Distributed and Parallel Databases 1 (1993), 9–32

# Optimization of Parallel Query Execution Plans in XPRS

WEI HONG AND MICHAEL STONEBRAKER {HONG,MIKE}@CS.BERKELEY.EDU
*Computer Science Division, EECS Department, University of California at Berkeley, Berkeley, CA 94720*

**Abstract.** In this paper, we describe our approach to optimization of query execution plans in XPRS, a multiuser parallel database system based on a shared memory multiprocessor and a disk array. The main difficulties in this optimization problem are the compile-time unknown parameters such as available buffer size and number of free processors, and the enormous search space of possible parallel plans. We deal with these problems with a novel two phase optimization strategy which dramatically reduces the search space and allows run time parameters without significantly compromising plan optimality. In this paper we present our two phase optimization strategy and give experimental evidence from XPRS benchmarks that indicate that it almost always produces optimal or close to optimal plans.

**Keywords:** parallel database systems, database machines, query processing, query optimization

## 1. Introduction

XPRS (eXtended Postgres on Raid and Sprite) is a multiuser parallel database system based on a shared memory multiprocessor and a disk array. The *shared everything* [20] environment of XPRS is shown in Figure 1 and an outline of the initial design of XPRS can be found in [21]. The underlining reliable disk array RAID is described in [14].

Shared memory multiprocesors are a very cost-effecive way to achieve high performance. Although they have their limitation in scalability, they have two major advantages over the *shared nothing* architecture [20] adopted by most other parallel database machines such as GAMMA [7] and BUBBA [5]. First, there are no communication delays because messages are exchanged through shared memory, and synchronization can be accomplished by cheap, low level mechanisms. Second, load balancing is much easier because the operating system can automatically allocate the next ready process to the first available processor. Simulation results in [1] show that the potential win of a shared memory system over a shared nothing system to be as much as a factor of two, given the same number of processors, disks and megabytes of main memory. Moreover, shared memory multiprocessors can also be used as nodes in a shared nothing architecture to reduce the total number of nodes and achieve higher performance. Therefore, it is important to explore how to take advantage of shared memory multiprocessors for parallel query processing.

Database applications are often I/O intensive. In order to keep up with the

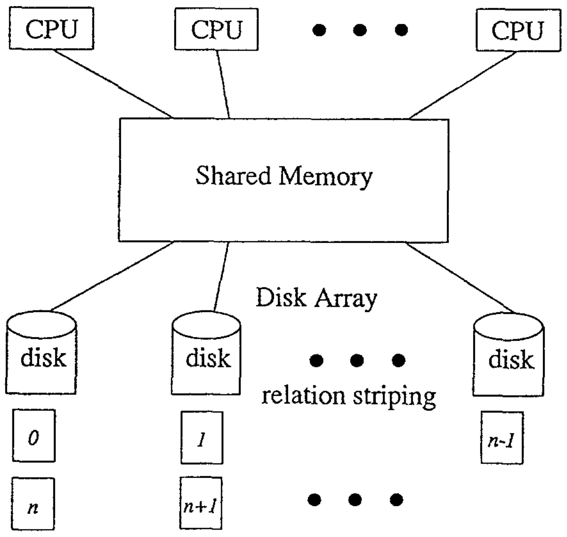


*Figure 1.* The parallel environment of XPRS.

I/O requests from multiple processors, XPRS uses a disk array to eliminate the I/O bottleneck. All relations are striped sequentially, block by block, in a round-robin fashion across the disk array (as shown in Figure 1) to allow maximum I/O bandwidth. Currently, we are simply using a nonredundant RAID 0 [14] configuration. Performance implications of running XPRS on other RAID configurations are discussed in [10].

In this paper, we address the problem of parallel query optimization in a shared everything environment and describe the XPRS approach. We will concentrate on two specific issues, namely dealing with compile-time unknown parameters and the large search space of parallel plans. In a multiuser environment, parameters such as the amount of available buffer space and number of free processors are unknown at compile time. Therefore, compile-time optimization must generate plans for an uncertain run-time environment. Second, the number of possible parallel query execution plans is so enormous that any exhaustive search algorithm is impractical. As a result, the search space must be heuristically reduced.

Our strategy is to divide query optimization into two phases. The first phase only optimizes sequential query execution plans with fixed parameters and is performed at compile time. The second phase is performed at runtime and finds the optimal parallelization of the optimal sequential plan chosen in the first phase based on the run-time environment. Obviously this two phase optimization approach greatly reduces the plan search space because it only explores parallel versions of the best sequential plan. In his paper, we will present experiment results that show that this approach does not compromise optimality of the resulting parallel query execution plan.

Most previous work on parallel query optimization has been done for a shared nothing environment, e.g., [16] and [17]. Earlier work on a shared memory environment (e.g., [15] and [12] ) only considers single operations, mainly joins and sorts. On the other hand we address the optimization of entire queries. Bultzingsloewen outlines in [4] six key issues in query optimization in loosely or tightly coupled multiprocessors with private disks and sketches the optimization strategy that is being implemented for the KARDAMOM database machine. However, the paper does not give any details about how to cut down the size of the search space of parallel query plans. Murphy and Shan propose in [13] an algorithm to parallelize a given sequential plan to achieve minimum duration time with computational resource requirements less than the given system bounds in a shared memory environment. It does not address the problem of how to choose a sequential plan to parallelize, and it assumes that the amount of available resources is known and therefore the algorithm cannot be used at compile time for a multiuser environment. In this paper, we will describe a way to handle unknown parameters at compile time and strategies to choose optimal sequential plans and their parallelizations.

Graefe and Ward propose in [8] a general approach for dealing with unknown parameters in compile-time optimization by introducing *choose-plan* nodes to generate multiple query plans, consequently, the run-time system must go through a decision tree to choose a plan according to the current system parameters. Although this general approach can be applied to XPRS, our solution is much simpler because we are concerned only with unknown buffer sizes and number of processors. Cornell and Yu show in [6] through an example of a three-way nestloop joins for which the optimal sequential query plan may change if the buffer size changes. However, the example does not consider hashjoin as a join algorithm. In XPRS, on the other hand, we always assume that there is enough main memory (approximately the square root of the size of the smaller join relation [19]) to use a hashjoin algorithm, which has been shown to be the best join method without the use of indices [19]. Our results show that with sufficient main memory to support hashjoins, the choice of the optimal query plan remains very stable with varying buffer sizes. Zeller and Gray point out in [22] that inaccurate estimates of intermediate result sizes can seriously jeopardize the performance of hashjoins. More generally inaccurate estimates can cause any query plan to have a running time substantially different than that predicated by the optimizer. In this paper, we assume accurate estimates for all intermediate results. It is left as a future research topic to explore the impact of uncertainty.

The rest of this paper is organized as follows. Section 2 explores the search space of parallel query execution plans and shows the complexity of the problem. Section 3 defines the optimization problem and presents our two phase optimization strategy along with the hypotheses that it is based on. Section 4 then discusses the performance of various parallelizations of a sequential plan and presents the overall architecture for parallel query processing in XPRS. Section 5 then describes some experiments that we have performed on XPRS to verify the hypotheses that our two phase optimization strategy is based on and shows that they are in general true with only small errors. Last, Section 6 concludes the paper and suggests future research.

