Reordering Query Execution in Tertiary Memory Databases*

Sunita Sarawagi Michael Stonebraker

Computer Science Division
University of California, Berkeley, CA 94720, USA
{sunita,mike}@cs.berkeley.edu

Abstract

In the relational model the order of fetching data does not affect query correctness. This flexibility is exploited in query optimization by statically reordering data accesses. However, once a query is optimized, it is executed in a fixed order in most systems, with the result that data requests are made in a fixed order. Only limited forms of runtime reordering can be provided by low-level device managers. More aggressive reordering strategies are essential in scenarios where the latency of access to data objects varies widely and dynamically, as in tertiary devices. This paper presents such a strategy. Our key innovation is to exploit dynamic reordering to match execution order to the optimal data fetch order, in all parts of the plan-tree. To demonstrate the practicality of our approach and the impact of our optimizations, we report on a prototype implementation based on Postgres. Using our system, typical I/O cost for queries on tertiary memory databases is as much as an order of magnitude smaller than with conventional query processing techniques.

1 Introduction

We investigate new, aggressive reordering strategies for speeding up queries on relational databases. The relational data model provides a set-oriented semantics where the order of processing tuples is unimportant for correctness. For instance, a SELECT query does not require that the relation be scanned sequentially; there is flexibility to fetch the data blocks in any arbitrary order. Database designers have used this flexibility in building multiple access paths to relations, designing multiple methods of processing joins and optimizing queries based on evaluation of different methods. However, once a plan is optimized, execution of the plan proceeds in a fixed manner with the result that data pages are demanded in a fixed order that was optimized for a given data layout. The only form of dynamic reordering available during execution is through low-level I/O device schedulers or in some cases by asynchronous prefetching. The reordering that existing schedulers can achieve is limited to I/O requests from multiple users or to batch prefetching from processes doing asynchronous I/O. These existing schedulers may have greater opportunities for optimization if prefetching is done in larger batches; however, prefetching in large amounts can adversely affect caching performance [6]. In this paper, we show that a effective way around this problem is to dynamically reorder execution to match the optimal data fetch order. If data in some part of the plan tree is "near by" now and will get "further away" later, it is advantageous to process the "near by" data first instead of waiting for the data "far away". This paper describes the implementation and evaluation of this simple idea in practical settings.

The key features of our framework for reordering execution are:

1. Relations are comprised of **chunks** that are available together.
2. Each query plan tree is divided into parts (called **subqueries**[1] here) that can be executed independently in arbitrary order.
3. A **scheduling unit** collects subqueries from many users and decides at runtime the order in which they are executed.
4. A **reorderable executor** communicates with the scheduler to process the query plan in the order dictated by the scheduler.

The scheduling unit (item 3 above) has been described in [29]; there we designed an algorithm that determines the order in which data should be fetched to reduce I/O cost and presented preliminary results using simple hand-compiled queries. In contrast, this paper deals with the design and implementation of the reorderable executor that can extract the subquery lists and execute them in an arbitrary order (items 2 and 4).

*This research was sponsored by NSF Grant IRI-9107455, ARO Grant DAAL03-91-G-0183, and DARPA Contract DABT63-92-C-0007. Additional support was provided by the University of California and Digital Equipment Corporation under Sequoia 2000 research grant #1243.

**Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996**

[1]The term *subqueries* is not to be confused with the SQL notion of subqueries. We use the subqueries to refer to parts of query.

For easy integration into an existing execution engine, we extended the plan tree data-structure with three new meta-operators that are added in an extra phase between optimization and execution of the plan tree. These operators enable the executor to communicate and synchronize with the scheduler for ordering the execution of subqueries.

The last part of this paper reports on a evaluation of the above ideas for reordering execution in the context of a tertiary memory database. We also quantify the overheads of reordering, and show that they are small compared to the performance gains, which are often as high as an order of magnitude.

1.1 Applications

Reordering execution can be beneficial in all cases where the access latency of data in various parts of the plan-tree varies widely and dynamically. We present below a list of potential situations:

- *Tertiary memory systems:* A typical tertiary storage device consists of a large number of tapes or optical disks (we will use the term *platter* to refer to both tapes or optical disks), a few read-write drives and even fewer robot arms to switch the platter between the shelves and the drives. The time to load and unload platters from drives is often high. It is, therefore, beneficial to order execution to first process data on the currently loaded unit before unloading it.
- *Cache systems:* Cached data is "nearer" than the uncached data and it might help to process the cached data before fetching more data that might replace it. Database cache-replacement algorithms [9, 14, 33] are extensively researched but none of these algorithms have considered applying execution reordering to adapt to the cached data.
- *Broadcast disks for mobile computing:* Broadcast disks [1, 17, 15] are gaining importance in mobile and asymmetric environments for reducing number of messages from the clients to the data servers. With broadcast disks, data is periodically transmitted by base stations or servers to multiple clients instead of clients explicitly requesting data from the servers. It will help to reorder the execution of the clients so that they process the plan tree in the order in which data is broadcast by the server instead of following a fixed order of processing.

