Copyright © 1990, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

PARALLEL QUERY PROCESSING IN XPRS

by

Wei Hong and Michael Stonebraker

Memorandum No. UCB/ERL M90/47

24 May 1990

PARALLEL QUERY PROCESSING IN XPRS

by

Wei Hong and Michael Stonebraker

Memorandum No. UCB/ERL M90/47

24 May 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

PARALLEL QUERY PROCESSING IN XPRS

by

Wei Hong and Michael Stonebraker

Memorandum No. UCB/ERL M90/47

24 May 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Parallel Query Processing in XPRS

Wei Hong and Michael Stonebraker

Computer Science Division, EECS Department

University of California at Berkeley

hong@postgres.berkeley.edu

mike@postgres.berkeley.edu

May 23, 1990

Abstract

In this paper, we present our design and some initial performance results of parallel query processing of XPRS[1], a parallel database machine based on shared-memory multiprocessors and disk arrays. In XPRS, we achieve load balancing by emphasizing on intra-operation parallelism and partitioning data equally among the processes. We solve the complex optimization problem of parallel query processing plans by dividing it into two steps. In the first step, we optimize sequential plans and in the second step we optimize parallelizations of sequential plans. Justifications to our approach are also provided in this paper.

1 Introduction

XPRS (eXtended Postgres on Raid and Sprite) is a parallel database machine based on shared-memory multiprocessors and disk arrays. The parallel environment of XPRS is shown in Figure 1. An outline of the initial overall design of XPRS can be found in [STON88] and the underlining reliable disk array system RAID is described in [PATT88]. This paper describes our approach to explore intra-query parallelism in XPRS.

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

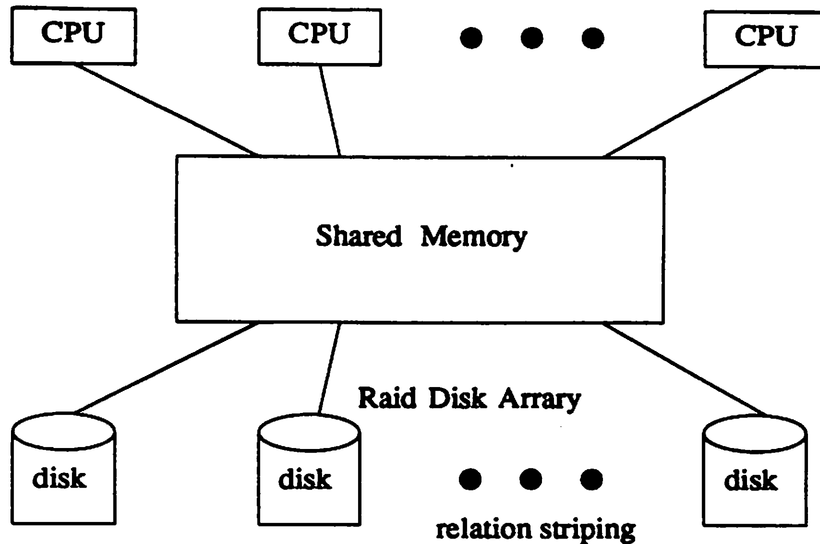


Figure 1: **The Parallel Environment of XPRS**

A shared-memory architecture has the following two major advantages over the *shared-nothing* architectures[STON86] adopted by other parallel database machines such as GAMMA [DEWI86] and BUBBA[COPE88]:

- **easy load balancing,** because a shared-memory system can automatically allocate the next ready process to the first available processor;
- **low message costs,** because message passing can be done through the shared memory.

In order to keep up with the I/O requests from the parallel CPU's, XPRS uses a disk array to eliminate the I/O bottleneck by parallelism among disks. In XPRS, all relations are striped across an array of disks and we have a two-dimensional file system[SELT89] to automatically balance the load among the disks. Therefore, to achieve load balance, all we need to do is to make sure that each process gets the same amount of tuples to process.

Our main goal is to achieve a close-to-linear speedup in query processing. With the advantages described above, we are confident that XPRS will outperform a shared-nothing system with an equivalent total number of processors, disks and amount of main memory

until XPRS hit the limit of the internal bus bandwidth. The simulation results in [BHID88] show that the potential win of XPRS over a shared-nothing system to be as much as a factor of two.

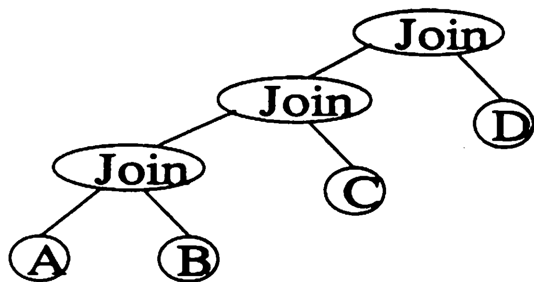
There are a few parallel query processing algorithms for shared-memory environments that have been published, e.g., [BITT83a, MURP89, RICH87, GRAE90]. However, they only deal with single operations, mainly, joins and sorts. They do not consider the amount of main memory buffer space available and other dynamic resource allocation issues in a multi-user system. In XPRS, we take all these issues into account, i.e., we will try to find optimal or near-optimal ways to parallelize a whole query processing plan in a dynamicly changing multi-user environment.