## 2. The space of parallel plans

In a uniprocessor environment, a query execution plan (which we call a *sequential plan*) is a binary tree consisting of the basic relational operation nodes. In XPRS, the basic operations include **sequential scan, indexscan, nestloop join, mergesort join,** and **hashjoin.** At run time, the query executor processes each plan sequentially in a postorder (depth-first) sequence. Intermediate result generation is avoided by the use of pipelining, in which the result tuples of one relational operation are immediately processed as the input tuples of the next operation. Figure 2 gives an example of a sequential plan for a fourway join, $A \bowtie B \bowtie C \bowtie D$. The shaded boxes in Figure 2 represent plan fragments in a possible parallelization of the sequential plan, which we will discuss momentarily. Notice that the inner relation (the right subtree) of the hashjoin at the root of the plan tree is also a join. This kind of plans are called *bushy* tree plans. Most conventional query optimizers [18] only consider *left-deep* tree plans that do not allow inner relations to be a join in order to reduce the search space of possible plans.

We call query execution plans that specify a parallel execution of a query *parallel plans*. Obviously, each parallel plan is a *parallelization* of some sequential plan and each sequential plan may have many different parallelizations. Parallelizations can be characterized in the following three aspects.

### 2.1. *Form of parallelism*

There are two forms of parallelism that we can exploit in query processing: *intraoperation* parallelism and *interoperation* parallelism. Intraoperation parallelism is achieved by partitioning data among multiple processors and having those processors execute the same operation in parallel. Interoperation parallelism is achieved by partitioning the query plan and executing different operations in parallel. As we will discuss in Section 4.1, all of the basic relational operations can be parallelized with intraoperation parallelism in a straightforward manner. Moreover, if we have a bushy-tree plan, we may have two operations that do not depend on each other's output. Therefore they can be executed concurrently with interoperation parallelism. For example, the sequential plan in Figure 2 can be parallelized with intraoperation parallelism in each node and interoperation parallelism between $A \bowtie B$ and $C \bowtie D$.

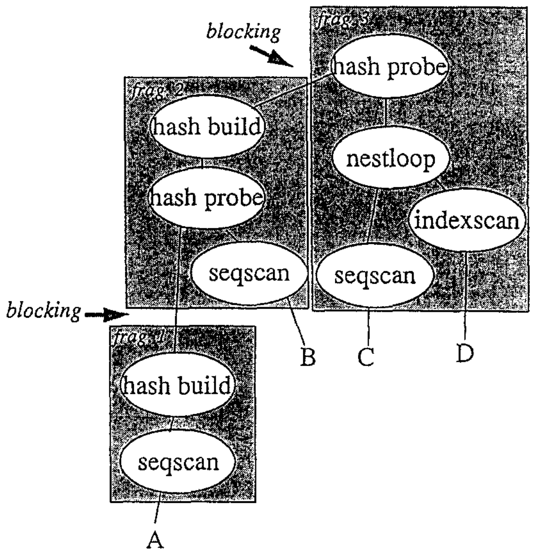


*Figure 2.* Example query execution plan.

## 2.2. *Unit of parallelism*

*Unit of parallelism* refers to the group of operations that is assigned to the same process for execution. We also call a unit of parallelism a *plan fragment* since it is a "fragment" of a complete plan tree. Each plan fragment will be executed in a pipelined fashion, and consequently should not contain any "blocking" between operations. Blocking happens for certain operations, such as hash and sort, when one operation has to wait for another operation to finish producing all the tuples before it can proceed. Plan fragments should be chosen so that such blockings only occur at plan fragment boundaries. For example, the shaded boxes in Figure 2 show that the sequential plan has three plan fragments with blocking only at the boundaries.

## 2.3. *Degree of parallelism*

*Degree of parallelism* is the number of processes that are used to execute a plan fragment. Presumably we want the degree of parallelism to be as high as possible; however, excessive parallelism may cause high resource conention, resulting in a loss of performance as will be shown in Section 4.1. An optimal degree of parallelism must be decided according to run-time system resource availability to achieve maximum performance.

In this paper only intraoperation parallelism and leftdeep tree sequential plans are considered for the following reasons:

- It is much easier to achieve load balancing with intraoperation parallelism because we can always partition the input data carefully into equal size portions. Load balancing becomes much harder with interoperation parallelism because different operations may have very different complexity and different sizes of input data.
- Intraoperation parallelism requires less main memory. For example, if we want to execute two hashjoins with interoperation parallelism, we need to allocate two hash tables, while intraoperation parallelism only requires one hash table at a time.
- Interoperation parallelism may turn sequential I/Os into random I/Os. For example, if we run two sequential scan operations in parallel, the disks may seek back and forth between the data of the two operations and therefore can only provide random I/O bandwidth. On the other hand, if we run the two operations one at a time, each operation may have sequential I/O bandwidth.
- Interoperation parallelism can only help mix up CPU-bound and I/O-bound operations to achieve better resource utilizations. A more detailed discussion on this subject is presented in [9]. Based on the results in [9], the optimization strategy that we are presenting in this paper can be easily extended to also deal with bushy-tree plans and inter-operation parallelism.

## 3. Optimization of parallel plans

This section defines the optimization problem, presents our two phase optimization strategy and the hypotheses that it is based on, and gives our intuition behind the hypotheses.

### 3.1. The optimization problem

The overall performance goal of a parallel database system is to obtain increased throughput as well as reduced response time in a multiuser environment. The objective function that XPRS uses for query optimization is a combination of resource consumption and response time as follows:

$$cost = resource\_consumption + w \times response\_time.$$

Here $w$ is a system-specific weighting factor. When $w$ is small, we mostly optimize *resource_consumption*. When $w$ is large, we mostly optimize *response_time*. *resource_consumption* is measured by the number of disk pages accessed and number of tuples processed, while *response_time* is the elapsed time for executing the query.

Our optimization problem is to find the parllel plan with the minimum cost among all possible parellelizations of all possible sequential plans of query. Suppose $P$ is a sequential plan and let $PARALLEL(P)$ be the set of possible parallelizations of $P$. Suppose $Q$ is a given query and let $SPLAN(Q)$ be the set of sequential plans of $Q$. Then $PPLAN(Q)$, the set of parallel plans of $Q$, is given by

$$PPLAN(Q) = \bigcup_{P \in SPLAN(Q)} PARALLEL(P).$$

$PPLAN(Q)$ is the search space to explore for intraquery parallelism.

In a multiuser environment, many system paramenters that affect query execution cost are unknown at compile time. In this paper, we specifically consider two of such parameters: available buffer size, $NBUFS$ and number of free processors, $NPROCS$. Buffer size not only affects the buffer hit rate but also determines the number of batches in a hashjoin [19]. The number of free processors determines the possible speedup of query execution. For ease of exposition, we assume that the buffer size and number of free processors are fixed during the entire query execution in this section. As we will describe in Section 4.2, our operational prototype only fixes these parameters during the execution of individual plan fragments.