1.2 Motivating example

We start with a few examples of potential benefits due to execution reordering on a tertiary memory system. Consider a **single-drive** tape jukebox with three relations stored across three tapes as shown in Figure 1. Relation S is stored as three contiguous chunks on two tapes, R as three chunks on three different tapes and T on a single tape. The size of each chunk of R is 1 GB and of S and T is 0.5 GB each. A 1 GB disk cache is used for staging data to and from the tertiary memory in units of 256 KB. Consider the following scenarios:

Figure 1: Layout of R, S and T on tape

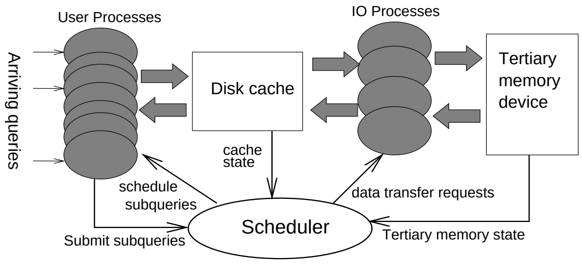


Figure 2: The architecture of the tertiary memory database system with the centralized scheduler

- User-1 submits a sequential scan query on S. The best order to fetch S is S_1, followed by S_3 and then S_2. Suppose after user-1 has processed all of S, user-2 submits a sequential scan on S. In this case, the best way to process user-2's scan is in the order S_3, S_2 and then S_1 since S_3 and S_2 are still in cache.
- User-1 submits query-1 and after tape-2 has been loaded to fetch S_1, User-2 submits an index scan on R. In this case, the best order for fetching the qualifying blocks of R is to first fetch R_2's blocks, then R_1's and finally R_3's.

In both these cases, we notice that simply changing the execution order of each user's query based on data layout is not sufficient when multiple users interact. The optimal execution order during multi-user processing could be quite different from the best static order. Thus, we need to be able to *dynamically* reorder execution.

1.3 Outline of the paper

Section 2 gives an overview of our architecture and reviews the design of the scheduling unit. Section 3 presents the design of the new executor which extracts the list of subqueries and executes them in arbitrary order. Section 4 describes our prototype and presents a performance evaluation. Section 5 presents related work. Finally, concluding remarks appear in Section 6.

2 Architecture Overview

Figure 2 sketches the architecture of our tertiary memory database system introduced in [29]. We assume a process-per-user architecture where each user-session has a separate process serving its queries. An arriving query is first compiled by the user process. The user-process then extracts the list of subqueries from the query and submits the list to the scheduler process.

We have a single centralized scheduler process that receives subqueries from all user processes and decides when they are executed. The scheduler maintains a set of I/O processes that transfer data between the disk cache and tertiary memory. As soon as all the data accessed by a subquery are available in the disk cache, the scheduler marks the subquery as "ready" for execution. The user processes contact the scheduler to collect ready subqueries and block until some subqueries are ready. After finishing execution of these ready subqueries, they send a notification to the scheduler, which can then decide to evict the cached data used by that subquery when desirable.

Each relation consists of a number of **fragments**. A fragment is the part of a relation that lies contiguously on a storage medium. For instance, in Figure 1, S_1, S_2 and S_3 are fragments of S. The fragments of a relation correspond to the data **chunks** of our framework introduced in Section 1 (item 1). We further restrict the size of each fragment based on the size of the cache, the latency of access on tertiary memory, the data transfer rate and the number of concurrent users as discussed in [29].

The scheduler (1) co-ordinates data movement between the disk cache and tertiary memory, (2) schedules query execution for each user process, and (3) decides what data is cached to or evicted from the disk cache. The scheduler makes these decisions based on system-wide information about pending subqueries from all users, the state of the disk cache and the tertiary memory, e.g, what platter is currently loaded. Details of how these decisions are made is given in [29]. When deciding on the order of executing subqueries, the scheduler's objective is to maximize the overall system throughput. Hence, for a given user-process, one or more subqueries could be scheduled for execution together in an arbitrarily interleaved fashion with those of other users.

3 Execution Engine

In this section we describe the design of the execution engine of the user processes. We first list the requirements needed by an execution engine to support reordering. Then we present the mechanism for supporting these requirements.

3.1 Specifications

1. *Submit to the scheduler a list of subqueries to be executed*
   The scheduler does not need to know all the details of the subquery, only which fragments are needed *together* in executing the subquery. Consider a nest-loop join between relations R and S where R has three fragments R_1, R_2 and R_3, and S has two fragments S_1 and S_2. The list of subqueries submitted to the scheduler, called the `SQ-list`, is:

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

2. *Execute subqueries out-of-order*
   There should be no **ordering constraints** between the subqueries submitted to the scheduler. For the two way join example above, the subquery (R_2, S_1), for instance, might be scheduled before the subqueries (R_1, S_1) and (R_1, S_2). Thus when the operators of a plan tree have precedence constraints on them, the subqueries must be submitted in multiple stages. For instance, for hash-join queries the inner fragments have to be fetched and the hash-table built, before processing any fragments of the outer relation.
3. *Execute multiple subqueries together*
   The scheduler could have more than one subquery ready for execution. We require that the executor be able to process multiple subqueries together. Executing one subquery at a time can lead to redundant computation for joins, since the scans on the outer relation cannot be shared across multiple fragments of the inner relation. For instance in the two-way join example, if S_1, S_2 and R_3 are cached, the scheduler will "ready" both the subqueries (R_3, S_1) and (R_3, S_2). The executor must be able to join R_3 with both S_1 and S_2 in one scan of R_3. Hence, although executing each subquery separately would allow for easy implementation, we must provide a means of executing multiple subqueries together.

