Shared Index Scans For Data Warehouses

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

[1] AT&T Labs, 180 Park Ave, P.O. Box 971 Florham Park, NJ 07932-0000 USA
kotidis@research.att.com
[2] Institute for Advanced Computer Studies, University of Maryland, College Park
{isis,nick}@cs.umd.edu

Abstract. Tree based indexing structures like B-trees, B^+-trees, Bitmap indexes and R-trees have become essential for getting good performance when accessing vast datasets. However, most database research seems to ignore the behavior that the disk hardware observes during index scans. In this paper we aim to refocus attention on efficiently utilizing the underlying hardware during concurrent index scans. 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 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.

1 Introduction

On Line Analytical Processing (OLAP) involves complex ad-hoc queries that access millions of records and perform interesting aggregations. The main cost in terms of the time consumed of executing these queries is not doing the actual arithmetic, but of retrieving the data that affects the calculated items. In a relational DBMS, materialized derived relations (views) have long been proposed to speed up query processing. In a data warehouse, these views store redundant, aggregated information and are commonly referred to as *summary tables* [4]. B^+-trees are used for realizing the views, however they offer limited indexing capability for multi-attribute queries. An alternative is to consider the view records as multi-dimensional points and organize them using R-trees [31, 27, 17]. Tree based indexes are also exploited in multidimensional architectures as in the proprietary tree structure of Essbase [8], or in the form of Cube Forests [16].

Another form of redundancy is to index the detailed data in order to provide fast access to individual records. This is achieved using multiple B-trees [13] or variations of B-trees like the Log-Structured Merge Tree (LSM-Tree) [24] and the recently proposed Y-tree [15]. Many commercial systems use variations of Bitmap indices [22, 23, 3] that offer query performance improvements at a low disk space overhead. In its simplest form a Bitmap index is a B-tree that instead of storing at the leaf-pages a list of record-ids for each key value, it stores a compressed bit-map. There is one such bit-map for each value of the key.

The previous discussion shows that tree-based indices are extensively used to store and/or index large volumes of enterprise data for decision support applications. 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 pre-fetching are well-studied techniques [10, 35, 2] 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. This is especially true in a highly competitive multi-user database environment were different access patterns from multiple queries are interleaved and produce a noisy I/O pattern with very high entropy. Figure 1 shows traces from 40 concurrent queries against a 3-dimensional R-tree. Y-axis represents the logical page id of a request. One can observe some sequential patterns of requests for close by pages, but the overall picture is rather noisy

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 [6], against a B-tree index scan, pre-fetching guided by the upper B-tree levels is exploited to boost I/O throughput for a single query . This is achieved by identifying the list of leaf pages that need to be accessed and by initiating prefetches of those pages. This is a case where the query does not ask for one page at a time but rather for a whole batch at once. Analogous methods are introduced in [18, 1, 32]. However, in a multi-user environment, aggressive pre-fetching for multiple queries on disjoint parts of the index will only amplify congestion in the buffer manager [2, 32]. In most cases, overlapping I/O among multiple queries is only exploited if it occurs within a small time-space window. Even queries with high locality of I/O will only gain small benefits from the available memory buffers. In our tests with skewed workload with high locality of I/O, the buffer hit ratio was less than 52% in all cases. For uncorrelated queries this number drops in single digits. This is due to the lack of synchronization, at the buffer manager level, among the execution of multiple concurrent queries.

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 SQL server supports "merry-go-round" scans by begining 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 processing index pages that are cached by the buffer manager 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.

We further exploit pre-fetching strategies, by grouping multiple accesses in a single *composite request* (analogous to multipage I/Os in [6]) that reduces communication overhead between the query threads and the buffer manager and permits advanced synchronization among them. 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. TEM identifies correlated I/O requests among multiple queries and "joins" them to reduce congestion in the buffer manager. Compared with conventional pre-fetching strategies for B-tree indices, our framework is more generic since it also covers Bitmap indices and R-trees and can be easily adapted to virtually any tree-based structure used in the data warehouse.

Another important difference is that our composite requests consist of pages that will actually be requested by the query. On the contrary, typical pre-fetching techniques retrieve pages that have high-probability of being accessed in the future, but might be proven irrelevant to the query. Furthermore, to our knowledge we are the first to formalize a synchronization technique among concurrent queries in a dynamic multi-user environment. 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.

Our experiments demonstrate that transcurrent query execution results in better utilization of the CPU and in high locality of I/O, which in turn translates to substantial performance benefits. Compared to an unsynchronized query execution model, it provides substantial improvements in terms of query execution time, query data flow, buffer hit ratio and disk throughput. These benefits increase as the workload in the data warehouse increases and offer a highly scalable solution to the I/O problem of data warehouses. Furthermore, TEM is easily integrated with existing systems and provides no overhead regardless the workload.