For $PP \in PPLAN(Q)$, let

$$Cost(PP, NBUFS, NPROCS)$$

be the cost of the parallel plan $PP$ given $NBUFS$ units of buffer space and $NPROCS$ free procesors. Our optimization problem is to find,

$$min\{Cost(PP, NBUFS, NPROCS) | PP \in PPLAN(Q)\}.$$

There are two major difficulties in this problem. First, the search space $PPLAN(Q)$ is orders of magnitude larger than $SPLAN(Q)$, therefore query optimization by exhaustive search [18] is impractical. Second, the dynamic parameters, $NBUFS$ and $NPROCS$ are unknown until query execution time, therefore compile-time optimization must deal with these unknown parameters. Our goals are to reduce the search space of parallel plans by heuristics and to perform as much compile-time optimization as possible.

### 3.2. *Two-phase optimization*

We achieve our goals by the following two-phase optimization strategy.

Let $BPP(Q, NBUFS, NPROCS)$ be the best parallel plan for query $Q$ given $NBUFS$ units of buffer space and $NPROCS$ free processors, $BP(Q, NBUFS)$ be the best sequential plan for $Q$ given $NBUFS$ units of buffer space, and

*Cost*(*P*,*NBUFS*) be the cost of a sequential plan *P* given *NBUFS* units of buffer space.

- *Phase 1.* Find the optimal sequential plan assuming that the entire buffer pool is available, i.e., find *BP*(*Q*,*ALLBUFS*) where *ALLBUFS* is the size of the whole buffer pool.
- *Phase 2.* Find the optimal parallelization of the optimal sequential plan, i.e., find

  $$min\{Cost(PP,NBUFS,NPROCS) \mid PP \in PARALLEL(BP(Q,ALLBUFS))\}$$

  where *NBUFS* and *NPROCS* are the run-time available buffer size and number of free processors.

Because we have a fixed buffer size *ALLBUFS* in Phase 1, it can be performed at compile time. Phase 2 still has to be performed at run time, because it takes the run-time parameters *NBUFS* and *NPROCS* into account and tries to dynamically determine the best parallelization of the sequential plan chosen in Phase 1. As we can see, this two phase optimization strategy overcomes the difficulties in our optimization problem, because it provides a nice partitioning between compile-time optimization and run-time optimization and significantly reduces the search space by restricting to the parallelizations of one particular sequential plan. The question is how well this strategy works. The effectiveness of this strategy is based on the following two hypotheses that we made for XPRS.

**The Buffer Size Independent Hypothesis.** *The choice of the best sequential plan is insensitive to the amount of buffer space available as long as the buffer size is above the hashjoin threshold, i.e.,*

$$Cost(BP(Q,NBUFS),NBUFS) \approx Cost(BP(Q,NBUFS'),NBUFS),$$

*where* $NBUFS \neq NBUFS', NBUFS \geq T, NBUFS' \geq T$, *and T is the hashjoin threshold.*
As we will discuss in details in the next subsection, certain exceptions to the above hypothesis do exist, however, they can be localized within specific operations and handled with a mechanism that dynamically chooses the implementation of an operation at run time according to real buffer sizes.

**The Two Phase Hypothesis.** *For a shared everything environment where only intraoperation parallelism is exploited, the best parallel plan is a parallelization of the best sequential plan, i.e.,*

$$BPP(Q,NBUFS,NPROCS) \in PARALLEL(BP(Q,NBUFS)).$$

We will verify these two hypotheses with experiment results in the Section 5.

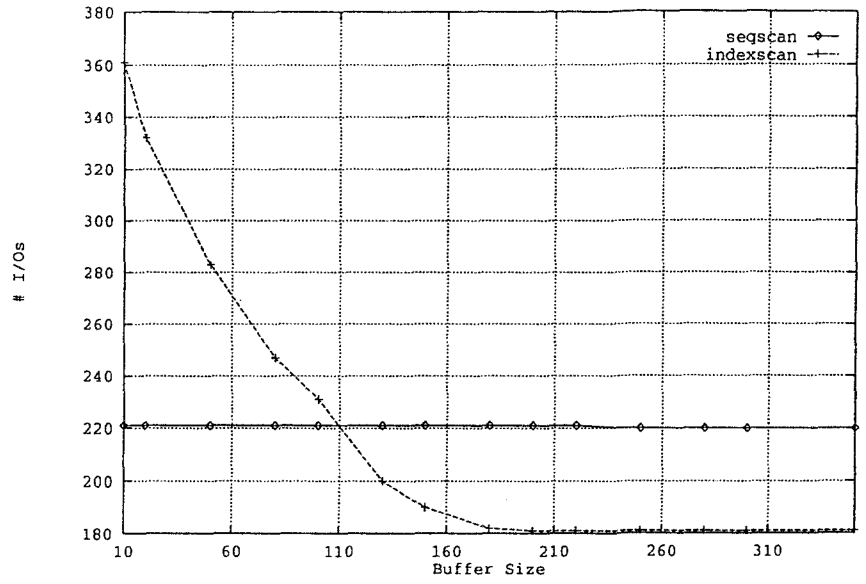


*Figure 3.* Cost of SeqScan vs. IndexScan.

## *3.3 Introduction of choose nodes*

We have identified two situations in the execution of single operations that may cause problems with the buffer size independent hypothesis. One is choosing between an indexscan using an unclustered index and a sequential scan. The other is choosing between a nestloop with an indexscan over the inner relation and a hashjoin. We will show these problematic cases by plotting the execution costs of some sample queries against varying buffer sizes. In general, the cost of a sequential plan is measured by resource consumption [18], which is a linear combination of I/O cost and CPU cost as follows,

$$Cost = \#page\_io + c \times \#tuples\_processed.$$

Here $c$ is another system-specific weighting factor. Since buffer sizes only affect the I/O cost of query execution, we will only plot the I/O costs (i.e., number of page I/Os) of query executions in the examples below.

***3.3.1. Sequential scan versus index scan.*** A sequential scan only needs one buffer page and additional buffer pages do not reduce query execution cost. However, the cost of an indexscan using an unclustered index is very sensitive to buffer size. If there is enough buffer space to hold all the pages that need to be fetched during the indexscan, then we only need to read each of those pages once. Therefore, with sufficient buffer space, if an indexscan touches less than all pages, it will have lower cost than a sequential scan. On the other hand, with few buffers, an indexscan may end up fetching the same page from disk many times and becomes more expensive than a sequential scan. Figure 3 gives an example of this situation by plotting the cost of each plan versus buffer size for a selection query on a 10,000-tuple relation from the Wisconsin benchmark. The I/O costs of this query are measured from real executions of this query on XPRS configured with different buffer sizes. Obviously, when the two curves cross, we require a mechanism to switch from a sequential scan to an indexscan, or vice versa. Note that this situation does not always happen. In most cases, the two curves are far apart and do not cross each other. This situation only arises for a certain selectivity range which causes these two curves to be close together. The example query in Figure 3 is carefully selected to illustrate this situation.

***3.3.2. Hashjoin versus Nestloop.*** A similar situation occurs in choosing between a nestloop with an indexscan over the inner relation and a hashjoin. With sufficient buffer space, a nestloop with indexscan will fetch ultimately all the pages in the outer relation and all the pages in the inner relation that match some tuple in the outer relation plus a small number of index pages. On the other hand, a hashjoin will need to fetch all the pages in both the outer and inner relations. If the join selectivity is small, the nestloop plan may not need to fetch all the pages in the inner relation and therefore will have a lower cost than the hashjoin plan. On the other hand, with few buffers, the indexscan in nestloop may have to fetch the same page many times and cause the nestloop to become more expensive than a hashjoin. Figure 4 shows an example of a "crossover" point between nestloop with an indexscan and hashjoin of another carefully selected join query between two Wisconsin benchmark relations. Again, this situation only occurs for a certain range of join selectivities.