There are two major difficulties that we have to overcome: first, the enormous search space of parallel query processing plans, second, the the dynamic parameters, e.g., number of free processors, amount of available buffer space, etc. Our strategy is to divide query optimization into two steps. The first step deals with sequential query plans and static parameters and is done at compile time, just like conventional query optimization. The second step parallelizes the sequential plan obtained at the first step according to the run-time system environment. We will show that our strategy does not compromise the optimality of parallel query plans.

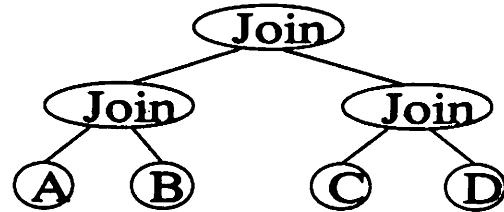
The rest of this paper is organized as the following. Section 2 explores the search space of parallel query plans and shows the complexity of our problem. Section 3 describes how intra-operation parallelism can be achieved in all the basic relational operators and shows some performance measurements of parallel scans. Section 4 formulates our optimization problem and justifies two key design hypotheses that lead to our two-step optimization strategy. Section 5 gives the overall query processing architecture of XPRS. Section 6 sketches future work and concludes the paper.

2 The Space of Parallel Plans

We call the query processing plans in conventional query processing which assumes a uniprocessor environment, *sequential plans*. A sequential plan is a tree consisting of basic relational



(a) a left-only-tree plan for a 4-way join



(b) a bushy-tree plan for a 4-way join

Figure 2: **Left-only-tree Plans v.s. Bushy-tree Plans**

operation nodes. In XPRS, the basic relational operators include **sequential scan, index scan, nestloop join, mergesort join** and **hashjoin.** Most optimizers, such as the System R optimizer[SELI79] and the original Postgres optimizer[FONG86] only consider *leftonly-tree*(see Figure 2a) plans to reduce the search space. However, in a parallel environment, *bushy-tree*(see Figure 2b) plans should be considered because they are better tailored towards parallelism. For example as in Figure 2, leftonly-tree plans will forbid the possibility of executing $(A\ JOIN\ B)$ and $(C\ JOIN\ D)$ in parallel.

We call the query processing plans in a multiprocessor environment, *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.

- **Form of Parallelism**

  There are two generic ways to achieve parallelism: *data partitioning* and *function partitioning.* Data-partitioning achieves *intra-operation* parallelism and function-partitioning achieves *inter-operation* parallelism. In a database system, we can have multiple processes working on the same relational operation but each with a different set of tuples. We call this *partitioned operations.* We can also have one set of processes working on one operation and one set of processes working on another operation. They can be

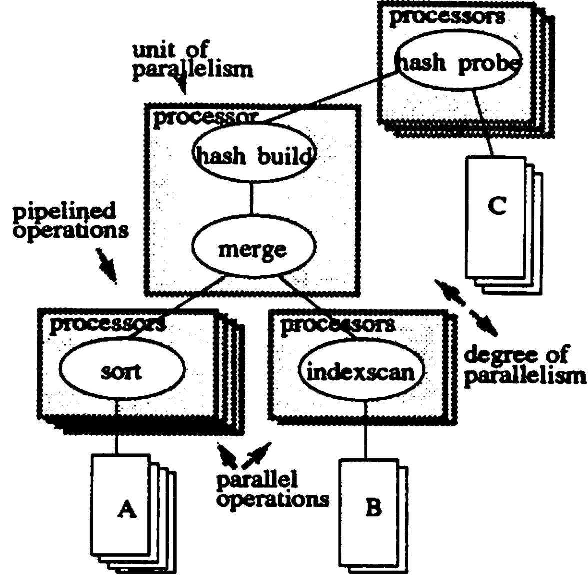


Figure 3: **An Example of Parallelizations**

fully in parallel or in a pipeline. We call these two forms of inter-operation parallelism *parallel operations* and *pipelined operations.*

Figure 3 shows an example of a parallelization of a three-way join, $(A\ JOIN\ B)\ JOIN\ C$, in which the first join is performed by mergesort and the second by hashjoin. For convenience, we have separated the two phases of a hashjoin into two nodes: *hash build* and *hash probe.* The sort, indexscan and hashjoin probe can be executed in parallel by multiple processes, as examples of *partitioned operations.* The sort and indexscan can be performed as *parallel operations*, and the sort, indexscan and merge can be performed as *pipelined operations.*

- **Unit of Parallelism**

Unit of parallelism refers to the group of operations that is assigned to the same process. In general it can be any connected subgraphs of a plan tree and can range from a single operation to multiple operations. We also call a unit of parallelism a *plan fragment.* For example in Figure 3, the hash build and merge is a plan fragment.

