Shared Index Scans For Data Warehouses

Yannis Kotidis[1]*, Yannis Sismanis[2], and Nick Roussopoulos[2]

[1] AT&T Labs–Research, 180 Park Ave, P.O. Box 971 Florham Park, NJ 07932 USA
kotidis@research.att.com
[2] University of Maryland, College Park
{isis,nick}@cs.umd.edu

Abstract. In this paper we propose a new "transcurrent execution model" (TEM) for concurrent user queries against tree indexes. Our model exploits intra-parallelism of the index scan and dynamically decomposes each query into a set of disjoint "query patches". TEM integrates the ideas of prefetching and shared scans in a new framework, suitable for dynamic multi-user environments. It supports time constraints in the scheduling of these patches and introduces the notion of *data flow* for achieving a steady progress of all queries. Our experiments demonstrate that the transcurrent query execution results in high locality of I/O which in turn translates to performance benefits in terms of query execution time, buffer hit ratio and disk throughput. These benefits increase as the workload in the warehouse increases and offer a scalable solution to the I/O problem of data warehouses.

1 Introduction

Tree-based indexes like B-trees, B^+-trees, bitmap indexes [16, 2] and variations of R-trees [19, 12] are popular in data warehousing environments for storing and/or indexing massive datasets. In a multiuser environment accessing these indices has the potential of becoming a significant performance bottleneck. This is because in an unsynchronized execution model, concurrent queries are "competing" for the shared system resources like memory buffers and disk bandwidth while accessing the trees. Scheduling of disk requests and prefetching are well-studied techniques [8, 23, 1] exploited by all modern disk controllers to maximize the performance of the disk sub-system. However, optimizing the execution of the I/O at the physical disk level does not always realize the potential performance benefits. Even-though most commercial systems perform asynchronous I/O, from the buffer's perspective the interaction with a query is a synchronous one: the query thread asks for a page, waits till the buffer manager satisfies the request and then resumes execution. This leaves the buffer manager with limited opportunities for maximizing performance. In most cases, overlapping I/O among multiple queries is only exploited if it occurs within a small time-space window.

* Work performed while the author was with the Department of Computer Science, University of Maryland, College Park

Recently, data warehousing products introduced the notion of *shared circular scans* (e.g. RedBrick). The idea is for a new scan to join (merge) with an existing scan on the index/table that currently feeds a running query. Obviously the latter scan will have to access the beginning of the index later. Microsoft's SQL server supports "merry-go-round" scans by beginning each index at the current position, however there is no explicit synchronization among the queries. In this paper we capitalize on, extend and formalize the idea of shared index scans. We propose a new "transcurrent execution model" (TEM) for concurrent user queries against tree indices, which is based on the notion of detached non-blocking query patches. This is achieved by immediately processing index pages that are in memory and detaching execution of disk-resident parts of the query that we call "patches". TEM allows uninterrupted query processing while waiting for I/O. Collaboration among multiple queries is accomplished by synchronizing the detached patches and exploiting overlapping I/O among them. We use a circular scan algorithm that dynamically merges detached requests on adjacent areas of the tree. By doing the synchronization *before* the buffer manager, we manage to achieve a near-optimal buffer hit ratio and thus, minimum interaction with the disk at the first time. Compared against shared circular scans, TEM is far more dynamic, since merging is achieved for any type of concurrent I/O, not just for sequential scans of the index.

We further exploit prefetching strategies, by grouping multiple accesses in a single *composite request* (analogous to multipage I/Os in [5]) that reduces communication overhead between the query threads and the buffer manager and permits advanced synchronization among them. An important difference is that our composite requests consist of pages that will actually be requested by the query. On the contrary, typical prefetching techniques retrieve pages that have high-probability of being accessed in the future, but might be proven irrelevant to the query. Another contribution of the TEM framework is that we address the issue of fairness in the execution of the detached patches and introduce the notion of *data flow* to achieve a steady flow of data pages to all query threads. This allows efficient execution of complex query plans that pipeline records retrieved by the index scans.