To solve these two "crossover" problems, we introduce two new *choose* nodes in the query tree, which can choose between the two join or two scan methods depending on which side of the cross over point the run-time buffer size belongs to. Now we modify the buffer size independent hypothesis as follows.

**Modified Buffer Size Independent Hypothesis.** *The choice of the best sequential plan with choose nodes is insensitive to the amount of buffer space available as long as the buffer size is above the hashjoin threshold, i.e.,*

$$Cost(BCP(Q,NBUFS),NBUFS) \approx Cost(BCP(Q,NBUFS'),NBUFS),$$

*where $BCP(Q,NBUFS)$ is the optimal sequential plan of $Q$ under buffer size $NBUFS$ with choose nodes, $NBUFS \neq NBUFS', NBUFS \geq T, NBUFS' \geq T$, and $T$ is the hashjoin threshold.*

## *3.4. Intuition behind hypotheses*

Let us first consider the buffer size independent hypothesis for single operations.

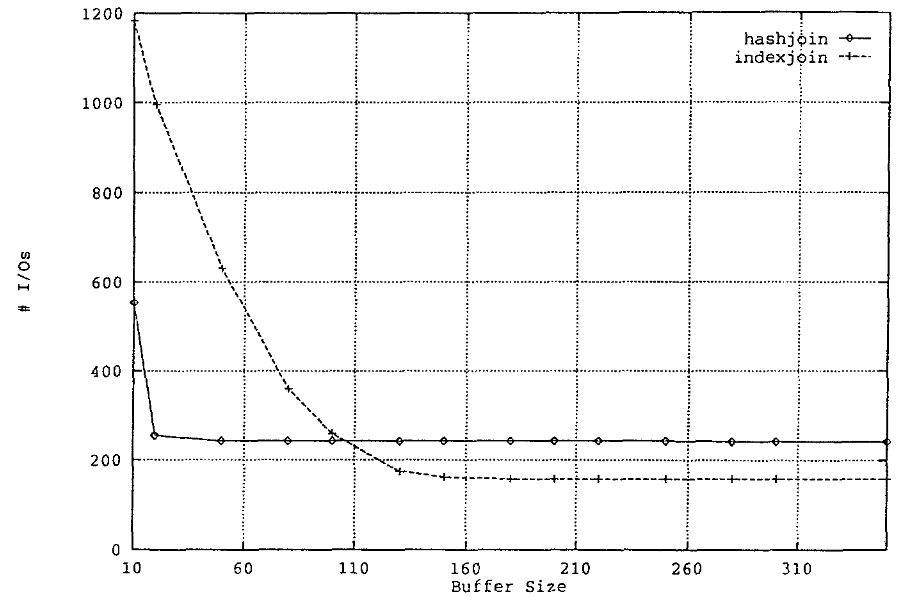


*Figure 4.* Cost of Nestloop with index vs. Hashjoin.

If there are no indices that can be used, the hypothesis obviously holds, because only a sequential scan plan is possible for scans and a hashjoin plan is always optimal for joins as long as the buffer size is above the hashjoin threshold [19]. Obviously, requiring the buffer size to be above the hashjoin threshold is important, because otherwise the hypothesis will not even be true for two-way join queries. If there are indices defined on a selection or join attribute, usually we want to take advantage of the indices to form plans that involve index scans, and the two problematic cases described above may occur. However, in this situation, we can encapsulate all the necessary plan switching in a choose node. Therefore, with choose nodes, the buffer size independent hypothesis is guaranteed to hold for all single operations. Furthermore, we postulate that this remains to hold for complete query plans consisting of one or more operations. Section 5.2 will show that this is in general true with only small errors.

The two phase hypothesis only holds for a shared everything environment and only for intraoperation parallelism. In a shared nothing environment, we also need to consider communication cost. Thus, the hypothesis may become false because the optimal sequential plan may incur excessive communication cost in its parallelization and therefore its parallelization can only be a suboptimal parallel plan. Moreover, if one of the subtrees of a bushy tree plan is CPU-bound and the other is IO-bound and we allow interoperation parallelism, even though this bushy tree plan may not be the optimal sequential plan, its parallelization may be cheaper than the parallelization of the optimal sequential plan because it can mix up CPU-bound tasks and IO-bound tasks to achieve better resource utilization [9]. On the other hand, if we only consider a shared everything environment and intraoperation parallelism, as we will show in Section 4.1, our parallelization of sequential plans does not incur any extra resource consumption. Since the optimal sequential plan has minimum resource consumption, its parallelization also has minimum resource consumption. Hence, the two-phase hypothesis is trivially true if we only optimize resource consumption. For response time, we will also show in Section 4.1, each sequential plan can achieve a near-linear speedup through intraoperation parallelism. Since the optimal sequential plan is also the fastest sequential plan, its parallelization will remain the fastest. Although this is not always true, Section 5.3 will show that the errors are very small.

## 4. XPRS query processing

In this section, we first describe the parallelization of individual operations in XPRS along with their performances. Then we present the overall architecture of XPRS query processing.

### *4.1. Implementations of intraoperation parallelism and their performances*

In XPRS, intraoperation parallelism is implemented in two ways: *page partitioning* and *range partitioning* . In page parititoning we partition relations across disk page boundaries and assign a subset of disk pages to each participating process to work on. In range partitioning we partition relations according the value of a certian attribute. Page partitioning is used for sequential scans and sorting. We can use the data distribution information in the system catalog to obtain a well-balanced range partition. For an index scan, range partitioning can also be facilitated by using the keys in the $B$-tree root node. A join operation can be parallelized by appropriately parallelizing its outer and inner paths. We can parallelize both the outer and inner paths for hashjoin and mergejoin, while we only parallelize the outer path for nestloop join. Also note that parallel hashjoin is the only operation with a critical section. It requires parallel access to a shared hash table because multiple processes may insert tuples into the same hash bucket simultaneously. To minimize the probability of conflict, the number of hash buckets should be large compared to the degree of parallelism.