The rest of this paper is organized as follows: section 2 discusses the motivation behind our architecture. In section 3 we make 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. These requests are handled by the buffer manager who will initiate an actual I/O to disk for all the pages not in the buffer pool.

The task of the buffer manager is to maximize the buffer hit ratio and therefore minimize the interaction with the disk. In the database literature there is an abundance of research on buffer allocation and replacement strategies (e.g. [30, 9, 7, 19, 5, 21]). For tree index structures, a Domain Separation Algorithm [25] introduced multiple LRU buffer pools, one for each level of the tree. Because of the multi-level organization, top level pages of the index have increased probabilities for staying in memory, resulting in about 8-10% better query performance than plain LRU. A similar algorithm (OLRU) is also discussed in [29] and is proved to be optimal under the assumption of a uniform distribution of index page reference densities. Variations of the Domain Separation algorithm are also discussed in [9, 20]. However in [7] 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. For example in an 1GB R-tree index, built with 16KB page size and 100% leaf page utilization [28], non-leaf pages contribute less than 0.5% of the index space. Even if LRU's hit ratio goes to 100% for non-leaf pages the increase in the overall buffer hit ratio will be insignificant. Similarly for compressed Bitmap indices most of the information is stored in the leaves of the tree that occupy most of the index. Therefore, for compacted indices, none of the previous algorithms is expected to behave differently than straightforward LRU.

In a concurrent multi-user environment there is limited potential for improving the buffering of the leaf-level pages. Each query thread works in pace with the buffer manager[1]: it requests one page at a time, await till it gets the page and then advances the index scan. 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 a 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 2 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.

The buffer pool allows the system to gain from overlapping I/O even if it doesn't occur at the same time-frame. The more buffers we have the larger this "grace-period" can be. In the unlikely event that the whole index fits in memory, all overlapping requests will be exploited, no matter their time-stamp. However for most cases where only 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. Two obvious solutions to the problem are: i) to increase the size of the buffers; and ii) to use an application specific replacement strategy like OLRU. However, another orthogonal optimization that, to our knowledge, has not been exploited is to increase the amount of overlapping I/O within the "grace-period" provided by the buffers. In Figure 2 we see that the probability of overlapping I/O slowly increases with the number of queries. Of course we do not want to introduce useless queries to increase our changes of overlap! Instead we want our existing workload to produce a heavier flow of correlated requests that will enable the buffer manager to better exploit overlapping I/O.

This is achieved by a non-blocking mechanism, in which page requests that are not in the buffers are detached from the current execution of the query thread 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, as demonstrated in Figure 2.

[1] Notice that the actual I/O is in most cases performed asynchronously.

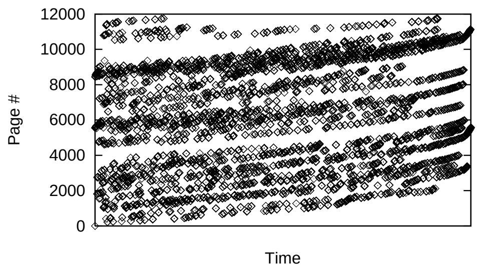

Fig. 1. Logical page requests during 40 queries against a single index

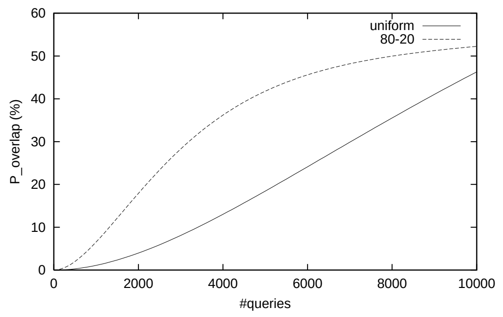

Fig. 2. Probability of overlapping I/O for uniform and 80-20 accesses

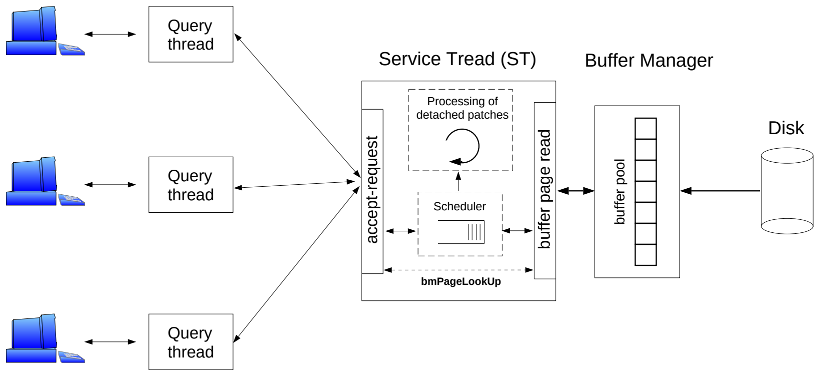

Fig. 3. System overview

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. Depending on the number of indices and disks in the system, the administrator may modify the configuration and assign multiple indices to each ST.