The rest of the paper is organized as follows: section 2 discusses the motivation behind our architecture. In section 3 we provide a detailed description of the TEM and discuss related implementation and performance issues. In section 4 we define data flow and show how to support time constraints in the scheduling of the detached query patches. Finally, section 5 contains the experiments and in section 6 we draw the conclusions.

2 Motivation

Most commercial data warehouses maintain a pool of session-threads that are being allocated to serve incoming user queries. In a data warehouse environment, the I/O is read-only and the query threads generate concurrent read page requests.

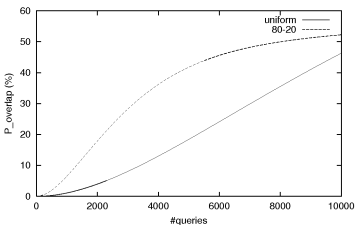

Fig. 1. Probability of overlapping I/O for uniform and 80-20 accesses

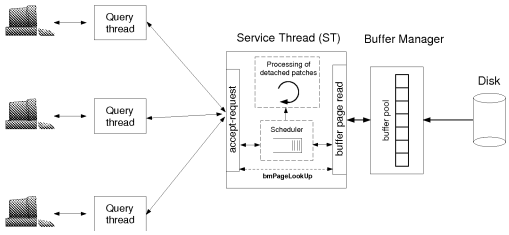

Fig. 2. System overview

In the database literature there is an abundance of research on buffer allocation and replacement strategies (e.g. [13, 4, 15]). For tree index structures, a Domain Separation Algorithm [17] introduced multiple LRU buffer pools, one for each level of the tree. Similar algorithms are discussed in [21, 7, 14]. However in [6] Chou and DeWitt point out that for indices with large fan-out the root is perhaps the only page worth keeping in memory. In data warehouses, indices are typically created and refreshed through bulk operations. As a result the trees tend to be rather packed and shallow and the potential improvements from a domain separation algorithm are limited.

In a concurrent multi-user environment there is limited potential for improving the buffering of the leaf-level pages. Given $n > 2$ concurrent queries on an 100MB index (6,400 16KB pages) and uniform distribution of accesses, the probability that the same data page is requested by 2 or more queries at any given time is:

$$p_{overlap}(n) = 1 - p(1/6400, 0, n) - p(1/6400, 1, n) \tag{1}$$

where:

$$p(a, k, n) = \frac{n!}{k! * (n-k)!} * a^k * (1-a)^{n-k} \tag{2}$$

is the standard binomial distribution. In Figure 1 we plot $p_{overlap}$ as the number of concurrent queries increases from 1 up to 10,000. We also plot the same probability for a more realistic 80-20 access pattern, where 20% of the pages receive 80% of the requests. These graphs show that for reasonable numbers of concurrent users, these probabilities are practically zero.

For most cases, where only a small part of the index fits in memory, overlapping requests have to occur within a relatively short time-window before the replacement strategy used flushes "idle" pages to disk. To overcome this limitation, we propose a non-blocking mechanism, in which requests for pages that are not in memory are detached from the current execution while the query thread advances its scan. We call this model of execution *transcurrent* because it detaches the processing of the requested I/O and delegates it as a *query patch* to an asynchronous *service thread*. This creates the illusion of a higher number of concurrent queries and results in increased chances of getting overlapping I/O.

3 Transcurrent Execution Model (TEM)

In this section we propose an architecture for the TEM. Our goal throughout the design was to require the least amount of modifications for integration into existing systems. The architecture introduces a new "Service Thread" (ST) for each active index, i.e. an index that is accessed by a query. This thread is interjected between the buffer manager and the query threads as shown in Figure 2.

The ST accepts all read-page requests for its corresponding index. Requests for pages that are in memory are immediately passed to the buffer manager and ST acts as a transparent gateway between the query thread and the buffer manager, as seen in the Figure.[1] For pages that are not in the buffer pool their requests are queued within the ST. The query thread that issued the request can either block until the page is actually read, or in some cases advance the index scan. In the later case the processing of the page is detached and performed internally, as a query patch, by ST when the page is actually fetched from disk.