- **Degree of Parallelism**

  Degree of parallelism is the number of processes we use to execute a plan fragment. For example in Figure 3, we assign two processes to work on the indexscan in parallel, thus the degree of parallelism is two.

Figure 3 gives a global picture of parallelization of sequential plans. The next section will zoom inside the boxes and shows how each relational operation can be parallelized.

3 Parallelization of Relational Operations

All query processing consists of execution of the basic relational operations as mentioned above. Fortunately, all these relational operations are suitable for intra-operation parallelism. In XPRS, we emphasize on exploring intra-operation parallelism because if we partition the input relations into equal-size chunks, intra-operation parallelism guarantees load balance. Load balancing becomes much harder with inter-operation parallelism. Parallelizations of individual relational operations are described belowed.

Sequential Scan is trivial to parallelize. We partition the disk blocks and have each process work on tuples from a different set of disk blocks.

Index Scan is more problematic. The basic approach to parallelize index scan is to partition the scan range into subranges and have each process scan for tuples within a subrange. For load balance, we have to make sure that each of the subranges contains approximately the same number of tuples. In Postgres, there is a vacuum daemon process that wakes up every once and a while and scans through the entire database to dump out-of-date tuples into archives and keep statistics about data distributions in the current database. Range-partitioning for index scan has to rely on these statistics. It is also possible to get the information we need for a balanced range-partition in the B-tree index itself.

Nestloop Join is another easy operation to parallelize. We partition the outer relation and have each process join a portion of the outer relation with the inner relation.

Hashjoin parallelization requires a large chunk of shared memory because we have to put the hash table in the shared memory so that in the first phase we can have multiple processes insert into the hash table and in the second phase have multiple processes probe into the hash table.

Sort parallelization is similar to index scan. First, we partition the entire range of the sort key into subranges such that each subrange contains approximately the same number of tuples according to the current database statistics. Then, we do a parallel scan to redistribute tuples according to the subranges and have each process sort tuples within a subrange independently. Last, we attach these sorted segments together and form an entire sorted relation.

Mergesort Join can be parallelized in almost the same way as sort.

As we notice from above, in parallelization of all the relational operations except hashjoin, the multiple processes all work independently of each other. In hashjoin parallelization, we have a shared hash table that introduces critical sections. However, possible conflicts only happen when two processes try to insert tuples into the same hash bucket at the same time. If we always make the number of hash buckets large enough, we can keep the possibility of conflict negligible. Therefore, if we partition the operand relations properly, we can always achieve load balance in the parallelizations of the basic relational operations and see a near-linear speedup. We have implemented parallel scans for XPRS and the following are some performance measurements.

Right now XPRS is being implemented on a Sequent Symmetry with 12 CPU's connected via an 80MB/s bus and 5 disks controlled by 2 dual-channel disk controllers. The operating system that we are using at the moment is Dynix, a BSD-based multiprocessor UNIX. XPRS will eventually move onto the Sprite operating system.

In the experiments, we created two sample relations with 10,000 tuples each. One relation has 100 bytes/tuple with the same fields as the Wisconsin-Benchmark relations [BITT83b]. The other relation has 1K bytes/tuple. The relations are striped across disks using a simple *mod* function, i.e., block x, is stored on disk (x *mod* number of disks). The relations are also partitioned among the parallel scan processes with a simple *mod* function, i.e., process i scans