3.2 Design

In this section, we describe the design of an executor that meets the specifications of Section 3.1. We base our discussion on the Postgres execution engine, in which each query plan is a tree of operators. All operators are implemented as iterators and support a simple start-next-end interface. Most relational database systems have analogous operator-based execution engines and can be extended similarly [13].

A query is first optimized as usual except for a few minor changes related to sorting via index scans that are discussed in Section 3.4. The optimized plan tree is then processed to extract the list of subqueries as discussed in Section 3.2.1. In Section 3.2.2 we discuss how execution proceeds out of order.

3.2.1 Extracting subquery lists

This proceeds in two phases: the fragmentation phase and the extraction phase.

Fragmentation phase: In this phase, each scan node on each base relation is replaced by a *combine node* that contains a list of scan nodes on the fragments of the base relation. The type of scan (sequential scan or index scan) on the fragments is the same as on the base relation. We assume that all the fragments of a relation have the same set of indices. For example, in Figure 3(a) we show the plan-tree of a 3-way join with three sequential scan nodes on base relations S, U and T. In Figure 3(b) we show the plan-tree after fragmentation.

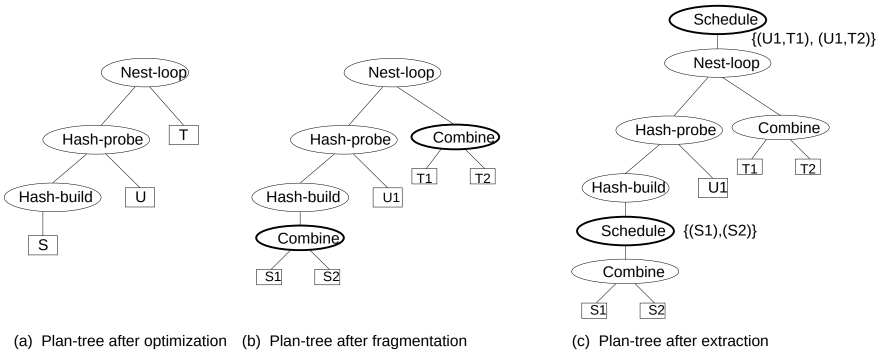


Figure 3: Example of a three-way join. The scan nodes are not shown for clarity; all of them are sequential.

Extraction phase: In this phase we extract the `SQ-lists` and insert special nodes called *schedule nodes* that are responsible for communicating with the scheduler and keeping synchronization information during execution. Each schedule node has an associated `SQ-list`. Because of precedence constraints between operators (Section 3.1(item 2)), we could have multiple `SQ-lists` in a plan-tree. For example, the plan-tree in Figure 3(b) has ordering constraints between the hash-build and hash-probe nodes. Hence, we added a schedule node before the hash-build node since the hash-build stage has to complete before starting processing on any nodes above it. We add a second schedule node at the top for the rest of plan-tree.

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

Find-Sub-Query for various nodes
Combine node:
  return list of fragments under the combine node
Hash-build, Aggregate or Sort node:
  `query-list` = Find-Sub-Query(subtree under node)
  if `query-list` non-empty
    add schedule node with `query-list` below node
  return `empty-list`
Join node:
  `listL` = Find-Sub-Query (left subtree)
  `listR` = Find-Sub-Query (right subtree)
  `query-list` = cross product of `listR` and `listL`
  If `listR` is empty, query-list = `listL`
  If `listL` is empty, query-list = `listR`
  return `query-list`

In Figure 3, the "Find-Sub-Query" call on the Hash-build node adds a schedule node with the list $\{(S_1), (S_2)\}$ and returns the empty-list. The "Find-Sub-Query" call on the Hash-probe node returns $\{(U_1)\}$ and on the right branch of the Nest-loop node returns the list $\{(T_1), (T_2)\}$. The "Find-Sub-Query" call on the Nest-loop node returns the cross product $\{(U_1, T_1), (U_1, T_2)\}$ that is stored in a schedule node at the top of the tree.

3.2.2 Executing queries out-of-order

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

For efficiency reasons (discussed in Section 3.1,item 3) we want to execute all subqueries of a plan-tree from a single plan-tree instead of building a separate plan-tree for each subquery. This requires us to keep track of what subquery of the plan-tree is currently being executed. We do so by marking the scan nodes of the subqueries currently being executed as `available` and all other scan-nodes `suspended`. The plan-tree is then processed as usual: starting from the root of the plan tree, successive "next" calls are made to each node of the tree. When a "next" call is made on a combine node it submits the "next" call to a scan node underneath it that is marked `available`. A "next" call on a `suspended` scan node returns no tuple. Thus, only scan-nodes of currently scheduled subqueries participate in execution.

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

1. Make a "next" call on the node underneath the schedule node to get the next tuple, t
2. If t is valid, return t
3. Else, submit the stored `SQ-list` to the scheduler, if it has not already been submitted.
4. Inform the scheduler of the completion of the last batch of scheduled subqueries, if any, and mark the scan nodes of those subqueries as `suspended`.
5. Make a blocking call to the scheduler to get the next collection of subqueries. Let Q be the collection of "ready" subqueries returned by the scheduler.
6. If Q is empty, then all subqueries have been executed, therefore return EOF.
7. Else, enable Q for execution by marking all the scan nodes appearing in Q as `available`.
8. Finally, make a "next" call on the node underneath and return the tuple obtained.