Our goal is to optimize the execution order of the requests that are queued in ST in order to achieve higher buffer hit ratio. Assuming that the buffer manager is using a variant of the LRU replacement policy, an optimal execution strategy for ST would be to put requests on the same page together, so that subsequent reads would be serviced immediately from the pool. For that, we use a circular scan algorithm [8, 23] that is frequently used for scheduling I/O at the disk controller level. By using the same type of algorithm for our synchronization step, we not only achieve our goal, which is to maximize hit ratio, but we also generate an I/O stream that is bread-and-butter for a modern disk controller.

3.1 Index Scans for TEM

In an traditional index scan, each page request will block the query thread until the page is brought in the buffers. In the TEM if the page is not in the buffer pool, then the request is queued and will be executed at a later time according to the scheduling algorithm described in the next subsection. We then have the option to block the query thread, until the page is actually fetched from the disk or let it scan the rest of the tree.

For indexes that are created using bulk-load operations in a data warehouse, non-leaf pages occupy a very small fraction of the overall index space. Assuming relatively frequent queries on the view, most of these pages will be buffered in memory. As a result there is no evidence in getting any improvement by advancing the search for non-leaf pages, since in most cases the page will be available in the buffers anyway and the scan will not be blocked. Therefore, we do not consider detaching execution of non-leaf page requests and the query thread is blocked whenever such a request is not immediately satisfied from the buffer pool.

[1] We assume that the manager provides a `bmPageLookUp`(*pageId*) function for determining whether the requested page is in the buffers.

On the contrary, for the leaf (data) pages, because of the asynchronous execution, the probability that the page is in the pool at the exact time that the request is made is very small, see Figure 1. Therefore for leaf pages it is faster to detach their requests without checking the buffers with the `bmPageLookUp`($pageId$) function, otherwise a lot of thread congestion is happening.

3.2 Synchronization of Detached Query Patches

The ST maintains the current position on the file of the last satisfied disk I/O. Query threads continuously generate new requests, either as a result of a newly satisfied page request, or because of the non-blocking execution model that allows the search algorithm to advance, even if some page-reads are pending. Incoming page requests are split into two distinct sets. The first called "left" contains requests for pages before the last satisfied page of the index and the set called "right" (assuming a file scan from left to right) contains requests for pages in-front of the last page accessed. These are actually multi-sets since we allow duplicates, i.e multiple requests for the same page by different threads. The next request to satisfy is the nearest request to the current position from the `right` set that is realized as a heap. When the `right` set gets empty, the current position is initialized to the smallest request in the `left` set and the `left` set becomes the `right` set.

An extension that we have implemented but not include in this paper is to permit pages in the `right` set, even if they are within some small threshold on the left of the current position. The intuition is that these pages are likely to be in the cache of the disk controller. This allows better synchronization of queries that are slightly "misaligned". We also experimented with an elevator algorithm that switches direction when reaching the end/start of the file. To our surprise this algorithm was much slower than the circular scan algorithm that we described, probably due to conflicts with the scheduling implemented at the hardware of our disk controller. We plan to investigate this matter on different hardware platforms.

3.3 Composite Requests for Increased Overlapping I/O

An opportunity for optimization arrives when processing nodes just above the leaves of the tree. When such a node is processed we group multiple requests for leaf pages into a single *composite* multi-page request. Notice that we do not want to apply the same technique when accessing intermediate nodes higher in the tree structure, otherwise the search is reduced to a Breadth First Search algorithm that requires memory proportional to the width of the index.

Composite requests are more efficient because of the lower overhead required for the communication between the query-thread and the ST. However, an even more important side-effect of using composite requests it that the ST is given more information at every interaction with the query threads and is able to provide better synchronization. For instance, 10 concurrent single-page requests

provide at any given point 10 "hints" to the scheduler (ST) on what the future I/O will be. In comparison composite requests of 100 pages each,[2] provide a hundred times more information at any given point. Looking back at Figure 1 this generates the illusion of having $10 * 100 = 1000$ concurrent queries for which now the probability of overlap is $\frac{p_{overlap}(1000)}{p_{overlap}(10)} = \frac{1.0998e-2}{1.1e-6} = 9998$ times (i.e four orders of magnitude) higher! Furthermore, due to the non-blocking execution of the index scan, the query threads are constantly feeding the ST with more information, resulting in even higher gains. In this sense, even-though transcurrent query execution and dynamic synchronization can be seen as two orthogonal optimizations they are very naturally combined with each other.