DeWitt et al. have shown in [7] that intraoperation parallelism can achieve near-linear speedup in response time in a shared nothing environments. We briefly show the same near-linear speedup in XPRS in a shared memory environment. In addition, we show performance varying the number of disks and number

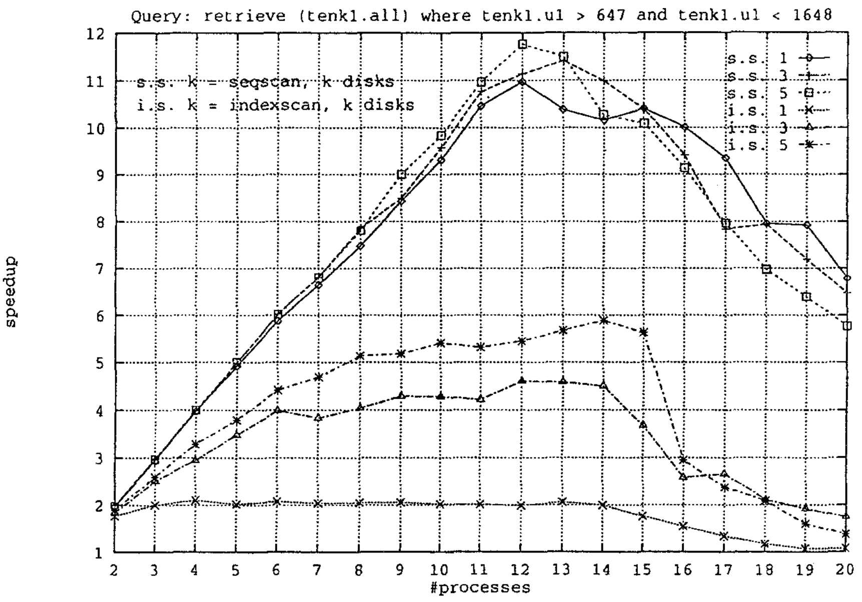


*Figure 5.* Speedup of parallel scan: small tuple.

of processes independently and also the degradation resulting from excessive parallelism.

Currently XPRS is operational on a Sequent Symmetry running the Dynix operating system with 12 CPU's connected via an 80MB/s bus and five disks controlled by three disk controllers.

In the experiments, we created the two 10,000-tuple relations from the Wisconsin benchmark, *tenk1* and *tenk2.* Because the tuple sizes in the Wisconsin benchmark relations are relatively small, we also created another 10,000-tuple relation, *ltenk1,* which has the same fields as tenk1 except that each tuple is filled to 1000 bytes by an extra string field. Appropriate indices were created according to the benchmark specification. All the relations are striped across a set of disks using a simple *mod* function, i.e., block $x$, is stored on disk ($x$ *mod #disks*). For example, if we only use two disks, then all the odd number blocks will be on one disk and all the even number blocks will be on the other. The relations are also partitioned among the parallel scan processes with a simple *mod* function, i.e., process $i$ scans block $x$ such that $x$ *mod #processes* $= i$. Before each execution, the file system cache is always cleared so that no blocks of the test relations are left in memory. All the processes are preforked so that process startup overhead is negligible.

We have measured the speedup of parallel scans and joins on the above relations varying the number of processes and number of disks. We present sample results

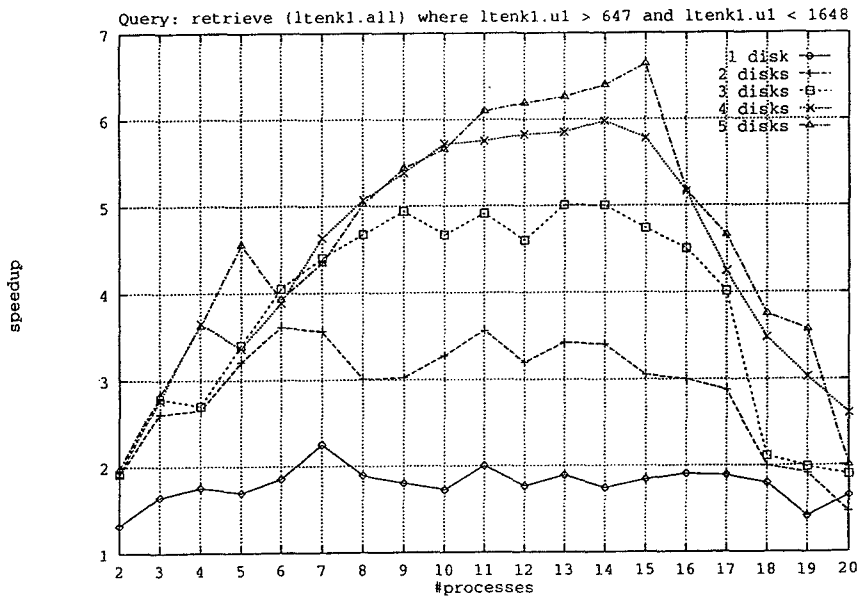


*Figure 6.* Speedup of parallel sequential scan: large tuple.

of the speedup measurements in Figures 5–7. Figure 5 and Figure 6 show speedup of scans, while Figure 7 shows speedup of joins. We have found that a sequential scan is CPU-bound when the tuples are small and becomes I/O-bound when the tuples get large. Moreover, index scans are I/O -bound because they do not need to examine every tuple in a page. Our experiment results show that parallel scans and joins can achieve close-to-linear speedup until they run out of processors if they are CPU-bound such as the sequential scan in Figure 5 and the join in Figure 7, or disk bandwidth if they are I/O-bound such as the index scan in Figure 5 and the sequential scan in Figure 6. Figure 7 also shows that the synchronization overhead caused by the shared hash table in hashjoins is negligible. In Figure 6, we see a drop in the speedup when the number of processes exceeds the number of disks. This is caused by the operating system read-ahead. When we have fewer processes than disks, the access pattern on each disk is sequential. Consequently normal file system read-ahead will prefetch the next disk block to be processed. When there are more processes than disks, the access pattern to each disk becomes random and the file system read-ahead is ineffective. Observe that there is always a performance drop when the degree of parallelism exceeds the number of processors. It results from the extra context switches and virtual memory overhead generated when the number of processes exceeds the number of processors. In addition, a process holding the shared buffer pool spin lock might be descheduled and the convoy problem [3] will occur.

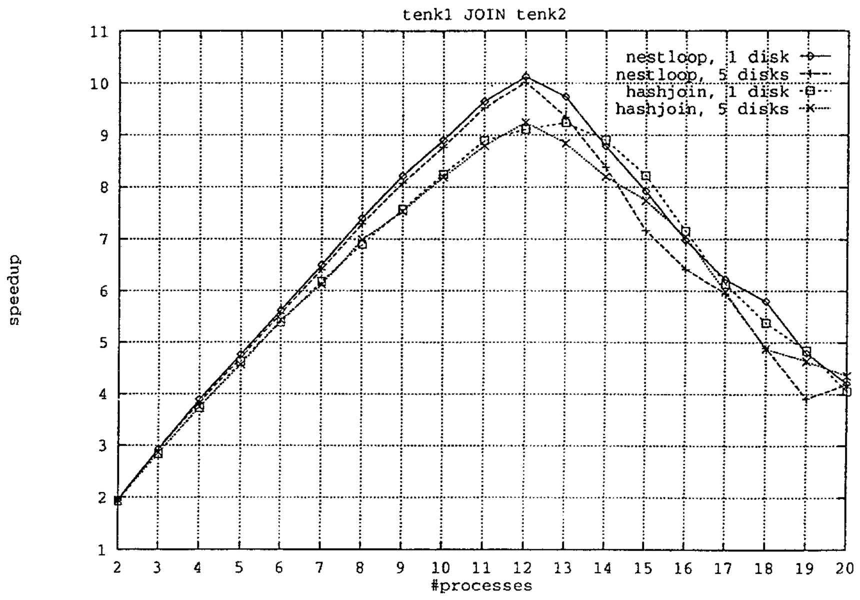


*Figure 7.* Speedup of parallel join: small tuples.