We will illustrate the above steps with the plan-tree in Figure 3. Initially, all the fragments are marked `suspended`. The first "next" call results in the submission of the list $\{(S_1), (S_2)\}$. Assume the scheduler makes (S_2) available first. As a result, the hash-build operation is partially completed. The scheduler is informed of the completion of subquery (S_2) (so it can uncache S_2 if needed) and a blocking request is made to get the next subquery. When the scheduler makes (S_1) ready, the rest of the hash-build operation is completed and the scheduler is informed of its completion. Next, the `SQ-list` $\{(U_1, T_1), (U_1, T_2)\}$ on the topmost schedule node is submitted. Assume both the subqueries are scheduled together. All data required by the plan-tree is now available. Hence, execution of the query is completed by pipelining the hash-probe and nest-loop operations.

The above scheme requires certain caution when scheduling *multiple* join subqueries together to avoid repetition of the following form: Consider the $R \bowtie S$ example of Section 3.1. Following the above scheme we first submit the `SQ-list` $\{(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2), (R_3, S_1), (R_3, S_2)\}$ to the scheduler. Suppose the scheduler next makes (R_1, S_1) ready, the executor finishes processing (R_1, S_1) and asks for the next set of ready subqueries. Suppose the next set of scheduled subqueries is $\{(R_1, S_2), (R_2, S_1), (R_2, S_2)\}$. To execute these three subqueries, scan-nodes of fragments R_1, R_2, S_1 and S_2 will be marked `available`, and the plan-tree will be processed as usual. But, by doing so, we have *repeated* the execution of subquery (R_1, S_1). To avoid such repetitions, the scheduler keeps track of subqueries already executed and uses this information for scheduling subqueries.

3.3 Handling Dependencies

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

1. First, we augment the plan-tree structure further with a special schedule node called the *resolve node*. The resolve node is added during the extraction phase immediately above the plan-tree node that introduces dependency between fragments. The resolve node, like the schedule node, contains a list of subqueries (`SQ-list`) that need to be executed first to resolve the dependencies. For instance, for an index scan, the resolve node is added immediately above the corresponding combine node and the `SQ-list` is the list of index trees on the indexed fragments. The `SQ-list` of the first schedule node above this resolve node cannot be established and hence is marked `unresolved`.
2. Next, we process nodes that introduce dependency in two stages: in the first stage a "ResolveDependency" call is made to compute the dependant list of subqueries and in the second stage after the subqueries are scheduled the rest of the node is processed. For instance, for the index scan node in the "ResolveDependency" stage the index tree is scanned and the list of matching TIDs sorted to get the list of blocks that needs to be fetched. In the second stage, after these blocks are fetched we complete the rest of index scan.

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

Nested loop join with runtime index on the inner relation: We demonstrate how to execute the hybrid join algorithm [8], which is an improvement over the standard nest-loop join.

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

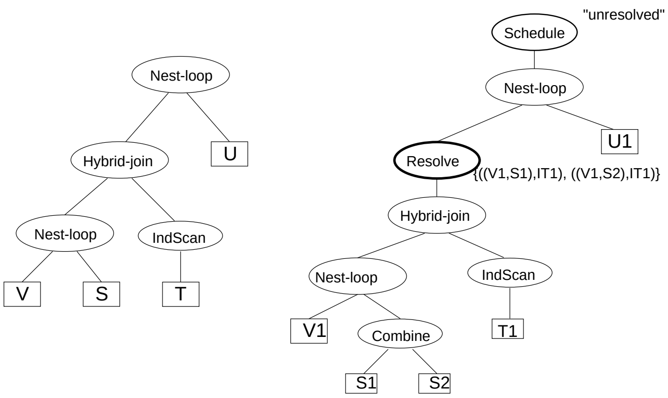


Figure 4: A plan-tree with dependency. The right-hand side is the plan-tree after the fragmentation and extraction phase. The sequential scan nodes on fragments have not been shown for clarity.

We adapt this algorithm to our framework. In the extraction phase, we add a resolve node above the hybrid join node as shown in the example of a four-way join in Figure 4. The `SQ-list` of the resolve node is a cross product of two subquery lists: one from the outer branch of the join node ($\{(V_1, S_1), (V_1, S_2)\}$) and the other from the inner branch ($\{(IT_1)\}$, the index tree of T_1). The schedule node above this resolve node is marked `unresolved`.

During execution, when "resolve-subquery" call is made to the resolve node, we submit the stored `SQ-list` ($\{((V_1, S_1), IT_1), ((V_1, S_2), IT_1)\}$) to the scheduler and wait for "ready" subqueries. When some set Q of subqueries are "ready", we use the hybrid join algorithm to get the list of blocks of the inner fragments. In our example, if $((V_1, S_1), IT_1)$ is "ready", we construct the hybrid join table using index tree, IT_1 and notify the scheduler of the completion of this subquery. Later, when $((V_1, S_2), IT_1)$ is "ready", we complete the join table. When all subqueries in the `SQ-list` are executed, we extract the list of blocks of the inner fragments that needs to be fetched. This completes the ResolveDependency call and we return the list to the schedule node above. The schedule node above the resolve node can then construct its `SQ-list` and execution proceeds as usual. In our example, the `SQ-list` of the schedule node is $\{(\mathrm{BL}(T_1), U_1)\}$ where $\mathrm{BL}(T_1)$ denotes the list of qualifying blocks of T_1. When this subquery is scheduled, the join is completed using the in-memory join table. The result tuples are pipelined to the Nest-loop join on U_1.