The ST accepts page requests generated from the query threads that scan the index. This is shown in Figure 3, where multiple query threads are communicating with the ST of the same index. This thread is interjected between the buffer manager and the query threads. Since concurrency control and locking in a database system are implemented at the buffer manager level, the ST does not need to deal with locking for concurrency control. Therefore, although TEM is mainly intended for data warehousing environments, it can also be adopted by OLTP systems.

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.[2] 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. We use a *QueryPatch* class to describe detached requests. This class contains among others:

– a logical *pageId* of the index page that needs to be retrieved.

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

– a *funcId* of the function that should be called by ST when the page becomes available along with necessary arguments in order to process the page.
– an *outFuncId* pointer to a function that redirects the output of the processed page. This can either go to the screen or another thread that processes the data retrieved from the index (e.g. in a join).
– the *queryThreadId* of the session thread initiated the request.

Our goal is to optimize the execution order of the requests that are queued in ST (see Figure 3) 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. If I/O requests from all queries were known a-priori, we could simply sort them, merge overlapping requests and execute them in any order. In a dynamic scenario, where new queries arrive asynchronously, we can use a circular scan algorithm [10, 35] that is frequently used for scheduling I/O at the disk controller level. For these system, the key idea is to align adjacent page requests and smooth-out costly random seeks on the disk. 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

TEM is used for querying hierarchical tree indexes like B-trees and R-trees. For simplicity in the notation we will be using as reference the standard R-tree structure. Since R-trees generalize the B-tree index for multiple keys, our technique is also applicable to B-trees/B^+-trees and Bitmap indexes.

As a running example we will be referring to a sample data warehouse that is built for analyzing supermarket transactions. A materialized view is used to store the total `quantity` of every `product` that each `customer` is buying. Table 1 provides some sample data for this view. The view is stored in a single 2-dimensional R-tree as proposed in [27, 17]. Each record is realized as a two-dimensional point where the `product` value defines the x-coordinate of the point and the `customer` value the y-coordinate respectively. The aggregate value is stored as the content of the point. For instance the first row of the table is mapped into point $(x = 1, y = 1, value = 26)$.

product	customer	sum(quantity)
1	1	26
1	2	24
1	4	41
1	5	9
2	2	6
2	3	5
3	1	3
3	2	27
3	4	7
3	5	15
4	1	6
4	2	10
4	3	1
4	4	21

Table 1. An aggregate view on `product` & `customer` attributes

Using this mapping the resulting R-tree index is shown in Figure 4. The buckets $1, 2, 3, \ldots, 8$ in the Figure represent nodes of the R-tree. Each node is stored in a different page on the disk with the page number shown on the top-left hand side of the node. The point-data of the view is stored at the leaf pages, as in a traditional B^+-tree. An intermediate node holds a list of entries that contain Minimum Bounding Rectangles for indexing the space of the children nodes as well as pointers to the children nodes [14].