3.4 Scheduling v.s. Cache Management

In our initial designs we thought of exploiting the large number of pending requested queued in ST for better cache management in a way similar to [22]. We tested an implementation of a modified LRU policy that avoids replacing pages that have pending requests on the right set of ST. Even though this resulted in a slightly higher buffer hit ratio than plain TEM, the gains did not show up in query execution times because of the per-request overhead of synchronizing ST's structures with the buffer manager. Our current implementation is cleaner, easier to integrate with existing systems and reorders the requests in a way that is ideal for LRU as shown in Figure 5.

4 Flow Control Extensions

Deferred requests are used in TEM as a mean to identify overlap among concurrent queries. A potential drawback is that a request that is diverted to the `left` set (see section 3.2) can be delayed while incoming requests keep pushing the request flow to the `right` set. Starvation is not possible, because the current file position is monotonically increasing up to the point that the last page of the index is read or the `right` set is exhausted, or both. However, for queries whose output is consumed by another thread, like a sub-query in a pipelined execution model; a delayed delivery of data will block the consuming thread.

In TEM, the ST executes detached leaf page requests while the query thread QT processes requests for non-leaf pages. Assuming that QT processes P_{QT} non-leaf pages per second and the ST processes P_{ST} leaf pages per second the aggregate processing for the query is: $P_{overall} = P_{QT} + P_{ST}$. Since output is only produced when scanning leaf pages, from the user point of view the effective progress of his query q that we denote as *data flow* (DF) is:

$$DF(q) = P_{ST} \tag{3}$$

Intuitively the larger this number is, the more bursty the output of the query gets. In the presence of many concurrent queries, a steady data flow for all of

[2] the fan out of a 2-dim R-tree with 16KB page size is 819. We assume that one in 8 leaves under a visited level-1 page are relevant to the query

them can be achieved by bounding the *idle time* t_{idle} of their leaf page requests. This idle time is defined as the period between two consecutive satisfied leaf page requests. The ST maintains a time-stamp information for each query that is running in the system and uses the index. This time-stamp is updated every time a leaf-page request is satisfied for the query. The administrator defines a "hint" for the maximum time W that a detached leaf-request is allowed to be delayed. Our implementation uses a *Flow Control Thread* that periodically checks for *expired* requests. Assuming that this thread awakes every T time-units and checks all time-stamps, then a query's output might block, waiting for a data page for a period of $t_{idle} = W + T$ and therefore the minimum data flow for the query will be:

$$DF_{low} = \frac{1}{W+T} \, pages/sec \tag{4}$$

Leaf page requests that have expired are inserted in a priority queue that uses the delay information for sorting them. As long as the ST finds expired requests in this queue, it processes them before those in the `right` set. The number of page requests in this queue is bounded by the number of concurrent queries in the system, as we only need at most one expired request per query to lower-bound the data flow. This prevents the data flow mechanism to become too intrusive if very small values for W and T are chosen.

5 Experiments

The experiments that we describe in this section use an implementation of TEM on top of the ADMS [18] database management system. We have used the TPC-D benchmark [9] for setting up a demonstration database. TPC-D models a business data warehouse where the business is buying `products` from a `supplier` and selling them to a `customer`. The measure attribute is the `quantity` of `products` that are involved in a transaction. We concentrate on an aggregate view for this dataset that aggregates the `quantity` measure on these three dimensions.

This view was materialized using a 3-dimensional R-tree stored in a raw disk device. We did not use regular Unix files in order to disable OS buffering. The total number of records in the view was 11,997,772 and the overall size of the R-tree was 183.8MB. The number of distinct values per attribute was 400,000, 300,000 and 20,000 for `product`, `customer` and `supplier` respectively. All experiments were ran in a single CPU SUN Ultra-60 workstation.