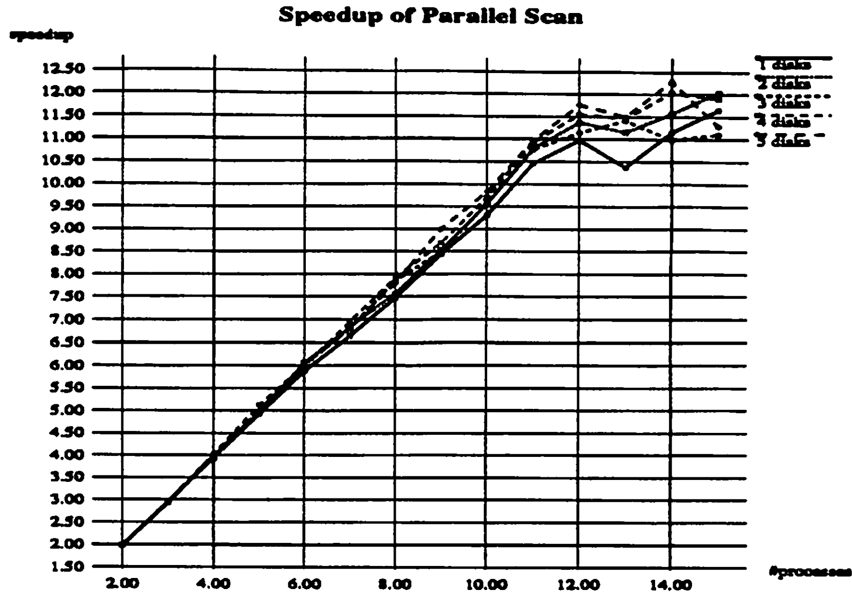


Figure 4: **Speedup of Parallel Scan: small tuples, forward scan**

block x such that x *mod* number of processes = i. We adopt this partition scheme instead of a range-partition scheme because this scheme may get the benefit of file system readahead in UNIX when the number of processes is smaller than the number of disks. The striped relations are stored in separate files, one on each disk. In our partition scheme, if we have less processes than the number of disks, then the I/O requests on each file will be sequential, and therefore the file system readahead will be turned on. Before each parallel scan, the file system cache is always cleared so that no blocks of the test relations are left in memory. All the processes are pre-forked so that process startup overhead is negligible. In order to isolate the effect of file system readahead, we scan the relations both forward and backward. When the relations are scanned backwards, no readahead is possible.

We have measured the speedup of parallel scans on the two relations varying the number of processes and number of disks. The results are presented in Figure 4-6. In Figure 4, because the tuples are small, the scans are completely CPU-bound and disk striping does not make much difference in performance. The speedup is only limited by the number of processors available. However, in Figure 5 and 6, because the tuples are large, the scans are I/O bound

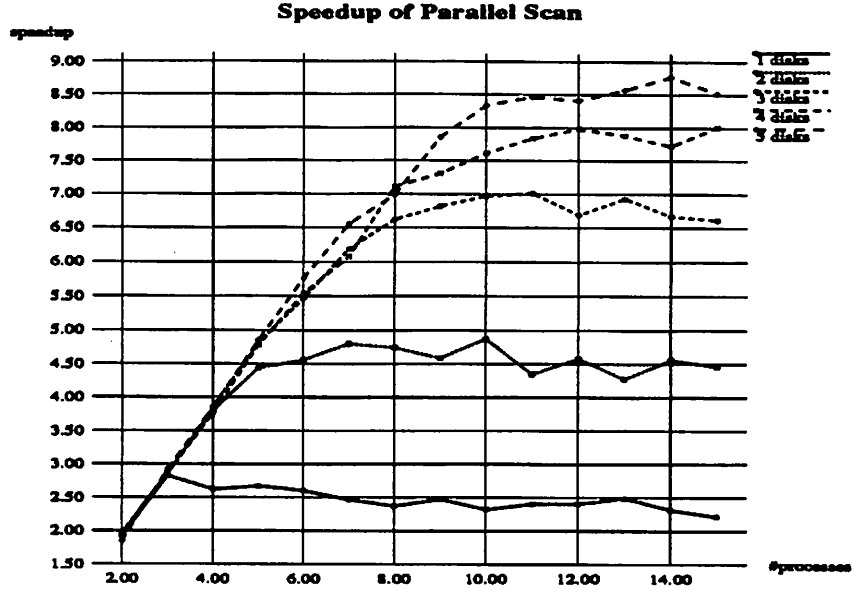


Figure 5: **Speedup of Parallel Scan: large tuples, backward scan**

and disk striping does show a big win. In Fiugre 5, we can see a near-linear speedup as we increase the number of processes until it saturates the disk bandwidth. However, in Figure 6, we see a drop in the speedup when the number of processes exceeds the number of disks. This is because after that point the access pattern to each file becomes random and the file system readahead is turned off.

4 The Optimization Problem and Two Key Hypotheses

We have understood the possible parallelizations of sequential plans and parallelizations of each individual relational operations. Now the question is how to optimize them.

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

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

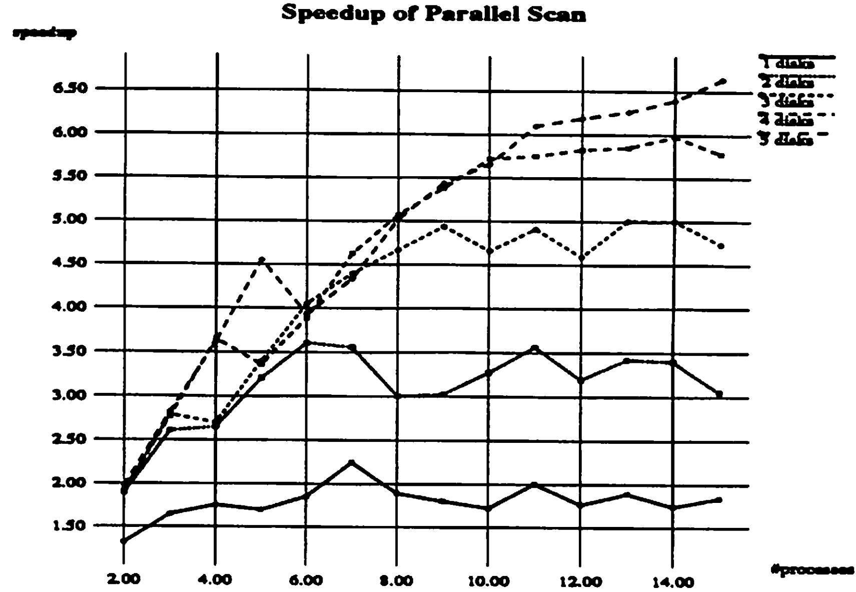


Figure 6: **Speedup of Parallel Scan: large tuples, forward scan**

$PPLAN(Q)$ is the space to explore for intra-query parallelism.