Index scans: Index scans are just a special case of the above nest-loop joins and can be handled in a similar manner.

Large objects: To support reordering between large object accesses of different tuples, we add a resolve node after the node that accesses the large objects. The `SQ-list` is derived from the plan tree underneath this node. For example in Figure 5, a restrict clause on a large object is above the join node between R and S. Therefore, the `SQ-list` $\{(R, S)\}$ is stored in the resolve node.

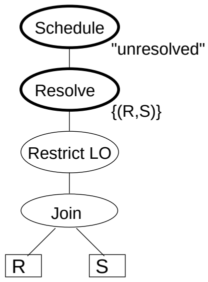


Figure 5: Adding resolving nodes for large object access.

During the resolution phase, we submit the `SQ-list` to the scheduler and when the subquery is "ready" a ResolveDependency call is made to the node accessing the large object. During this call, the join is completed and the resultant tuples along with the IDs of large objects required by them are collected in an in-memory table (like for the nest-loop join above). The list of large objects is then returned. The schedule node submits this collected list to the scheduler. The scheduler fetches the large objects in an efficient order. When a large object is "ready" the corresponding tuple is processed further and the scheduler is notified of its completion. Cases where whole of the large object is not needed will require modification of the function that selects the part to be fetched. The execution of the function has to be split into two phases, where in the first phase the function selects the blocks of the large object to be fetched (like in index scans) and in the second phase the function actually processes the data.

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

3.4 Preventing reordering failures

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

4 Performance evaluation

The architecture described in this paper is implemented on a DEC Alpha AXP workstation running Digital UNIX (OSF/1 V3.2). It is a modification of the Postgres [32] database system that was extended with a multi-threaded scheduler and the I/O process as described in Section 2. The user processes are the original Postgres backends, modified to support the new nodes and the fragmentation and extraction procedures. The user and I/O processes communicate with the scheduler using RPCs. The scheduler maintains as many I/O processes as the number of drives in the tertiary memory device to allow parallel data transfer from all the drives. To facilitate measurements on robots which were unavailable, we implemented a tertiary memory device simulator. The simulated device used a magnetic disk for data storage but serviced I/O requests with the same approximate delay as an actual device. This also allowed us to vary critical parameters like the number of drives, switch time etc for measuring sensitivity of our results to these parameters.

For our experiments, we used a 512 MB local magnetic disk drive as a cache. This cache size is arguably smaller than the cache expected to be used by production systems storing terabytes of data on tertiary memory. However, because of the practical inconvenience of loading huge datasets and running multiple experiments on them, we have scaled down the size of the cache, the size of the datasets and the number of concurrent users proportionately. The size of a storage block was set to 256 KB since this was the size used by the original Postgres storage manager for staging data from tertiary memory devices [26]. Each tuple of a relation consisted of ten integer fields that enable selection based on different selectivities (as in the Set Query Benchmark [25]) and a text field that is used to pad each tuple to a total (internal) size of 300 bytes. We ran a series of experiments to compare the following three approaches for processing queries:

- NoPrefetch where data is fetched in units of a storage block (256 KB) on demand and no prefetching whatsoever is used.
- Prefetch where we use both sequential prefetch (for sequential scans) and list prefetch (for index scans). The size of the prefetch unit was set to 32 storage blocks (8 MB), which is used in some database systems that use prefetching [27].
- Reorder which is our scheme of reordering execution as described in this paper.

We start with a few anecdotal cases of simple scan queries (Section 4.1). Often, more useful insights can be obtained by running particular query instances

	tape stacker	Magneto-Optical jukebox
switch time (sec)	30	14
transfer rate(MB/sec)	2	0.5
seek rate (MB/sec)	200	-
seek startup (sec)	2	0.3
number of drives	1	2
platter size (GB)	10	1.3 (both sides)
number of platters	10	32

Table 1: Tertiary Memory Parameters: The switch time is a summation of the average time needed to rewind any existing platter, eject it from the drive, move it from the drive to the shelf, move a new platter from shelf to drive, load the drive and make it ready for reading.

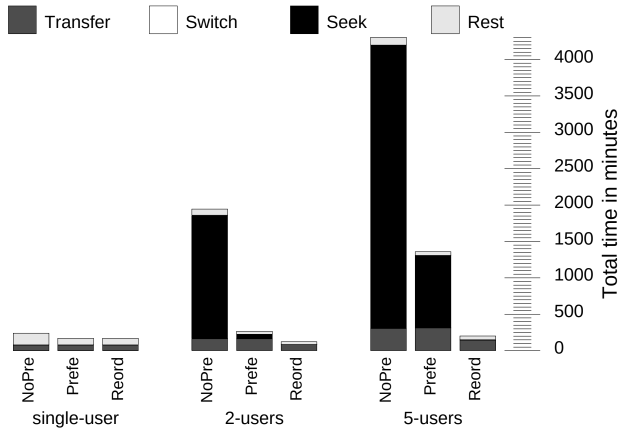


Figure 6: Difference in total execution time for three methods (NoPrefetch, Prefetch, Reorder) with sequential scans. "Rest" refers to the part of the total query processing time not spent in doing tertiary memory I/O.

where it is easy to analyze where and why one approach performs better than the other. Later, we report measurements on a mixed multi-user workloads (Section 4.2) to evaluate average case performance. Finally, in Section 4.3 we measure the overhead of scheduling.