In a multiuser environment there will be multiple commands concurrently being processed, and the effective parallelism is the sum of the degree of parallelism of the individual command. It is important to ensure that this sum does not exceed the number of processors, and the XPRS algorithms are sketched in the next section.

## *4.2. Architecture of parallel query processing in XPRS*

Figure 8 gives an overall architectue of XPRS query processing, which consists of a master backend process and multiple-slave backend processes. The master backend is responsible for the optimization, scheduling and coordination, while the slave backends execute in parallel whatever plan fragments assigned to them by the master backends. The XPRS optimizer is a modified version of the POSTGRES optimizer that performs a [18] style exhaustive search to find the optimal sequential plan. We also have a postprocessor that uses cost functions similar to those in [19] and [11] to determine if two join or scan methods (specifically sequential scan versus unclustered index scan and nestloop with index versus hashjoin) may have "crossover" points in their cost curves against buffer size. If such points exist, the postprocessor will modify the optimal plan generated by the optimizer by adding appropriate *choose* nodes, which completes

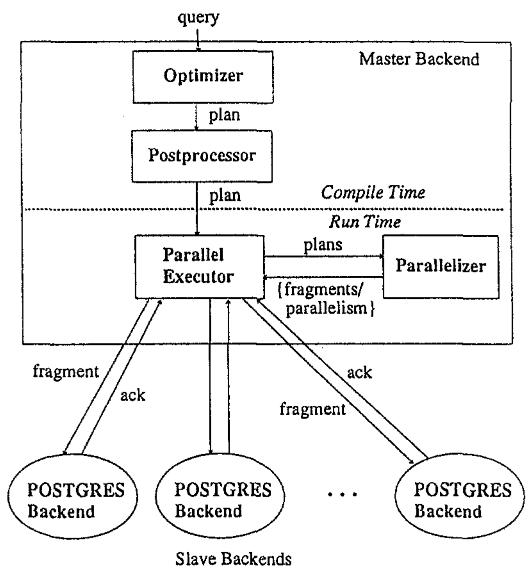


*Figure 8.* Architecture of XPRS query processing.

the compile-time optimization, i.e., the first phase of our two-phase optimization.

At run-time, the parallel executor passes a sequential plan to the parallelizer. The parallelizer performs the second phase of our two-phase optimization, i.e., to find the optimal parallelization for a selected sequential plan. It decomposes a sequential plan into a set of plan fragments, decides a processing schedule for all the plan fragments and assigns a degree of parallelism to each plan fragment that is to be executed, then passes a set of selected plan fragments along with their parallelism back to the parallel executor. The parallel executor then distributes the plan fragments to the slave backend processes according to the degree of parallelism of each fragments. The slave backends execute the plan fragments in parallel and send an acknowledgment back to the parallel executor after they finish executing. The parallel executor will then ask the parallelizer for more plan fragments to execute. The whole process repeats until all the plan fragments are finished.

The XPRS parallelizer is the key component for exploiting parallelism. It initially decomposes a sequential plan into plan fragments across blocking boundaries. Obviously larger plan fragments will cause less temporary relation overhead. However, the size of plan fragments is also constrained by the run-time available buffer space. For example, a plan fragment can contain as many as two hashjoins, i.e., one hash probe node followed by a hash build node. If the XPRS parallelizer cannot obtain the minimum required memory for both hashjoins, it will decompose the plan fragment into two smaller fragments between the hash probe node and the hash build node, save the intermediate results of the hash probe into a temporary relation and the following hash build node will read from the temporary relation after the previous hash probve is finished. Subsequent to the decomposition, we only need enough memory for one hashjoin at a time. Although the architecture in Figure 8 also allows interoperation parallelism, we will not discuss the details of the XPRS scheduling algorithm for interoperation parallelism in this paper since it has been addressed in [9]. For intraoperation parallelism, the parallelizer only needs to choose one plan fragment that is ready to execute and assign it the maximum degree of parallelism constrained by the disk bandwidth and number of free processors. The performance curves in Section 4.1 have shown the consequences of excessive parallalelism. Suppose that $N$ is the run-time number of available processors and $B$ is run-time available disk bandwidth (IOs/second). The parallelizer will estimate the I/O rate of the chosen fragment, for example, $C$ (IOs/second), and choose the degree of parallelism for the fragment as the following,

$$parallelism = min(N, B/C).$$

## 5. Verification of hypotheses

In this section, we verify the hypotheses that our two-phase optimization strategy is based on through XPRS experiments. In the experiments, we first run a wide variety of benchmark queries using all the possible execution plans under different buffer sizes and measure the real execution costs and then calculate the relative errors that result from our hypotheses. We will show that such errors are extremely small. We first present experiment results on the buffer-size-independent hypothesis, then present results on the two-phase hypothesis.

### *5.1. Choice of benchmarks*

We have chosen to use two benchmarks. The first benchmark is the Wisconsin benchmark [2], a standard benchmark for measuring query processing power. Because the Wisconsin benchmark has no more than three-way joins and has limited join selectivities, we also developed a random benchmark generator to generate addtional varieties of complex queries.

The relations in our random benchmark have the following POSTQUEL definition:

```
create  r  (ua1 = int4, a1 = int4, ua20 = int4,
            a20 = int4, ua50 = int4, a50 = int4,
            ua100 = int4, a100 = int4,
            filling = text)
```

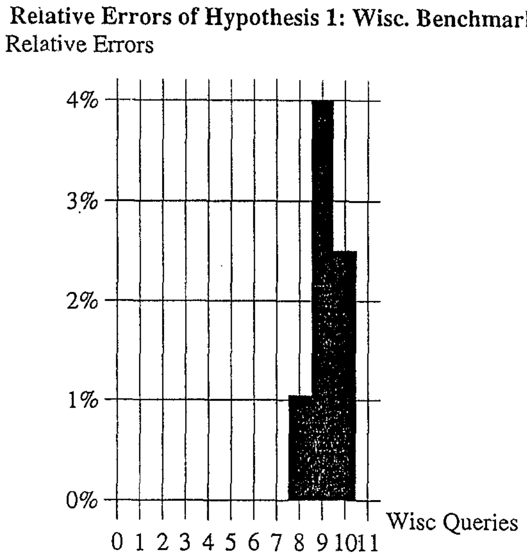


*Figure 9.* Relative errors of the buffer-size-independent hypothesis on the Wisconsin benchmark.

We generated 10 random relations with cardinalities ranging from 100 to 10,000. All the integer attribute values are randomly distributed between 0 and 9999. All the "ua" attributes are unclustered attributes and all the "a" attributes are clustered attributes. The number following "ua" or "a" indicates the number of times each value is repeated in the attribute. For example, each value in "ua20" is duplicated 20 times. We define indices on all the integer attributes so that the optimizer can always have the choice of generating an indexscan. The attribute "filling" is a variable length string, and is used to vary the tuple size of different relations. The generator can make the "filling" attribute 68 bytes or 968 bytes so that resulting tuple size is 100 bytes or 1000 bytes so that there can be a mixture of IO-bound and CPU-bound queries.