The optimization problem is to minimize query processing cost among all the possible parallel plans. In XPRS, we measure plan cost by response time. Obviously, plan costs depend on two parameters: amount of available buffer space and number of free processors. 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 processors. Our optimization problem is to find,

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

There are two major difficulties in this problem. First, the search space $PPLAN(Q)$ is orders of magnitude larger than $SPLAN(Q)$, therefore the conventional wisdom for query optimization to do exhaustive searches simply does not work. Second, the dynamic parameters, $NBUFS$ and $NPROCS$ are unknown until query execution time, therefore compile-time optimization will have to deal with these unknown parameters. Our objectives are to reduce the complexity in plan searching and to do as much as possible at compile time. Fortunately, we have justified the following two design hypotheses for XPRS which greatly simplify our

optimization problem.

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

1. **The Buffer-Size-Independent Hypothesis**

   The choice of the best sequential plan does not depend on the amount of buffer space available, i.e., $Cost(BP(Q, NBUFS), NBUFS) \approx Cost(BP(Q, NBUFS'), NBUFS)$, where $NBUFS \neq NBUFS'$

2. **The 2-Step Hypothesis**

   The best parallel plan is a parallelization of the best sequential plan, i.e., $BPP(Q, NBUFS, NPROCS) \in PARALLEL(BP(Q, NBUFS))$.

Our justifications are based on the following assumptions.

- *There is no overlap between CPU and I/O.*

  This is an assumption most optimizers make in their cost models, simply because with overlapping, response time becomes too hard to estimate. We have adopted this assumption in the cost model of our XPRS optimizer. Conventional query optimization minimizes a linear combination of I/O cost and CPU cost. Under this assumption, it also minimizes response time.

- *Buffer size is always above the hashjoin threshold, approximately the square root of the size of the smaller relation[DEWI84, SHAP86], and large enough hold all the selected data pages in a non-clustered index scan.*

  XPRS is designed to have a large amount of main memory buffer. For example, joining two 10 gigabytes relations with a disk block size of 4K requires 6.4 megabytes of buffer space for a hybrid hashjoin. XPRS is expected to routinely have much more buffer space than this amount. It has been proved in [DEWI84, SHAP86] that hashjoin is

always the best join plan for unindexed relations as long as the buffer size is above the the threshold. Our buffer-size-independent hypothesis is based on this result.

We justify the buffer-size-independent hypothesis by our experiment results on the Postgres optimizer. The Postgres optimizer is a standard conventional query optimizer, i.e., it uses a generic exhaustive search algorithm on all the possible bushy-tree plans and a set of standard of cost functions as in [SHAP86]. Given a query and a buffer size, the Postgres optimizer finds the optimal sequential plan.

In the experiment, 10 random relations are generated in a Postgres database. The number of attributes in the relations is randomly distributed between 5 and 20. 75% of the attributes are 4-byte integers (25% other data types). 50% of the integer attributes are indexed. No actual tuples are generated, but appropriate statistics are fabricated and loaded into the system catalogs to make the join and qualification clause selectivities more realistic. The number of tuples in a relation is randomly chosen among 10, 1,000, 10,000, 100,000, 1,000,000 and 10,000,000. 30% of the attributes are assumed to have unique values and therefore the number of distinct values in these columns is the same as the number of tuples in the relation. The number of distinct values in the other 70% columns is randomly distributed between 1 and the number of tuples in the relation. 100 random queries on the above 10 relations are generated for the experiment. The number of relations in a query is randomly distributed between 1 and 6. Each relation in a query is equijoined to another randomly chosen relation in the query on two randomly chosen integer attributes. 50% of the relations in a query also have one qualification on some random attribute. The length of the target list and the attributes in the target list of the queries are also randomly chosen. The following is an example of the random queries.

retrieve (r3.a17,r4.a6,r3.a8)

where r0.a8=1 and r6.a9=1

and r3.a9=r6.a1 and r6.a0=r4.a4 and r4.a6=r0.a10

Each of the 100 random queries is optimized by the Postgres optimizer assuming buffer size ranging from 16 to 16384 pages, i.e., 128KB to 128MB (Postgres has 8K pages). For each