4.1 Simple scan tests

We first did a set of experiments on a simulated tape stacker (Table 1) involving only sequential and index scans to demonstrate some of the basic cases where reordering is effective. Our objective is to show how conventional query processing techniques, although acceptable for single user queries perform badly when multiple users interact.

The first set of experiments are with a single user. In Figure 6(a) we show the total time taken to process the sequential scan with the three schemes: NoPrefetch, Prefetch and Reorder. We also show the part of the total time spent in data transfer, platter switch and seeks on tertiary memory. We note that the Prefetch and Reorder schemes are 20% better than NoPrefetch. This is mainly due to I/O-CPU overlap. The total I/O done is the same in all three schemes but NoPrefetch does not enable effective overlap between I/O and CPU.

We then let two users run the same scan query, the second user submitted the query after the first one had scanned just more than 512 MB of the relation. The total time in all our multi-user experiments is defined as the time between the submission of the first query and the time when the answer to the last query is returned. As shown in Figure 6(b), the total time with Reorder is one-fifteen of NoPrefetch and less then one-half of Prefetch. With Reorder, the second user started the scan from the remaining part of the relation instead of the beginning as in the other two schemes. Thus, both users synchronized their processing perfectly, so that they processed the same data blocks at the same time. In contrast, with Prefetch the second user had to re-fetch every data block since the cache can only hold 512 MB.

We next repeated the query with five users to measure how these results scale. Each user submitted its query after the first one had scanned somewhere between one-tenth to one-half of the entire relation (selected randomly). In this case, Reorder takes almost one-fifth the time taken by Prefetch. By synchronizing the scans of the different users, Reorder not only makes better use of cached data, it also incurs smaller seek cost. For Prefetch almost 80% of the total time is spent in seeks whereas for Reorder the seek cost is negligible. We expect this trend to continue as we increase the number of users and stagger their scans such that simple LRU based cache replacement policies cannot ensure proper reuse of cached data.

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

4.1.1 Index scans

In this experiment, we report the performance of unclustered index scans.

We used two 25 GB relations spread across 5 different tapes in units of 5 GB each. The first relation was stored on tapes 1 through 5 and the second on tapes 2 through 6. The fragment size was again 256 MB. The indices reside on magnetic disk. The selectivity of the index scan was 0.01%. In Table 2[2] we show the performance of a single-user index scan. NoPrefetch is almost two orders of magnitude worse than the other two schemes because it does too many random I/Os. Since the index scan is unclustered, each block access could result in an I/O request to any of the five tapes of the tertiary memory. This leads to high platter switch and seek overhead. Schemes Prefetch and Reorder convert the unclustered I/O to clustered I/O by pre-scanning the index tree, sorting the qualifying TIDs and fetching the data blocks in their storage order. This results in significant reduction in the number of platter switches and the the seek cost.

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

Table 2: Difference in total execution time with index scans.

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

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

[2] In the conference version of the paper, there was an experimental error in the timings reported in Table 2, column 5 (seek time and hence total time). This is the corrected table. I would like to thank Professor David Dewitt for asking questions that lead me to find the mistake.

Description	Default
Workload	
# queries per user	5
# users	3
% of 2-way join queries	50
% index scans	80
Index selectivity	0.1-10%
# of relations	10
Relation size	100 MB to 10 GB (Uniform distribution)
Fragment size	$\leq$ 85 MB ($\frac{1}{6}th$ cache size)
Data layout	each relation stored from 1 to 5 platters

Table 3: Experimental setup for mixed workload.

switches incurred is almost two orders of magnitude more with PREFETCH than with REORDER.

This experiment demonstrates that statically reordering index scans reduces random I/O considerably for single user index scans. But, with multiple users static reordering is not sufficient for reducing random I/O. Summarizing, the sequential example showed how the amount of data transferred can be reduced by doing better scheduling of queries that share data accesses. The index scan example showed how the number of platter switches can be reduced by doing better scheduling of queries that share common platters.

4.2 Multiuser-mixed workload tests

Next, we used a mixed multi-user workload of 2-way joins and selects to identify conditions where reordering pays-off and where it does not by taking measurements under different configurations of cache sizes, number of drives, etc. We also report measurements on a real HP magneto-optical jukebox (performance characteristics summarized in Table 1) that is connected to our prototype[3]. Table 3 summarizes the details of experimental setup.

In Figure 7(a) we plot the total time for this workload on the tape-jukebox and the MO-jukebox with one drive each[4]. On the tape-jukebox, the total time with PREFETCH is about one-fifth of NOPREFETCH while REORDER is one-seventh of PREFETCH. On the MO-jukebox, the total time with PREFETCH is about one-third of NOPREFETCH and REORDER is about one-third of PREFETCH. For both NOPREFETCH and PREFETCH, the execution time is dominated by I/O on tertiary memory unlike in our reordering scheme. As shown in Figure 7(a), the main I/O bottleneck is platter switches for both NOPREFETCH and PREFETCH. REORDER performs better since it greatly reduces the number of platter switches. For the MO-jukebox the platter switch cost is not as high as for the tape-jukebox. Therefore, we observe smaller relative gains with REORDER for the MO-jukebox.