The queries in the random benchmark are generated in the following way. To generate a random join of $k$ relations, we first randomly choose $k$ relations. Then we start with the first relation in the chosen list and the rest in the unchosen list. We randomly pick a relation in the unchosen list, join it with a randomly picked relation in the chosen list on two randomly chosen attributes and move it from the unchosen list to the chosen list. We repeat this operation until the unchosen list becomes empty; and we have generated a random join on $k$ relations. Next we generate random selections on the relations. Each relation has a 50% probability of having a restriction of either an equality condition or inequality conditions with a lower bound and a upper bound. The target list is also randomly selected from all the attributes of all the relations with each attribute having 50% probability of being chosen.

## 5.2. *Experiments on the buffer-size-independent hypothesis*

In the experiments, we first have the XPRS query optimizer generate all possible sequential plans for each query. The XPRS optimizer is capable of generating all possible plans in the style of [18]. If a plan is known to be dominated by another plan, it is discarded. For example, since we know that hashjoin is the best join plan without the use indices (assuming that our buffer size is above the hashjoin threshold) [19], we can always remove nestloop and mergejoin plans which do not use indices from the test plans, which cuts down our experiment running time substantially. Another example is that mergejoin plans with inner relation and outer relation exchanged always have the same cost and therefore only one of them need to be executed. After selecting interesting execution plans, we run each plan with different buffer sizes varying from the minimum amount of buffer space to make all hashjoins possible to *MAXBUFS*, the buffer size that can hold all the relations in main memory. For each execution we measure the actual execution cost. To avoid the problem of inaccurate estimate of intermediate result sizes to hashjoin, we always execute the plans that do not contain hashjoins first and then set the sizes of intermediate results to the real sizes. The effect of inaccurate size estimates on our results is left as a future research topic.

From the statistics collected in the experiments, we know for each query, $Q$, the real execution cost of each query plan under different buffer sizes, i.e., given any plan $P$ and buffer size *NBUFS*, we know $Cost(P, NBUFS)$ and $BP(Q, NBUFS)$. Let $BCP(Q, NBUFS)$ be the plan obtained by adding appropriate choose nodes to plan $BP(Q, NBUFS)$. Since we know the cost of all the plans that are only different from $BP(Q, NBUFS)$ in some join or scan methods, we can also calculate the cost of $BCP(Q, NBUFS)$ under any buffer sizes. The relative error of the buffer size independent hypothesis is computed with the following formula:

$$\begin{aligned} FixBufCost &= Cost(BCP(Q, MAXBUFS), NBUFS), \\ MinCost &= Cost(BP(Q, NBUFS), NBUFS), \\ Error(Q, NBUFS) &= \frac{FixBufCost - MinCost}{MinCost} \end{aligned}$$

Figure 9 shows the graph of average relative errors of the Wisconsin benchmark queries. For each query $Q$, we first compute $Error(Q, NBUFS)$ for each buffer size, then we compute the average relative error over the buffer sizes. As we can see from the graph, the first few queries have 0 error. This is because those are the selection queries and two-way join queries for which the *choose* nodes can always choose the optimal plan. Errors occur in the three-way join queries,

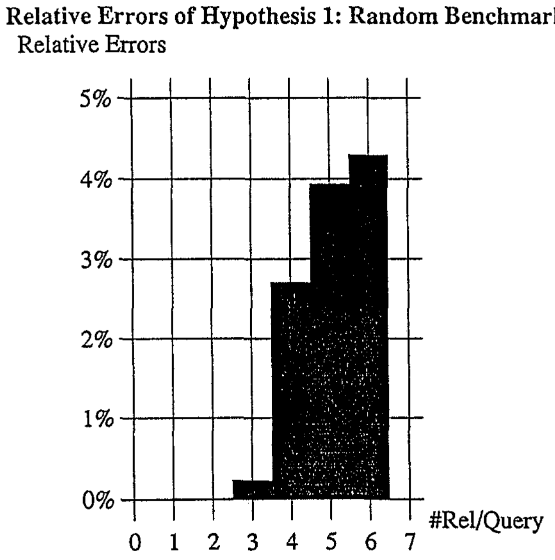


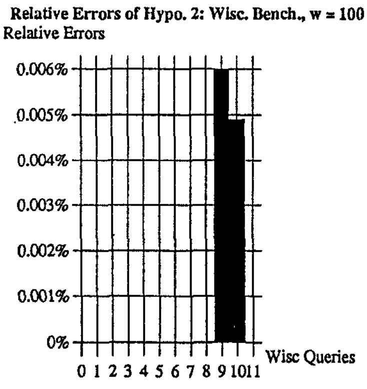
*Figure 10.* Relative errors of the buffer-size-independent hypothesis on the random benchmark.

but they are never above 4%.

Figure 10 shows the graph of average relative error on the random benchmark. In the experiment, 120 queries of up to six-way joins are executed. The relative error is averaged over queries that have the same number of relations (20 of them each). As we can see from the graph, the average relative errors remain below 5%.

Thus, two different benchmarks support the buffer size independent hypothesis. With choose nodes we have encapsulated enough of the plan switches required when buffer size changes so that average relative error is never above 5%.

We have also studied the detailed statistics from our experiments to find out the cause of these errors. We found that these errors are caused by plans with similar costs but different join orders and thus different I/O access patterns. For example, the plan $(A \bowtie B) \bowtie C$ may have a similar cost as the plan $(B \bowtie C) \bowtie A$ when the buffer size is fixed to *MAXBUFS*, but they have completely different I/O acess patterns. In other words, they have different working sets from the buffer manager's point of view. Therefore, their buffer hit rate changes differently when the buffer size varies. Hence, the plan that is optimal for buffer size *MAXBUFS* may become suboptimal for other buffer sizes because it may have higher buffer miss rate for certain buffer sizes. However, since this only happens between plans with similar costs to start with, the errors are very small, as shown by our experiment results.



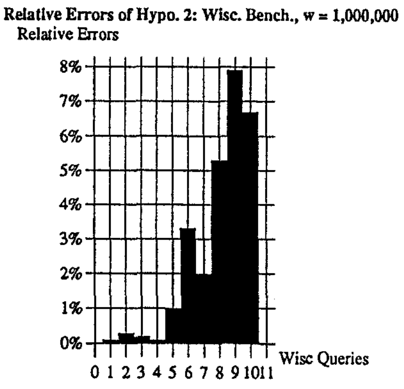


*Figure 11.* Relative errors of the two-phase hypothesis on the Wisconsin benchmark.

## 5.3. *Experiments on the two-phase hypothesis*

The experiments that we run to justify the two-phase hypothesis are similar to those described in the previous subsection. We have also used the queries from the Wisconsin benchmark and the random benchmark. For each test query, we first generate all the possible sequential plans as described in the previous subsection. We ran each plan with varying degrees of intra-operation parallelism and buffer sizes. The degree of parallelism is varied from 1 to 12 (the number of processors in our system) and the buffer size is varied in the same way as in the previous subsection. For each execution of a plan, *P*, with *NBUFS* buffers and *NPROCS* processes, we measure the resource consumption and response time, and compute the cost of the execution, *Cost*(*P*,*NBUFS*,*NPROCS*). Let *BP*(*Q*,*NBUFS*) be the best sequential plan with buffer size *NBUFS*. The relative error of the two-phase hypothesis is calculated with the following formula:

$$\begin{aligned} TwoPhaseCost &= Cost(BP(Q,NBUFS),NPROCS), \\ MinCost &= \underset{\{P\}}{Min}\, Cost(P,NBUFS),NPROCS), \\ Error(Q,NBUFS,NPROCS) &= \frac{TwoPhaseCost - MinCost}{MinCost} \end{aligned}$$

Because the cost of parallel plans depends on the system-specific weighting factor *w*, we need to calculate errors for different values *w*.