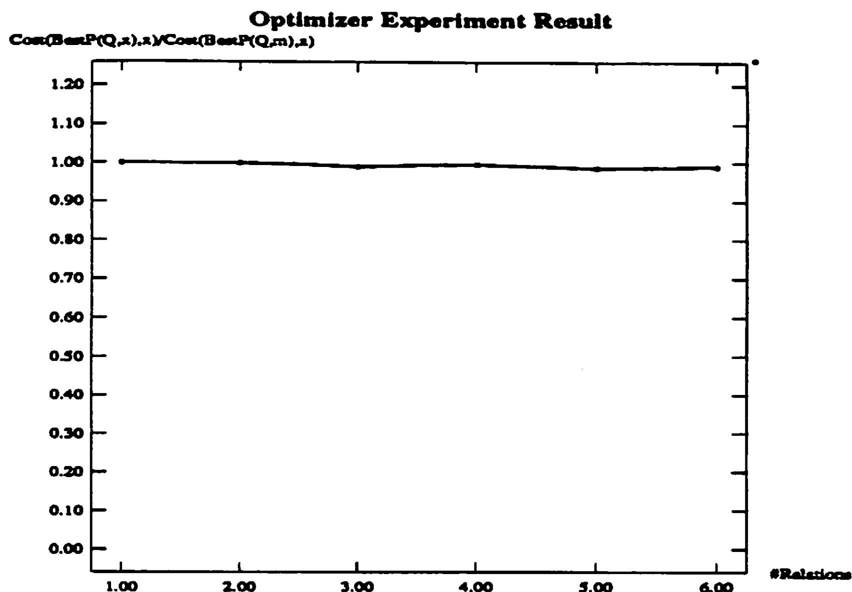


Figure 7: **Justification to Buffer-Size-Independent Hypothesis**

query, Q, we first find the best plan assuming the entire buffer pool is available. Suppose this plan is $BP(Q, ALLBUFS)$. Then for each buffer size, $NBUFS$, we compare the cost of $BP(Q, ALLBUFS)$ and $BP(Q, NBUFS)$. We found that,

$$Cost(BP(Q, ALLBUFS), NBUFS)/Cost(BP(Q, NBUFS), NBUFS) \approx 1,$$

for all $NBUFS$ above the hashjoin threshold as is shown in Figure 7.

Our justification to the 2-step hypothesis is based on the results from the previous section that intra-operation parallelism can achieve near-linear speedup. The basic reasoning is the following:

1. **the optimal sequential plan is the fastest sequential plan assuming no CPU-I/O overlap;**
2. **by exploiting intra-operation parallelism, given the same number of processors, all the sequential plans get an equal speedup;**
3. **therefore, the parallelization of optimal sequential plan remains the fastest.**

5 XPRS Query Processing

Based on the above two hypotheses, we can overcome the enormous complexity of parallel plan optimization by the following 2-step algorithm.

- **Step 1.** Find the optimal sequential plan given a fixed amount of buffer space, i.e., find $BP(Q, ALLBUFS)$ where $ALLBUFS$ is the size of the whole buffer pool.
- **Step 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 the number of free processors.

Because we have a fixed buffer size $ALLBUFS$ in the Step 1, Step 1 can be carried out at compile time. It is the same as a conventional query optimization. Step 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 Step 1.

Figure 8 gives an overall architecture of XPRS query processing. The XPRS optimizer is the same as any conventional optimizer. We will simply use the Postgres optimizer. The parallelizer takes a sequential plan and decomposes it into a set of plan fragments, decides the degrees of parallelism for each fragment and passes the fragments to the parallel executor. The executor sends the plan fragments to a number of Postgres backend processes running on separate processors according to the degree of parallelism determined by the parallelizer. The multiple Postgres backend processes execute the plan fragments in parallel. After they finish executing their plan fragments, they send acknowledgements back to the parallel executor and the parallel executor will mark the corresponding fragments as completed and send the modified plan tree back to the parallelizer. Then the whole parallelization process repeats.

The XPRS parallelizer is the key part for exploiting parallelism. In fact, it does not have to examine all possible parallelizations of a sequential plan. The following are the heuristics that it uses in the different aspects of parallelization.

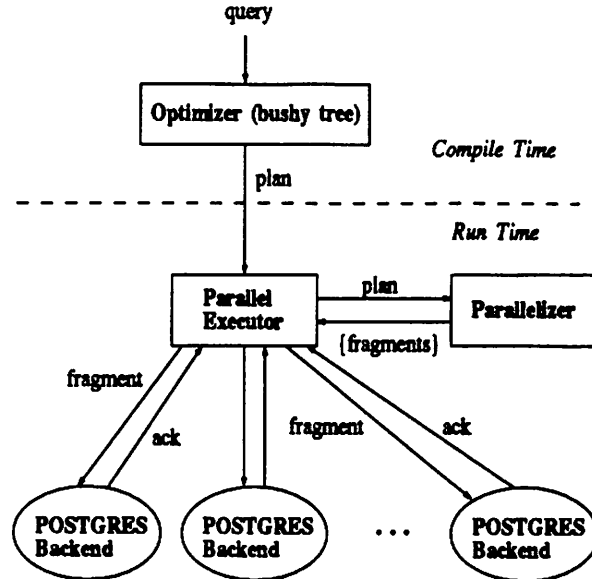


Figure 8: **The Architecture of XPRS Query Processing**

- **Form of Parallelism**

  The parallelizer tries to exploit intra-operation parallelism as much as possible for the sake of load balance. It only considers executing two independent plan fragments in parallel when there are some free processors left by previous fragments. No pipelined operations are considered to avoid process waiting.