Increasing the number of drives: Since the main bottleneck is platter switches, increasing the number of drives from 1 to 2 decreases the gap between the reordering and non-reordering based schemes as shown in Figure 7(b). For the two-drive case we plot only the total execution time since it is difficult to separately account for the time spent in doing various I/O activities, example data transfer on one drive might be overlapped with seeks on another. For REORDER there was negligible change in execution time when we increased the number of drives from 1 to 2 since the total execution time was not bound by tertiary memory I/O.

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

We observed that for REORDER there was no change in execution time due to increased number of drives since the total execution time was not bound by tertiary memory I/O. The performance of NOPREFETCH and PREFETCH improve until all required platters are always loaded. At this stage, the only gain with reordering is through reduction in seek and transfer cost.

Decreasing working set: For the experiments so far, the transfer cost incurred with all three schemes was not significantly different. One of the merits of our query scheduling policies is better reuse of the cached data. Therefore, we expected to observe significant reduction in transfer time too with REORDER. Closer inspection of the workload revealed that there was very little opportunity for reusing data since the degree of sharing between the three concurrent users was limited. Each of the three users picked at most two of the ten relations in the database with equal likelihood. Hence there was little chance of overlap between the component relations of queries running concurrently. To verify this claim, we repeated the 2-drive experiments, with five users instead of three and skewed the access requests so that 80% of the accesses go to 30% of the data. We observed that the transfer time for REORDER was almost one-half of that with PREFETCH for the skewed dataset (Figure 7(c)).

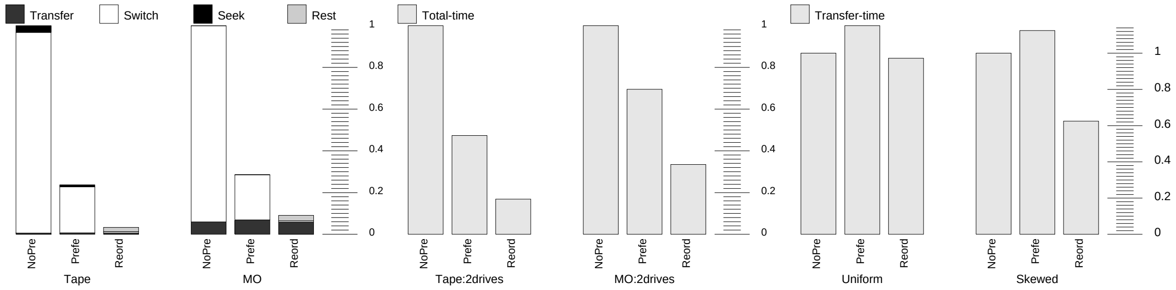
There experiments demonstrate that reordering is beneficial for tertiary memory databases either when the platter switch or seek costs are high or when the degree of sharing between queries is large.

4.3 Scheduling overhead

Finally, we measured the overheads of reordering in our prototype. For the experiments presented earlier, reordering has definitely paid off, whatever be the scheduling overhead. But an important question is

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

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



(a) Single Drive (b) Two Drives (c) Single Drive, Tape jukebox

Figure 7: Difference in total execution time for three methods (NoPrefetch, Prefetch, Reorder) using the mixed workload. The execution time is normalized by the time taken by scheme NoPrefetch to allow drawing on the same scale.

how well these benefits scale with increasing number of users or increasing number of fragments. The answer is crucially dependent on the scheduling overhead that we present next.

We measure the following overheads: (1) The per-fragment overhead that is directly proportional to the number of fragments in the query, e.g., the time to fragment a plan-tree. Measured as a percentage of the time to scan a cached fragment, this overhead was typically 0.06% (1.5 milliseconds). (2) the per-subquery overhead: e.g, the time spent in the extraction phase or in communicating with the scheduler. Measured as a fraction of the time spent in processing a two-way hash-join query on cached data, this overhead was typically 0.15% (5 milliseconds). (3) the per-session overhead e.g., time spent by the scheduler in deciding what subquery to schedule next. Unlike the previous two overheads this overhead depends on factors like the number of users concurrently active and the number of fragments per relation and can only be measured as a function of these factors. We plot this overhead as a function of number of users (1 through 9) and total number of fragments in the database (10 to 100) in Figure 8. The overhead per subquery increases only at a rate of 2 millisecond per additional user and less than 1/4th millisecond per additional fragment. The total overhead is thus measured to be typically less than 30 milliseconds per subquery and less than 1% of the total execution time.

5 Related Work

There are six areas of work that are relevant to the research presented here: prefetching, page scheduling for join execution, parallel query scheduling, multiple query optimization, dynamic query optimization and batching in OODBs.

Prefetching is useful both in operating systems [6, 19, 28] and database systems [33, 11, 2] especially when accompanied by execution reordering, e.g., list prefetch [23, 4, 8] used with index scans. Our system extends prefetching to entire plan trees and not simply to index scans. A significant difference is that, we can reorder based on dynamic conditions like cached data, the state of the I/O device and the data needs of other queries whereas existing prefetching techniques reorder execution based on static storage layout.

Page scheduling on page join graphs as discussed in [24, 22, 20] is an example of reordering two-way joins queries. However, their methods are specific to join queries and require implementation of new join algorithms — our method is meant to be a general scheme for reordering any node of a plan tree. For parallel [3, 5, 16, 34, 13] and distributed query scheduling [31, 7], plan trees have to be analyzed for establishing pipelining and ordering dependencies in a manner somewhat analogous to our subquery extraction step. However, our method is different in two ways: first, for efficiency reasons discussed in this paper (Section 3.1), we execute all subqueries from a single plan-tree whereas most parallel and distributed systems construct different plan-trees for subqueries to be scheduled on different processors and second, our model for communicating and synchronizing with the scheduler for deciding online the order in which subqueries are scheduled places a different set of requirements than on these systems.