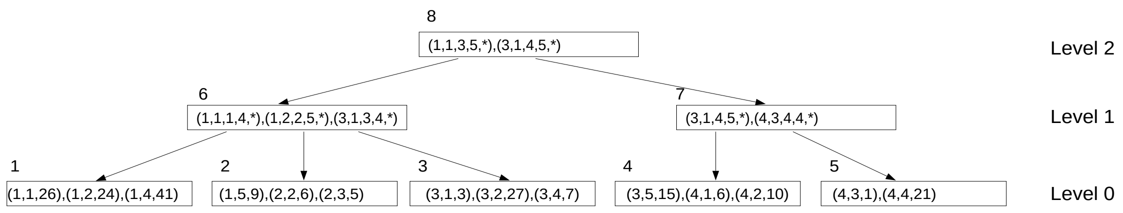

Fig. 4. R-tree for sample view

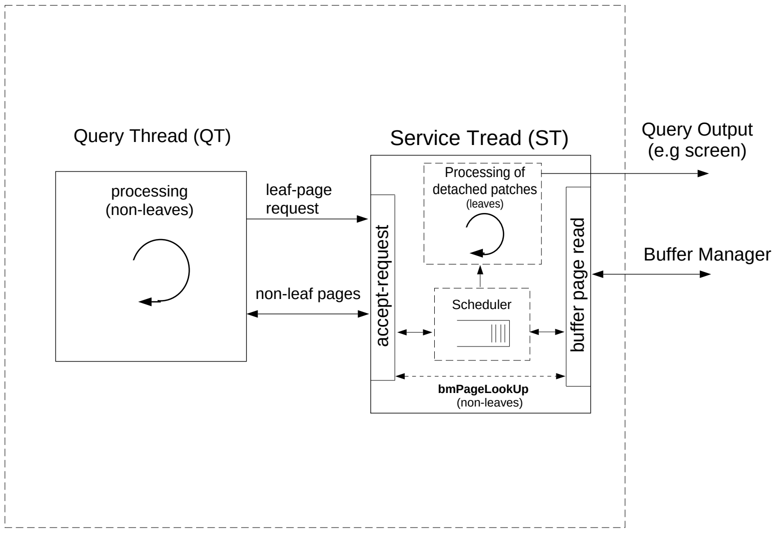

Fig. 5. Processing of detached I/O in TEM

Compared to a B^+-tree representation, this R-tree can be efficiently used for answering arbitrary *range queries* on both `customer` and `product` attributes. Such a query is depicted as a two-dimension rectangle in the (*customer, product*) attribute space. For instance a query that retrieves sales of products with id 1 through 3 to customer 1 can be encoded using rectangle $Q = (x_{min} = 1, y_{min} = 1, x_{max} = 3, y_{max} = 1)$.

For querying we assume a standard R-tree search algorithm [14] that descends the tree from the root following subtrees that overlap the query rectangle. For the previous query the search path will be (numbers correspond to page-numbers): $8 \to 6 \to 1 \to 2 \to 3 \to 7 \to 4$.

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 stop the query thread, until the page is actually fetched from the disk or let it scan the rest of the tree structure. However, the potential benefits for doing that depend on how far the algorithm has proceeded in traversing the tree. For instance at the root-level the index scan can not resume until the root is fetched from the disk. However, if we defer the processing of the left-most level-1 page of Figure 4, we can still work with its sibling page. Even more options are available when accessing a leaf data page. In such cases we can always advance the search until the tree is exhausted.

For indexes that are created using bulk-load operations in a data warehouse, non-leaf pages are typically a very small fraction of the overall index space. For an 1GB `product, customer` view of our example, only about 5MB of disk space is consumed by the non-leaf node pages of the corresponding R-tree.[3] 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

[3] we assume a 16KB page size and 4-byte integer attributes

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. On the contrary, for the remaining 99.5% of 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 2. Therefore for leaf pages it is faster to detach their requests without checking the buffers with the `bmPageLookUp`$(pageId)$ function, otherwise a lot a thread congestion is happening. If these pages happen to be in memory, the ST will immediately pool them from the queue as described in the next section and process them as a query patch.

Figure 5 depicts the proposed transcurrent execution model. The query thread (QT) initiates page requests while searching the index. These requests are scheduled internally by the ST. Execution of leaf page requests is always detached and performed by the service thread.[4] Non-leaf pages that are in the pool are handled by the QT. Otherwise, the QT is blocked until the page is fetched in memory by the ST. Therefore, processing of non-leaf pages is only performed by the query thread.

3.2 Synchronization of Detached Query Patches

ST reorganizes queued requests in a way that maximizes the performance of the buffer manager. This can be achieved by dynamically merging requests on adjacent areas of the tree through a circular scan or elevator algorithm. This family of algorithms has been shown [35] to perform well for I/O with significant read sequentiality. This is the case for index scans of B-trees, Bitmap indices and R-trees, at least when leaf pages are being scanned. As we mentioned earlier, these algorithms also result in better performance by the underlying disk controllers. This is because scheduling based on logical block (page) numbers has been shown to work well [35, 2], even if the underlying mapping to physical block numbers is unknown. Even in the case of a RAID-box that re-shuffles requests to different disks, a primary sequential access pattern generated by ST will give the best performance. This is because most partitioning schemes used in RAID have been designed for applications with primary sequential I/O (e.g. video streams). However, ST does not aim to do the job of, or replace disk controllers. It is only used as a mean to synchronize requests before the buffer manager and to allow deferred query execution, so that concurrent queries will share and not compete for memory buffers.

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". However, picking the right threshold requires knowledge of the hardware platform used.

We also experimented with an elevator algorithm that switches direction when reaching the end/start of the file. We did not expect better performance, but to our surprise this algorithm was much slower than the circular scan algorithm that we described, probably due to conflicts with the scheduling algorithm 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

Assume a query that descends the tree of Figure 6 following the paths denoted by the thick arrows. The working set for this query is:

$$13 \rightarrow 10 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 11 \rightarrow 5$$

[4] The spinning arrows in the Figure denote processing of requests within the threads.

