join using the hash table as described in the above two steps but with the roles of R and S interchanged.

For joining tuples within a bucket we suggest using *classic hash* [SHA86] because it requires only one scan for tuples of the outer relation and hence does not require any cache space for the probing relation. The main argument against using the classic hash method is that it can cause too many page faults and hence too many reads on the magnetic disk cache if the hash table does not fit in main memory. However, this is acceptable to us since we save on the more expensive reads to tertiary memory. If we restrict our attention to hash tables that fit in main memory, fewer tuples can be joined in each pass, causing the number of switches to increase. Also, in our method of reading the tuples of a relation in a pass, we order the reads so that an extent is fetched at most twice in each pass of the algorithm. So, having a smaller number of passes also means that the number of extents fetched is smaller.

Cost Analysis

Assume for ease of explanation that all partitions have the same number of tuples (T_P) of R

and S. Let N be the number of partitions where $N = T_R/T_P$. Let X_R be the expected number of extents of R over which T_P tuples are spread.

	modified hybrid hash	original hybrid hash
# of switches between R & S:	N	$2N$
# of extents read/written:	$2X_R(N-1)+E_R+E_S$	$3(E_R+E_S)-4E_P$

Original hybrid hash: The number of switches is $2N$ because joins within each partition require 2 switches – one from R to S and a second from S back to R. The accounting for the number of extents is as follows:

$E_R + E_S$: for the initial partitioning phase
$E_R + E_S - 2E_P$: for writing out all but the first partitions for both relations
$E_R + E_S - 2E_P$: for reading the partitions in the subsequent join phases.

Modified hybrid hash: A pass of the algorithm completes the join of two partitions, one with R as the probing relation, and the other with S as the probing relation and requires two switches – once from R to S and then from S back again to R. The total number of switches is $2(N/2) = N$. The number of extents read in the initial partitioning phase is $E_S + E_R$. In the subsequent join phase, for each partition except the first one, T_P tuples are fetched from each of R and S. If X_R is the number of extents over which T_P tuples are spread, the number of extents fetched per partition is $2X_R$. Hence the total number of extents fetched in the join phase is $2X_R(N-1)$.

The superiority of the modified method over the original method is difficult to decide by just examining the cost formulas. Also, it is difficult to analyze the seek overhead of the partitioning phase in the original method. So, we will simply list some of the advantages of the modified method over the original method.

- There is no need to write the partitions into tertiary memory. This is useful for both
  - write once tertiary memory and
  - tape-based tertiary memory (since writing out N buckets during hashing can result in many seeks between the relation partitions).
- The TID list allows one to estimate the number of tuples in each bucket. This enables one to cache a variable number of buckets during each pass and thus utilize the available buffer space effectively. This pays off when the hash function is highly skewed.
- Knowing the TID list one can fetch the qualifying tuples in one pass in a manner that makes accesses efficient. This also means that subsequent passes over the relations will not require fetching the entire relation but only the extents that contain tuples for the selected sets of buckets.

3.3 Fragmented Merge Sort Join

If both R and S are sorted on the join attribute, the sort merge method pays off for joins. Again, the conventional method of fetching a tuple of one relation, switching to the other, and scanning it for a match can be very inefficient and we suggest the following modification. Cache as many tuples of the smaller relation, R, as fit in the cache, switch to S, and do the join between cached tuples of R and S. Whenever a tuple is passed over in the sort-merge operation it is evicted from the cache. Starting from the first unpassed tuple, cache as many tuples of S as possible, switch to R and continue as before.

The comparative cost estimates of the two methods is given next:

	fragmented merge sort	original merge sort
# of switches between R & S:	$2E_R/E_M$	$2E_R$
# of extents fetched:	$E_R + E_S$	$E_R + E_S$

For the traditional method, the number of switches is $2E_R$ because for each extent of R we need to switch once from R to S and then back again to R to get the next extent. For the modified method the number of switches is $2E_R/E_M$ since for every E_M extents of R we might need two switches in the worst case.

4 Conclusions and Future Work

In this paper we have presented methods of executing single relation queries and two way joins on tertiary memory. We extended the cost model of traditional query optimizers to include the cost components of both tape-based and disk-based tertiary memory devices. However, just changing the weighting factor on the cost formula is not sufficient for adapting traditional methods to tertiary memory and we need to change the execution methods to do I/O more carefully. Some of the main ideas of our methods are:

- avoid fetching extents randomly – fetch data in batches filling the cache as much as possible to amortize latency of tertiary memory.
- convert random I/Os to ordered I/Os by preprocessing the I/O requests using TID lists (obtained from relation indices or sequential scans)

We compared our methods against the traditional methods using analytical models and simulations and showed why traditional methods fare poorly on tertiary memory. Our modifications achieve 10 to 1000 fold reduction in I/O cost for a synthetic workload.

A number of issues are still unexplored for query optimization on tertiary memory. First we intend to extend the two way join methods for relations spread over more than one medium. Next, we need to devise methods for doing general n-way joins. Since I/O is expensive, it might pay to do multiple query optimization so that multiple queries can share access to one relation. Finally, we wish to develop methods of doing query scheduling on tertiary memory databases.

References

[CAR93] M.J. Carey, L.M. Haas, and M. Livny. Tapes hold data, too: challenges of tuples on tertiary store. *SIGMOD Record*, 22(2):413–417, 1993.

[DOZ91] J. Dozier and H.K. Ramapriyan. Planning for the EOS data and information system. In *Global Environment Change*, volume 1. Springer-Verlag, Berlin, 1991.

[KAT91] R.H. Katz et al. Robo-line storage: High capacity storage systems over geographically distributed networks. Sequoia 2000 Technical Report 91/3, University of California at Berkeley, 1991.

[STO91] M. Stonebraker and J. Dozier. Large capacity object servers to support global change research. Sequoia 2000 Technical Report 91/1, University of California at Berkeley, 1991.

[SHA86] L.D. Shapiro. Join processing in database systems with large main memory. *ACM Transactions on database systems*, 11(3):239–264, 1986.

[STO91a] M. Stonebraker. Managing persistent objects in a multi-level store. *SIGMOD Record*, 20(2):2–11, 1991.

[STO91b] M. Stonebraker. An overview of the Sequoia 2000 project. Sequoia 2000 Technical Report 91/5, University of California at Berkeley, 1991.