5.1 Comparison of TEM Against an Unsynchronized Execution

For the first experiment, we executed between 10 and 200 concurrent random range queries, each running in a different thread.[3] We used three different configurations. The first, which is denoted as "CEM" in the graphs, refers to the "conventional" execution model, where all queries are unsynchronized. The second

[3] Due to space limitation we omit similar results for skewed workload

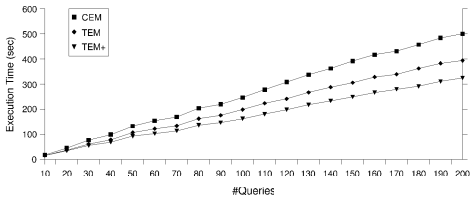

Fig. 3. Total execution time for 10-200 concurrent queries

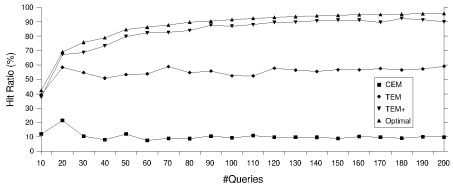

Fig. 4. Buffer hits

configuration used the transcurrent execution model with single page requests while the last one used composite requests as described in section 3.3. These configurations are denoted as TEM and TEM+ respectively. The ST maintains a pre-allocated pool of request objects that are used in the `left` and `right` sets. For the TEM/TEM+ configurations we set the request pool size to be 1MB and the buffer pool size of ADMS to 15MB. Since CEM does not use the ST, we gave the extra 1MB worth of memory to the buffer manager and set its pool size to be 16MB.

Figure 3 depicts the overall execution time for all queries for the three configurations, as the number of concurrent queries increases from 10 to 200. For relatively light workload (10 concurrent queries), the overall execution time is reduced by 13.9% in TEM and 16.9% in TEM+. As the number of queries increases, the differences between the three configurations become even more clear. The effective disk I/O bandwidth, which is computed from the number of page requests serviced per second was 8.67MB/sec for the CEM and 13.38MB/sec for the TEM+.

Figure 4 shows the buffer hit ratio as the number of queries increases. Because of the congested query I/O the unsynchronized execution achieves a very poor hit ratio. For the TEM+ the hit ratio increases with the number of concurrent queries and is almost optimal as shown in the Figure.

In Figure 5 we plot the (logical) page requests for 40 queries after they are reordered by the ST and passed to the buffer manager. The ST dynamically aligns requests at the `right` set to exploit spatial locality. These groups are further stacked as shown in the Figure. This I/O pattern is ideal for the LRU policy because of its high time-space locality.

5.2 Experiments with Flow Control

For the following experiment, we implemented the flow-control extensions described in section 4. This new configuration is denoted as TEM+/FC. We set the time-out period W to be 1sec and the sampling period of the Flow Control Thread T to 0.1sec. We then executed 50 concurrent queries and measured the average idle time for each one of them in the tree configurations (CEM,TEM+,TEM+/FC). For the CEM this idle time was 1.7sec on the average for all queries and can be justified from the heavy congestion in the disk for

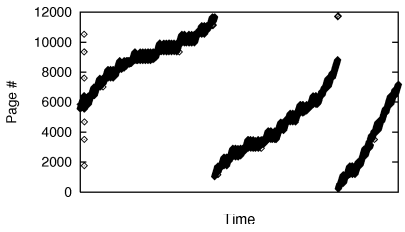

Fig. 5. Requests made from the ST to the Buffer Manager (TEM+)

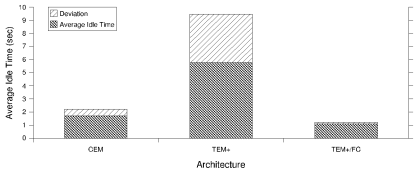

Fig. 6. Idle time comparison

50 concurrent queries. For TEM+ the average idle time is much higher at 5.8sec on the average and 21sec in the worst case. Notice that the total execution time is much lower for TEM+ : 91.35sec, v.s. 134.96 sec, i.e. 32.3% lower. The reason that the idle time is higher is because of detached query patches of leaf page requests that are being delayed in the `left` set as described in subsection 4. In Figure 6 we plot the average idle time over all queries along with the computed standard deviation. This graph shows that not only TEM+/FC provides the lowest idle time but also has the smallest standard deviation, which means that it treats all queries fairly.