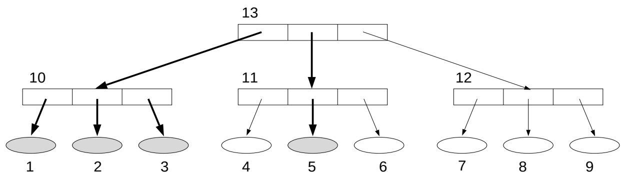

Fig. 6. DFS Index Scan

The query traverses two sub-trees generating one request at a time. An opportunity for optimization arrives when processing level-1 nodes; i.e nodes just above the leaves of the tree. When such a node is processed we know which of the underlying leaf pages the query will request. For instance after page 10 is read, we can group requests for leaf pages 1,2,3 into a single composite multi-page request denoted as $\{1, 2, 3\}$. 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.

For handling multi-page requests the *QueryPatch* class is extended to contain a (sorted) list of page numbers: $r = \{p_1, p_2, \ldots, p_n\}$. When testing a composite request against the current position on the file, we use the smallest page number of the group p_1. If it is smaller, the multi-page request is added to the `left` set, otherwise it is added to the `right` set. Using composite requests the query shown in Figure 6 will generate the following requests:

$$13 \rightarrow 10 \rightarrow \{1, 2, 3\} \rightarrow 11 \rightarrow \{5\}$$

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,[5] provide a hundred times more information at any given point. Looking back at Figure 2 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. Due to the non-blocking execution the ST gets more input from the query threads and, therefore makes more informed decisions that further allow the query threads to process the index, and generate new requests, at a higher pace.

3.4 Scheduling v.s. Cache Management

In the context of tertiary storage management an idea similar to TEM has recently been deployed in STACS [33] to optimize the use of disk cache for files read from tapes and minimize the number of tape mounts. A difference is that for each query, STACS using a combination of bit-sliced indices knows in advance the files that are needed. Thus the main concern is about cache admission/release policies, while in TEM the focus is not on cache management but on exploiting intra parallelism of index scans and overlapping I/O before reaching the cache. TEM should impose no overhead while scheduling the requests. This is because disk accesses are in the orders of milliseconds, while for tertiary storage the latency of mounting a tape can be several minutes.

In our initial designs we also though of exploiting the large number of pending requested queued in ST for better cache management. 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 Figures 9, 10.

[5] 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

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. When this occurs, the `left` and `right` sets will be swapped and therefore, in the worst case, the delayed request will be satisfied after a full scan of the file.

From the user point of view, an index scan produces results only when leaf pages are processed. These pages in R-trees and B^+-trees hold the actual values while in Bitmap indices and B-trees contain compressed bit-maps and pointers where the records are stored respectively. For some queries a delayed delivery of data (leaf) pages might be unacceptable. This is the case 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 query thread continues execution even-though read requests for leaf pages are "delayed" by the scheduler. This means that internally, the scan advances at all times. This is demonstrated in Figure 5. The spinning arrows within the query thread and the ST denote internal asynchronous processing of index pages. 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 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. A minimum idle time can not be fully guaranteed because it depends on other parameters such as the load on the CPU, the disk subsystem etc, and in addition a delayed request will have to wait other delayed requests that have longer idle times. However, the number of page requests in the priority queue is bounded by the number of concurrent queries in the system. This is because we only need at most one expired request per query to lower-bound the data flow. This also 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 [26] database management system. For these experiments we have used the TPC-D benchmark [11] for setting up a demonstration database. TPC-D models a business data warehouse where the business is buying `products` from a `supplier` and sells them to a `customer`. The measure attribute is the `quantity` of `products` that are involved in a transaction. For the first set of experiments, we concentrate on an aggregate view for this dataset that aggregates the `quantity` measure on these three dimensions. The SQL description for the view is:[6]

[6] fact_table is used to store all sales data.

```
create view View1 as
select product, customer, supplier,
       sum(quantity) as total_quant
from fact_table
group by product, customer, supplier
```

This view was materialized in the disk using a single 3-dimensional R-tree stored in a raw disk device. We did not use regular Unix files in order to disable OS buffering. This is a common technique recommended by all commercial-strength database systems since they provide superior buffering techniques for database applications than the OS [34]. The buffer manager of ADMS, as in most commercial database systems, uses LRU replacement policy for tree indices. As we already mentioned in section 2, for indices that were bulk-loaded we do not expect any differences between the performance of plain LRU and a domain separation algorithm.

We used a SUN Ultra-60 workstation with a 360MHz UltraSPARC-II CPU and an 18GB SEAGATE Cheetah hard drive connected through an ultra wide SCSI bus. The drive controller operates at a lower level than the DBMS (or the OS) buffering and thus all optimizations for reducing seeks or pre-fetching are performed for raw-devices also. This drive is able to provide a sustained data transfer of 18.11MB/sec for serial reads from the raw device. We created a 2GB TPC-D dataset (scale factor=2). 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.