Our technique is reminiscent of the way multiple query optimizers combine queries with common subexpressions [30]. [21] discusses policies for scheduling a batch of select and hash-join queries for sharing in-memory hash-tables. Queries are thus scheduled for execution in a data-driven manner the way we do. However, such optimizers typically schedule at whole relation level and do not consider reordering within a scan unlike our scheme.

Dynamic query optimization [4, 10] is another technique that involves plan tree modification at runtime. However, in contrast to our work, the emphasis in that area is on choosing dynamically from some fixed set of execution plans. Once the choice is made, execution proceeds in a fixed order.

In object oriented databases, the navigational nature of queries can lead to bad I/O performance mak-

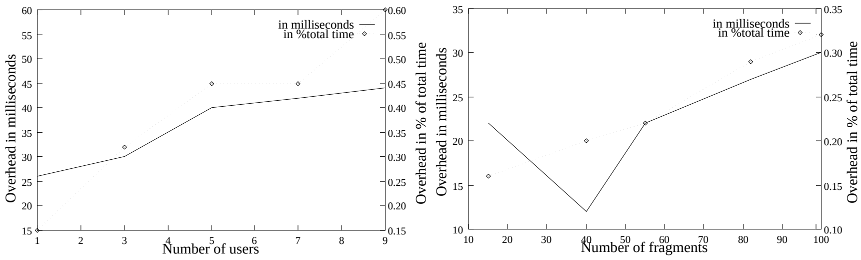


Figure 8: The per-session overhead as a function of the number of users and number of fragments. The y-axes are overhead in milliseconds per subquery (left) and overhead as a percentage of the total execution time (right).

ing it important to do prefetching [12] and batching [18]. [18] presents ways of modifying the plan-tree to replace object-at-a-time references with an assembly operator that collects multiple object references first and then reorders them to optimize I/O accesses. However the main difference between their scheme and ours is that, they cannot handle reordering across different operators of a plan-tree or across data reference of different users.

Another concurrent work on modifying query plans to reorder I/O access on tape is reported in [35]. They propose a scheme for pre-executing functions that access large objects so as to allow I/O requests of different large objects in the same tuple stream and across multiple users to be reordered. However, they do not allow the order of proccessing tuples to be modified unlike in our case.

Summing up, our distinction from related work is that we propose the first system that provides a general framework for reordering execution of plan trees dynamically in an extended relational database system.

6 Conclusion

In this paper, we have explored a simple, yet powerful, idea of reordering execution to tune to the optimal data fetch order. Existing methods of query execution provide but a limited flexibility of reordering data fetches during execution. Our proposal is based on the premise that in a multi-user environment when access latency of data varies widely, significant performance advantage can be gained by dynamically reordering execution.

We proposed a general framework for reordering all parts of the plan tree. For building a reorderable execution engine, we extended the plan tree data-structure with three new meta-nodes that are added in an extra phase between optimization and execution of the plan tree. These operators enable the executor to communicate and synchronize with the scheduler for ordering the execution of subqueries. Our changes are restricted only to these new operators and the extra phase and thus enable modular extension of existing execution engines. We extended the Postgres execution engine and used it for building a prototype of a tertiary memory database.

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

Our proposed general framework is applicable to other situations where tuning data to some external order of arrival is important, e.g., a broadcast disk-based mobile computing client. The data chunks can be determined by the pages broadcast together. The size of the data chunks is important for limiting the overhead of reordering. For our prototype, typical overhead per subquery was 30 milliseconds. Hence, as long as the processing time per subquery is much larger than this reordering can be used profitably. The scheduling unit would be responsible for watching the broadcast data stream, caching relevant data when appropriate and scheduling ready subqueries for execution.

Future work in the area should consider the impact of execution reordering on query optimization: executing queries in parts invalidates some of the assumptions and cost functions used by the optimizer. In this paper, index scans posed one such scenario. There are other issues specific to tertiary memory systems that need to be addressed: (1) estimating the access cost when some relations are stored permanently on disk and others on tertiary memory; (2) including the size of the disk cache in optimizing queries. When the disk cache is smaller than the relation, sorting is no longer an option. Another topic for future work is providing support for cancelling submitted subqueries to the scheduler when a restrict or a join node yields an empty result.

Acknowledgements

We would like to thank Professor Dr. Kurt Mehlhorn of MPI Informatik for allowing access to necessary resources in the last five days of preparing the camera-ready copy.

We would also like to thank the anonymous reviewers for their feedback.

References

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

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

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

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

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

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Application-controlled caching, prefetching and disk scheduling. Technical Report TR-493-95, Princeton University, 1995.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] M. Mehta, V. Soloviev, and D. Dewitt. Batch scheduling in parallel database systems. In *Proc. International Conference on Data Engineering*, pages 400–410, 1993.

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

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

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

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

[26] M. A. Olson. Extending the POSTGRES database system to manage tertiary storage. Master's thesis, University of California, Berkeley, 1992.

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

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

[29] S. Sarawagi. Query processing and caching in tertiary memory databases. In *Twenty first conference on Very Large Databases*, Sep 1995.

[30] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

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

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

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

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

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