- **Unit of Parallelism**

  Obviously larger plan fragments will cause less temporary relation overhead. However the size of a plan fragment is limited by blocking edges in the plan tree. We draw a *blocking* edge between two operations if one operation has to wait for the other operation to finish producing all the tuples before it can start. All edges coming out of a sort node or a hash node are blocking edges. Obviously, we do not want to include any blocking edges in a plan fragment. Initially, the parallelizer will decompose a sequential plan tree across blocking edges and get a unique set of plan fragments which are the largest possible plan fragments. See Figure 9 for an example.

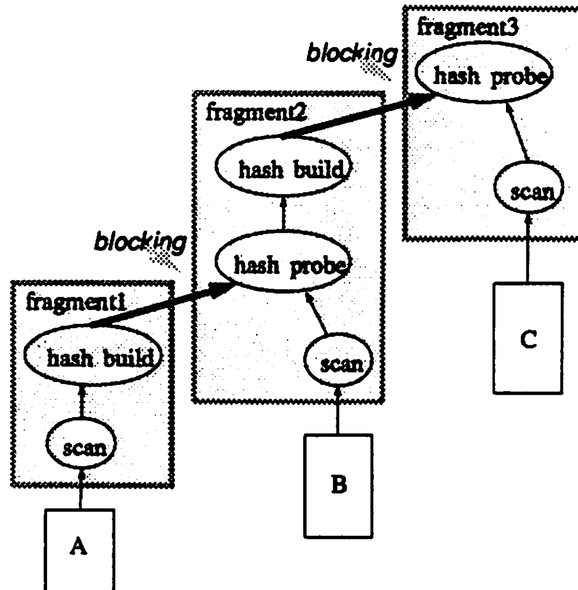


Figure 9: **An Example of Initial Plan Fragments**

The size of plan fragments are also constrained by the runtime available buffer size. For example, we can not take two hashjoins in a plan fragment unless there is enough buffer space to hold both hash tables. In case of insufficient buffer space, the parallelizer will have to decompose the plan fragments into even smaller fragments.

- **Degree of Parallelism**

  The degree of parallelism is ultimately limited by the disk bandwidth and number of free processors. The parallelizer will make sure not to create too many processes to saturate the disk bandwidth. And to be considerate of other users on the system, the parallelizer will adjust the degree of parallelism according to the current system load average. Some detailed formulas are given in [STON89].

6 Conclusion and Remaining Work

We have sketched the overall design of XPRS query processing and shown some initial performance numbers of parallel scans. In summary, our approach is unique in two aspects.

First, we emphasize on intra-operation parallelism and guarantee load balance by partitioning data equally among the processors. Second, we use a 2-step optimization algorithm to overcome the enormous complexity in searching for an optimal parallel plan without compromising optimality. So far, the implementation for parallel scans have been completed and the implementation for parallel joins is moving along. The parallelizer has also been partially implemented. However, there is still a lot of work remaining to be done:

1. Complete Implementation and Performance Evaluation

   First, we need to complete the rest of the implementation and test our hypotheses and measure speedups in a fully functional system. Our justification to the buffer-size-independent hypothesis is based on the experiment on the Postgres optimizer. After we complete the implementations, especially on hashjoins, we will test this hypothesis in terms of real execution costs instead of estimated optimizer costs. If it turns out to be not valid, we will need to have a more elaborate optimizer at the first step. A simple extension to our current approach is to try to identify the intervals of buffer sizes within which the choice of optimal plan does not change and find the optimal plans within these intervals. It is easy to achieve this for single operations. By intersecting intervals of all the operations in a query plan, we can get the intervals of the whole plan. And we can also calculate the plan cost in each of the intervals. By comparing the plan costs in corresponding intervals, we can find the optimal plan for each interval. As long as there is only a small number of intervals, this extension will work perfectly fine. It will generate the same number of plan trees as conventional query optimization and for each plan tree the extra cost is a little interval intersecting. In our current design, we postulate that parallelization overhead is small. This also needs to be confirmed by a working system.