6 Conclusions

In this paper we argued that conventional index scans and buffering techniques are inadequate for utilizing modern disk hardware and thus fail to support a highly concurrent workload against tree indexes. We showed analytically and through experiments that in an unsynchronized execution, overlapping I/O is only exploited if it occurs within a small time-window. We then introduced the transcurrent execution model (TEM) that exploits intra-parallelism of index scans and dynamically decomposes each query into a set of disjoint query patches. This allows uninterrupted processing of the index, while the disk is serving other I/O requests.

Our experiments demonstrate that the transcurrent query execution results in substantial performance benefits in terms of query execution time, buffer hit ratio and disk throughput. These benefits increase as the workload in the warehouse increases and offer a highly scalable solution to the I/O problem of data warehouses. In addition, TEM can be easily integrated into existing systems; our implementation of ST using posix-threads showed no measurable overhead from the synchronization algorithm and data structures, for 2 up to 500 concurrent queries in a single CPU workstation.

References

1. P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling.

ACM Transactions on Computer Systems, 14(4):311–343, 1996.
2. C. Y. Chan and Y. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of ACM SIGMOD*, pages 355–366, Seattle, Washington, USA, June 1998.
3. S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1), September 1997.
4. C.M. Chen and N. Roussopoulos. Adaptive Database Buffer Allocation Using Query Feedback. In *Procs. of VLDB Conf.*, Dublin, Ireland, August 1993.
5. J. Cheng, D. Haderle, R. Hedges, B. Iyer, T. Messinger, C. Mohan, and Y. Wang. An Efficient Hybrid Join Algorithm: A DB2 Prototype. In *Proceedings of ICDE*, pages 171–180, Kobe, Japan, April 1991.
6. H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Procs. of VLDB*, Sweden, August 1985.
7. W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM TODS*, 9(4):560–595, 1984.
8. R. Geist and S. Daniel. A Continuum of Disk Scheduling Algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, 1987.
9. J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems- 2nd edition.* Morgan Kaufmann, San Franscisco, 1993.
10. J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weiberger. Quickly Generating Billion-Record Synthetic Databases. In *Proc. of the ACM SIGMOD*, pages 243–252, Minneapolis, May 1994.
11. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD*, Boston, MA, June 1984.
12. Y. Kotidis and N. Roussopoulos. An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. In *Proceedings of ACM SIGMOD*, pages 249–258, Seattle, Washington, June 1998.
13. R. T. Ng, C. Faloutsos, and T. Sellis. Flexible Buffer Allocation Based on Marginal Gains. In *Procs. of ACM SIGMOD*, pages 387–396, Denver, Colorado, May 1991.
14. C. Nyberg. Disk Scheduling and Cache Replacement for a Database Machine. Master's thesis, UC Berkeley, July 1984.
15. E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data*, pages 297–306, Washington D.C., May 26–28 1993.
16. P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings of ACM SIGMOD*, Tucson, Arizona, May 1997.
17. A. Reiter. A Study of Buffer Management Policies for Data Management Systems. Technical Report TR-1619, University of Wisconsin-Madison, 1976.
18. N. Roussopoulos and H. Kang. Principles and Techniques in the Design of *ADMS±*. *IEEE Computer*, 19(12):19–25, December 1986.
19. N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of ACM SIGMOD*, pages 89–99, Tucson, Arizona, May 1997.
20. N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of ACM SIGMOD*, pages 17–31, Austin, 1985.
21. G. M. Sacco. Index Access with a Finite Buffer. In *Proceedings of 13th International Conference on VLDB*, pages 301–309, Brighton, England, September 1987.
22. A. Shoshani, L.M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional Indexing and Query Coordination for Tertiary Storage Management. In *Proceedings of SSDBM*, pages 214–225, Cleveland, Ohio, July 1999.
23. B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS*, Santa Clara, CA, May 1994.