5.1 Query Description

For searching the view we used queries with ranges on the grouping attributes `product`, `customer` and `supplier`. These queries are formulated in SQL using the template of Figure 7.

`SELECT`	$product, customer, supplier, total\_quant$	
`FROM`	View1	
`WHERE`	$product \geq min_{product}$	and $product \leq max_{product}$
`AND`	$customer \geq min_{customer}$	and $customer \leq max_{customer}$
`AND`	$supplier \geq min_{supplier}$	and $supplier \leq max_{supplier}$

Fig. 7. Query template for View1

We used two different sets of queries:

Uniform Set: For these queries, the minimum value for each attribute is selected uniformly from the attribute's domain. The upper value is then chosen to create a random range that covers up to 25% of the attribute's values. For the three dimensional dataset that we used the maximum selectivity of an uniform query is $0.25^3 = 1.56\%$.

80–20 Set: For this query-set the lower value for each attribute is chosen using the 80-20 self-similar distribution [12] and the upper value as in the previous set.

5.2 Comparison of TEM Against an Unsynchronized Execution

For the first experiment, we used the *Uniform Set* of queries and varied the number of concurrent queries in the system from 10 up to 200. Each of these queries was executed in a different thread. 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 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 (QueryPatch) objects that are used in the `left` and `right` sets. For the TEM/TEM+ configurations we set the request pool size to be

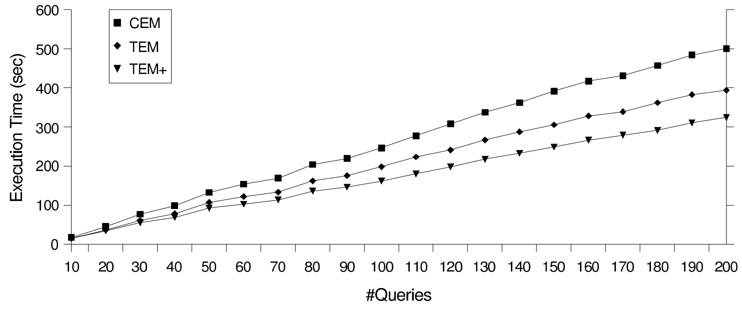

Fig. 8. Total execution time for 10-200 concurrent queries

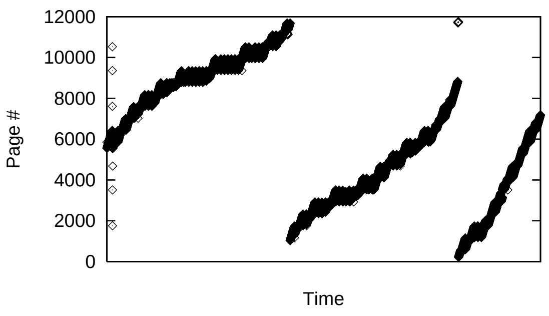

Fig. 9. Requests made from the ST to the Buffer Manager (TEM+)

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 8 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. For 200 concurrent queries the system with the TEM+ reduces the overall execution time from 500.4sec down to 324.4sec, a 35.2% reduction over the unsynchronized execution. 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+. Compared to the raw bandwidth of the disk, the TEM+ utilizes 74% of the disk serial transfer rate with all 200 query threads and the service thread running in a single CPU workstation.

In Figure 9 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.

Figure 10 shows the buffer hit ratio as the number of queries increases. Because of the noisy query I/O (similar to that of Figure 1) the unsynchronized execution achieves a very poor hit ratio. For the TEM+ the hit ratio increases with the number of concurrent queries up to 92.4% for 200 queries. This is because the more the queries the higher the probability of having overlapping I/O gets, see Figure 2. In Figure 10 we also plot the optimal buffer hit ratio. This was computed by examining the I/O traces of the queries and reordering the requests in a way that maximizes the buffer hits. Notice that the TEM+ provides a near-optimal performance.

Table 2 shows the results of a second experiment using 40 and 200 concurrent queries of the second query set (80–20 Set). This query set is very skewed with high locality of I/O. The execution time column shows the time to execute all queries. The effective disk I/O was computed as the overall I/O generated by all queries divided by the total execution time.