2. New Hashjoin Algorithm

   Because hashjoin is the best join algorithms for unindexed relations given enough buffer space, we expect to see a lot of hashjoin nodes in the query plans. However, traditional

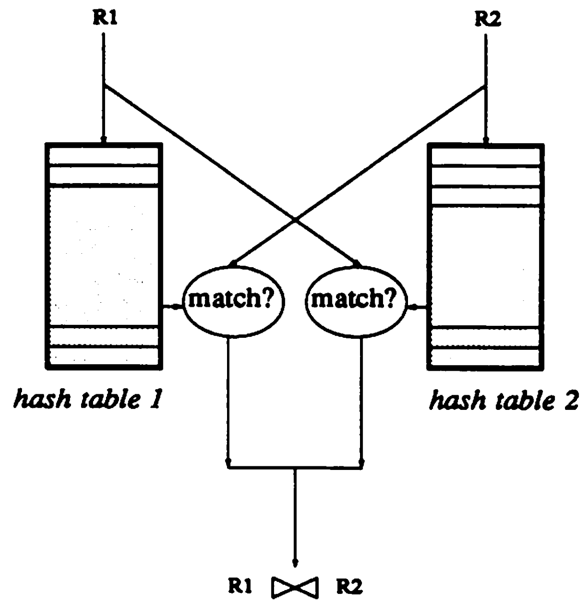


Figure 10: **Non-blocking Hashjoin**

hashjoin algorithm introduces a lot of blockings into the plan tree because it consists of two phases: the *build* phase and the *probe* phase, and the probe phase has to block until the build phase finishes. Alternatively, we have a non-blocking hashjoin algorithm as illustrated in Figure 10. The idea is use two hash tables, one for each relation. Every time we get a tuple from one relation, we look into the hash table of the other relation. If we find a match, we output a result. In any case, we always insert the tuple into the hash table of its own relation. Obviously, there is no blocking in this hashjoin algorithm. We can detect matches as soon as they show up along with building the hash tables. The advantage of non-blocking hashjoin is that by reducing the number of blocking edges in a plan tree, we can increase the possible sizes of plan fragments. The disadvantage of non-blocking hashjoin is that it requires more than twice as much memory for two hash tables instead of one. We plan to implement non-blocking hashjoin in XPRS and study the performance implications.

3. Abstract Data Types

So far, we have only dealt with traditional relational operations. Because Postgres is an extendible database, we also need to deal with various kinds user-defined operations on abstract data types. For example, we may need to provide an interface to let the user provide special functions for data partitioning and gathering, or special heuristics for parallelization.

4. Multiple Query Optimization

   In the design described above, we have only considered optimization of a single query. The problem of how to allocate the shared memory and processors to a set of queries submitted by different users at the same time has not been addressed. We plan to study this problem after we get more experience with single query parallel execution.

References

[BHID88] Bhide, A. and Stonebraker, M., "A Performance Comparison of Two Architectures for Fast Transaction Processing, " Proc. 1988 IEEE Data Engineering Conference, Los Angeles, CA, Feb. 1988.

[BITT83a] Bitton, D., et. al., "Parallel Algorithms for the Execution of Relational Database Operations," ACM-TODS, Sept. 1983.

[BITT83b] Bitton, D., et. al., "Benchmarking Database Systems: A Systematic Approach," Proc. VLDB, 1983.

[COPE88] Copeland, G., et. al., "Data Placement in Bubba," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, IL, June 1988.

[DEWI84] Dewitt, D., et. al., "Implementation Techniques for Main Memory Data Base Systems," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, MA, June 1984.

[DEWI86] Dewitt, D., et. al., "GAMMA: A High Performance Dataflow Database Machine," Proc. 1986 VLDB Conference, Kyoto, Japan, Sept. 1986.

[FONG86] Fong, Z., "The Design and Implementation of the POSTGRES Query Optimizer," M.S. Report, Univ. of California, Berkeley, CA, Aug. 1986.

[GRAE90] Graefe, G., "Encapsulation of Parallelism in the Volcano Query Processing System," Proc. 1990 ACM-SIGMOD.

[MURP89] Murphy, M. and Rotem, D., "Processor Scheduling for Multiprocessor Joins," Proc. 1989 IEEE Data Engineering Conference, Los Angeles, CA, Feb. 1989.

[PATT88] Patterson, D., et. al., "RAID: Redundant Arrays of Inexpensive Disks," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, IL, June 1988.

[RICH87] Richardson, J., et. al., "Design and Evaluation of Parallel Pipelined Join Algorithms," Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, CA, May 1987.

[SELI79] Selinger, P., et. al., "Access Path Selection in a Relational Data Base System," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, MA, June 1979.

[SELT89] Seltzer, M., "Analysis of Extent Allocation Policies in File Systems," M.S. Report, Univ. of California, Berkeley, CA, 1989.

[SHAP86] Shapiro, L., "Join Processing in Database Systems with Large Main Memories," ACM-TODS, Sept. 1986.

[STON86] Stonebraker, M., "The Case for Shared Nothing," Proc. 1986 IEEE Database Engineering, March 1986.

[STON88] Stonebraker, M., et. al., "The Design of XPRS," Proc. 1988 VLDB Conference, Los Angeles, CA, Sept. 1988.

[STON89] Stonebraker, M., et. al., "Parallelism in XPRS," Electronics Research Laboratory, University of California, Berkeley, CA, Report M89/16, February 1989.