Figure 11 shows the average relative error of the Wisconsin benchmark queries. For each query, *Q*, the relative error, *Error*(*Q*,*NBUFS*,*NPROCS*), is averaged over all combinations of *NBUFS* and *NPROCS*. As we can see for small values of *w*, (which means that we mostly optimize resource consumption,) the relative errors are near 0. For large values of *w*, (which means that we mostly optimize response time,) the relative error never exceeds 8%.

Figure 12, shows the average relative error of the random benchmark. Due to the enormous computing resources demands of this experiment, we have only run queries of up to five-way joins from the random benchmark. As we can see, the average relative error never exceeds 6%.

By looking into the detailed statistics in our experiments, we have discovered that errors only occur when there are two plans that have very close sequential costs but are parallelized in different ways. For example, for a one-variable query with certain selectivities, the cost of the index scan plan can be approximately the same as the cost of the sequential scan plan (i.e., the sum of the number of index pages and the number of selected data pages is about the same as the total number of data pages). However, indexscans are parallelized through range partitioning while sequential scans are parallelized through page partitioning, which is easier to guarantee load balance. Therefore, even though the sequential execution of the sequential scan plan may be slightly slower than the sequential execution of the indexscan plan, the sequential scan plan may get better speedup when parallelized. Thus, the parallel version of the sequential scan plan may run faster than the parallel version of the indexscan plan because of their different ways of parallelization, which violates the two-phase hypothesis if we optimize response time. Our experiment results show that such errors are very small because each plan can achieve a near-linear speedup when parallelized through intraoperation parallelism and these plans have close sequential costs to start with. Although we have only validated this for queries with a small number of relations due to limited computational resources, we believe that this result is also true for queries with a large number of a relations.

## 6. Conclusion

In this paper, we have described our approach to the optimization of parallel query execution plans in a shared everything environment. We have demonstrated that we can achieve near-linear speedup with intraoperation parallelism in XPRS and there is severe performance penalty for excessive parallelism. We have presented experiment results that justify our two-phase optimization approach, which reduces the complexity of parallel query optimization without significantly compromising optimality of the resulting parallel plan. The first phase of our two-phase optimization is a conventional query optimization with a fixed buffer size that finds the optimal sequential plan augmented with appropriate *choose* nodes that are encapsulated in individual operations. The second phase finds the best parallelization of the sequential plan obtained from the first phase according to run-time environment. Our approach achieves run-time flexibility while still doing most of the optimization at compile time.

We are currently studying the trade-offs between intraoperation parallelism and interoperation parallelism more closely and looking into the scheduling and

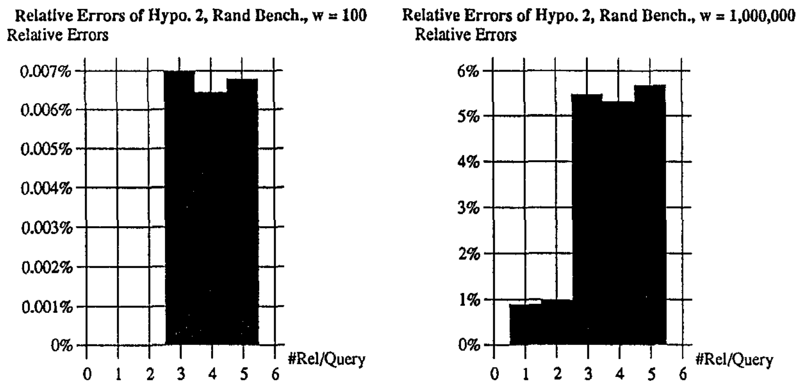


*Figure 12.* Relative errors of the two-phase hypothesis on the random benchmark.

memory allocation issues in processing multiple queries submitted by different users in parallel. In the experiments presented in this paper, we have assumed perfect estimates of intermediate result sizes and uniform data distribution. The effects of inaccurate size estimates and skewed data distributions also need to be studied more carefully.

## Acknowledgments

This research was sponsored by the National Science Foundation under contract MIP 8715235.

## References

1. A. Bhide and M. Stonebraker, "A performance comparision of two architectures for fast transaction processing," in *Proc. 1988 IEEE Data Engineering Conf.*
2. D. Bitton, et al., "Benchmarking database systems: A systematic approach," in *Proc. 1983 VLDB Conf.*
3. M. Blasgen, et al., "The convoy phenomenon," *Oper. Syst. Rev.* vol.13, no. 2, 1979.
4. G. Bultzingsloewen, "Optimizing SQL queries for parallel execution," *SIGMOD Rec.* vol.18, no. 4, 1989.
5. G. Copeland, et al., "Data placement in Bubba," in *Proc. 1988 ACM-SIGMOD conf.*
6. D. Cornell and P. Yu, "Integration of buffer manegement and query optimization in relation database environment," in *Proc. 1989 VLDB Conf.*
7. D. DeWitt, et al., "GAMMA: A high performance dataflow database machine," in *Proc. 1986 VLDB Conf.*
8. G. Graefe and K. Ward, "Dynamic query evaluation plans," in *Proc. 1989 ACM-SIGMOD Conf.*
9. W. Hong, "Exploiting inter-operation parellelism in XPRS," in *Proc. 1992 ACM-SIGMOD Conf.*

10. R. Katz, and W. Hong, "Performance implications of disk arrays to shared memory database machines," to appear in *Distributed Parallel Databases,* 1992.
11. L. Mackert, et al., "Index scans using a finite LRU buffer: A validated I/O model," *ACM-TODS,* Sept. 1989.
12. M. Murphy and D. Rotem, "Processor scheduling for multiprocessor joins," in *Proc. 1989 IEEE Data Engineering Conf.*
13. M. Murphy and M. Shan, "Execution plan balancing," in *Proc. 1991 IEEE Data Engineering Conf.*
14. D. Patterson, et al., "RAID: Reduntant arrays of inexpensive disks," in *Proc. 1988 ACM-SIGMOD Conf.*
15. J. Richardson, et al., "Design and evaluation of parallel pipelined join algorithms," in *Proc. 1987 ACM-SIGMOD Conf.*
16. D. Schneider and D. DeWitt, "A performance evaluation of four parellel join algorithms in a shared nothing environment," in *Proc. 1989 ACM-SIGMOD Conf.*
17. D. Schneider and D. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in *Proc. 1990 VLDB Conf.*
18. P. Selinger, et al., "Access path selection in a relational data base system," in *Proc. 1979 ACM-SIGMOD Conf.*
19. L. Shapiro, "Join processing in database systems with large main memories," *ACM-TODS,* Sept. 1986.
20. M. Stonebraker, "The case for shared nothing," in *Proc. 1986 IEEE Data Engineering Conf.*
21. M. Stonebraker, et al., "The design of XPRS," in *Proc. 1988 VLDB Conf.*
22. H. Zeller and J. Gray, "An adaptive hash join algorithm for multiuser environments," in *Proc. 1990 VLDB Conf.*