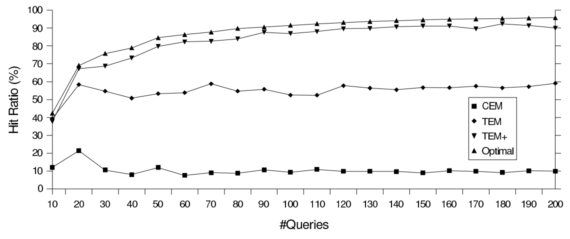

Fig. 10. Buffer hits

#Queries	Execution Model	Execution Time	Hit Ratio	Effective I/O
40 queries	CEM	92.87 sec	43.11%	10.21 MB/sec
	TEM+	74.29 sec	79.48%	12.76 MB/sec
200 queries	CEM	445.69 sec	51.76%	10.28 MB/sec
	TEM+	344.67 sec	92.13%	13.29 MB/sec

Table 2. Performance data for the 80–20 Query Set

5.3 Experiments with Flow Control

For the following experiments, we implemented the flow-control extensions described in section 4. This new configuration is denoted as TEM+/FC. For the analysis we used a set of 50 concurrent queries from the uniform set. For each query we computed the average idle time, that is the time between two satisfied consecutive leaf page requests, see subsection 4. Notice that this time also includes the overhead of processing the data page. For the 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. Figure 11 shows the average idle time for each individual query, for the three execution models. For the CEM this idle time is 1.7sec on the average for all queries and can be justified from the heavy congestion in the disk for 50 concurrent queries. For TEM+ the average idle time is higher at 5.8sec on the average and 21sec in the worst case for query #50. Notice that the total execution time is much lower for TEM+ : 91.35sec, vs. 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. Figure 11 shows that the TEM+/FC architecture outperformed the other two. On the average the idle time was just 1.09sec over all queries resulting in a *data flow* of 938 processed records per second to each thread. In Figure 12 we plot the average idle time for all queries along with the computed standard deviation. This graph shows that TEM+/FC provides the lowest idle time and has the smallest standard deviation, which means that it treats all queries fairly. On the contrary the deviation in the CEM is higher due to the thrashing in the buffer pool and the disk. The price for this performance is rather small. The total execution time for all 50 queries was 103.88 sec for TEM+/FC, about 13.7% slower than the TEM+, but still 23% faster than CEM.

5.4 Experiments with B^+-trees

TEM provides benefits when used with one dimensional indices like B-trees and Bitmap indices. For demonstration we used a 10MB B^+-tree. The B^+-tree record consisted of a integer key and a float measure. The total number of tuples indexed was 1,264,626. The distribution of keys was uniform. For the first run we used the *Uniform set* and varied the number of concurrent queries from 10 to 200. Figure 13 shows the increased buffer hits (for 2MB of buffers) compared to the conventional model. The response time reduction was about 15% regardless of the load.

In a final experiment we tested TEM against random range queries with varying selectivity. Figure 14 shows the hit ratio for 10 up to 100 concurrent queries with selectivities from 1%, up to 40%. As the selectivity and/or the number of queries increase, TEM's performance gets better. It is worth pointing out that in most cases, TEM has a near-optimal performance. For example for 10 queries with selectivity 1% TEM achieves hit ratio of 6.94% while the optimal hit ratio for this workload was 7.04%

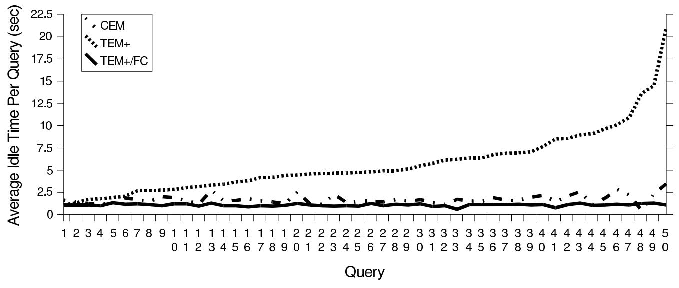

Fig. 11. Average idle time per query

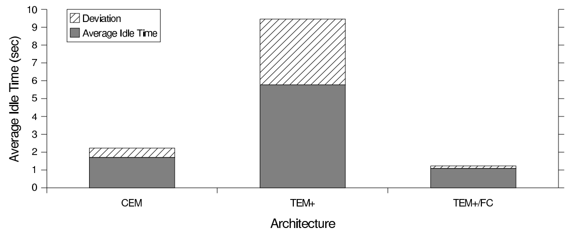

Fig. 12. Idle time comparison

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. Pending requests are dynamically merged for adjacent areas of the tree. We further proposed the use of multi-page (composite) requests that allow smaller synchronization overhead and higher probability of overlapping I/O among the detached query patches. For queries that require a steady flow of processed tuples we introduced and used the *data flow* as a metric of the pace of each query's progress.

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. G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. In *Proceedings of ICDE*, pages 538–547, Vienna, Austria, 1993.
2. 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.
3. C. Y. Chan and Y. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 355–366, Seattle, Washington, USA, June 1998.
4. S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1), September 1997.

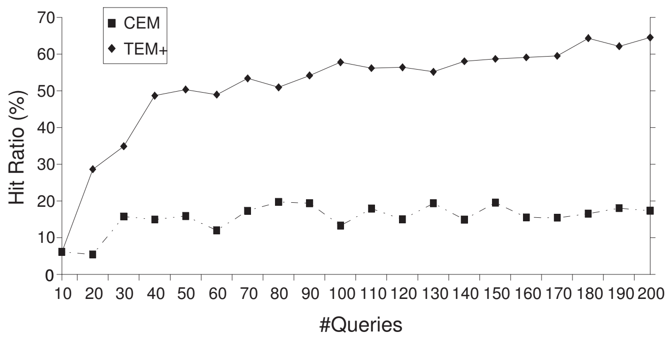

Fig. 13. Buffer hits

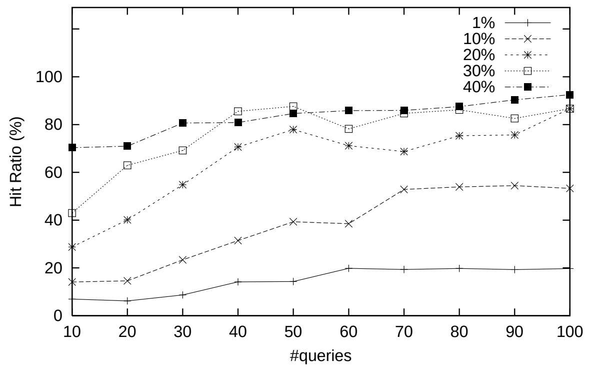

Fig. 14. Varying selectivities/#queries

5. C.M. Chen and N. Roussopoulos. Adaptive Database Buffer Allocation Using Query Feedback. In *Procs. of the 19th Intl. Conf. on Very Large Data Bases*, pages 342–353, Dublin, Ireland, August 1993.
6. 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.
7. H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on VLDB*, pages 127–141, Stockholm, Sweden, August 1985.
8. Robert J. Earle. Arbor Software Corporation, US patent #5359724, Oct 1994. "http://www.arborsoft.com".
9. W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM TODS*, 9(4):560–595, 1984.
10. R. Geist and S. Daniel. A Continuum of Disk Scheduling Algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, 1987.
11. J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems- 2nd edition*. Morgan Kaufmann, San Franscisco, 1993.
12. 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.
13. H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proceedings of ICDE*, pages 208–219, Burmingham, UK, April 1997.
14. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.
15. C. Jermaine, A. Datta, and E. Omiecinski. A Novel Index Supporting High Volume Data Warehouse Insertions. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 235–246, Edinburgh, Scotland, U.K., September 1999.
16. T. Johnson and D. Shasha. Hierarchically Split Cube Forests for Decision Support: description and tuned design. Working Paper, 1996.

17. Y. Kotidis and N. Roussopoulos. An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–258, Seattle, Washington, June 1998.
18. C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques. In *Proceedings of the EDBT*, pages 29–43, Venice, Italy, 1990.
19. R. T. Ng, C. Faloutsos, and T. Sellis. Flexible Buffer Allocation Based on Marginal Gains. In *Procs. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 387–396, Denver, Colorado, May 1991.
20. C. Nyberg. Disk Scheduling and Cache Replacement for a Database Machine. Master's thesis, UC Berkeley, July 1984.
21. 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.
22. P. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 24(3):8–11, Sept 1995.
23. P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 38–49, Tucson, Arizona, May 1997.
24. P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.
25. A. Reiter. A Study of Buffer Management Policies for Data Management Systems. Technical Report TR-1619, Mathematics Research Center, University of Wisconsin-Madison, 1976.
26. N. Roussopoulos and H. Kang. Principles and Techniques in the Design of $ADMS\pm$. *IEEE Computer*, 19(12):19–25, December 1986.
27. N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 89–99, Tucson, Arizona, May 1997.
28. N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of 1985 ACM SIGMOD*, pages 17–31, Austin, 1985.
29. G. M. Sacco. Index Access with a Finite Buffer. In *Proceedings of 13th International Conference on VLDB*, pages 301–309, Brighton, England, September 1987.
30. G. M. Sacco and M. Schkolnick. A Mechanism for Managing the Buffer Pool in a Relational Databas System Using the Hot Set Model. In *Proceedings of 8th International Conference on VLDB*, pages 257–262, Mexico City, Mexico, September 1982.
31. S. Sarawagi. Indexing OLAP Data. *IEEE Bulletin on Data Engineering*, 20(1):36–43, March 1997.
32. S. Sarawagi and M. Stonebraker. Reordering Query Execution in Tertiary Memory Databases. In *Proceedings of the 22nd VLDB Conference*, pages 156–167, Mumbai(Bombay), India, September 1996.
33. 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.
34. M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.
35. B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS*, pages 241–251, Santa Clara, CA, May 1994.