PARALLELISM IN XPRS

Michael Stonebraker, Paul Aoki, and Margo Seltzer
EECS Dept.
University of California, Berkeley

Abstract

This paper discusses the approach being taken by XPRS, a software data base machine being built at Berkeley, toward intra-query parallelism. There are two components to our design. The first is a unique two-dimensional file system called FTD which automatically balances arm activity among multiple disks. The second component is the novel way that we construct parallel plans by making use of information about available main memory and the load average per processor. Both components are described in this paper.

1. INTRODUCTION

This paper explores parallelism in the software data base machine XPRS. An initial paper presenting the outline of the system appeared in [STON88] and one on the reliable disk system in [PATT88]. This paper explores the file system that we are constructing and the novel way we will obtain intra-query parallelism in the resulting system.

Recent systems that exploited intra-query parallelism include Gamma [DEWI86, DEWI88], Bubba [COPE88], Non-stop SQL [GRAY87, BORR88] and the Teradata 1012 [TERA85]. Earlier approaches to parallelism included those of Distributed INGRES [STON83], and SDD-1 [ROTH80]. All of the above systems can be classified as **shared-nothing** systems [STON86] and have the common characteristic that intra-query parallelism is **structural** in nature. For example, suppose the following distribution criteria is utilized to partition the EMP relation among 5 sites:

salary < 1000	to site 1
1000 <= salary < 2000	to site 2
2000 <= salary < 3000	to site 3
3000 <= salary < 4000	to site 4
salary >= 4000	to site 5

In this case, a query to find the names of employees who earn $2500 will be processed only on site 3. Hence, the data distribution determines the processing site (or sites). In this environment there are sure to be two serious problems:

1) load balance
2) message costs

Unless salaries of employees are uniformly distributed and queries on salary ranges are likewise uniform, the 5 sites in the system will not have an equal amount of work to do. In this case, the entire system will bottleneck on the overloaded processor and system response time will be determined by the speed of this

This research was sponsored by the National Science Foundation under Grant MIP 8715235 and by a grant from IBM.

machine.

This load balance issue has prompted Gamma to favor distributing tuples based on a hash function to achieve uniformity [DEWI86]. However, a hashing approach will guarantee that all 5 processors will execute every query. For single record queries like finding the name of the $2500 employee, this will result in 5 times as much work being done as necessary. In a transaction processing environment, such a performance penalty would surely be a show stopper.

The second problem in shared nothing architectures is message costs. When a query is executed in parallel on multiple systems, there are inevitable messages that must be sent between systems to synchronize multiple computations. Likewise, data tuples must sometimes be sent to other sites, again generating message traffic. This has prompted Bubba to advocate spreading data over less than all sites to lower the amount of intra-query parallelism that can be used so that the number of messages declines [COPE88].

Since XPRS is a shared memory system, neither of these disadvantages need be experienced. A shared memory system will automatically allocate the next computation to the first available processor, and automatic CPU load balancing is obtained. Moreover, we will indicate in Section 2 the design of a file system, FTD (Files -- Two Dimensions) which will automatically balance the load on multiple disks in an XPRS system. Consequently, XPRS can use **dynamic** intra-query parallelism, i.e. the number of parallel plans into which a query is broken can be varied over wide ranges without compromising CPU or disk load balance and without incurring messages between separate computer systems. The reasons to dynamically vary the number of plans concern the load average on the system as well as the amount of main memory buffer space available. In Sections 3 we discuss our approach to parallel query plans in XPRS and explain this unique dynamic use of parallelism.

Lastly, in a shared memory environment, tuples and messages can be passed through shared memory and the serious overhead of messages for data transmission is thereby reduced. Consequently, we are confident that XPRS will outperform a shared nothing system with an equivalent total number of CPUs, disks and megabytes of main memory until XPRS "hits the wall". This will occur when the internal system bus of a shared memory system bottlenecks and no more processors and disks can be added.

Others have proposed query processing algorithms for a shared memory environment, e.g. [BITT83, MURP89, RICH87]. Unlike other work, our algorithms are unique because they:

1) deal explicited with the amount of main memory available
2) deal with complete plans not just single join operations
3) deal with dynamic allocation of resources in a multi-user system

2. THE DESIGN OF FTD

2.1. RAID -- The Underlying Disk System

XPRS is assumed to run on a collection of RAID disk systems. Each disk system is logically composed of N data disks, 1 parity disk, and 1 redundant spare disk. The parity disk contains the bitwise parity of the N data drives. The extra disk will be configured into the system upon the occurrence of a disk failure.

In RAID a read of an object (a sector, block or track) can proceed as in a normal disk system. However, a write must have substantial extra processing. In particular, the old data object and the corresponding parity oject must be read. With this information, the RAID controller can determine the new contents of the parity object. Then, it writes both the new data and the new parity.

If a drive has a head crash, then RAID switches the extra drive into the disk array and starts a background task to reconstruct the contents of the dead disk onto the new one by reading the other N - 1 data disks plus the parity disk. If a user read occurs before a disk block is reconstructed, then the above procedure must be followed to obtain the lost contents on the fly.

The RAID controller will automatically stripe [LIVN86, SALE86] data across all N data disks on a track by track basis. The reason for track striping rather than sector striping is to obtain greater bandwidth

on sequential reads. With block striping, each disk will return one sector every rotation. With track striping, it can easily do track at a time reads and return a track in one rotation. This order of magnitude increase in disk bandwidth is desirable to exploit. Lastly, the RAID controller actually stripes the parity bits across all N + 1 drives so that writing the parity information for different I/O operations can take place in parallel. Thereby a RAID system can do a maximum of N parallel reads and N/2 parallel writes.

It will be useful to think of a collection of N+2 RAID disks as providing an abstraction for N **logical** disks, each one having T tracks and having a very long mean time between failures.

The initial RAID prototype is being constructed out of 3 1/2" drives with N = 4. Since the platters of such drives are not removable, a computer system with R such RAID disk systems can can be thought of as having a two dimensional storage system consisting of D = R * N logical drives each with the capacity of T tracks. FTD must construct a file system from this storage model as noted in the next section.

2.2. The Design of FTD

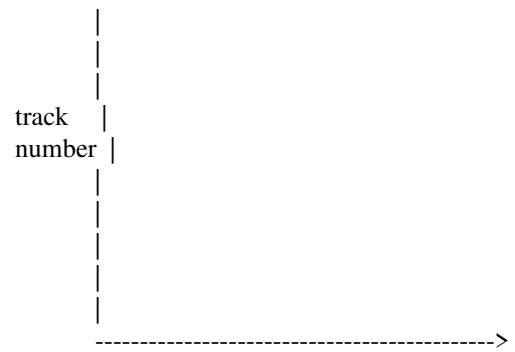
The objective of FTD is to deliver good performance in three different kinds of applications:

1) OLTP, such as TP1 [ANON85]. Here, random reads and writes of small (say 4K) blocks is the typical access pattern.

2) Large Objects. Here one requires the ability to materialize large objects, such as images, very quickly. Since, they will be stored by the data manager, POSTGRES, [STON86b, WENS88] as sequential bytes in some file, this environment requires very high performance sequential reads and writes of large number of bytes (say 10 mbytes).

3) Complex Queries. Here, XPRS will use intra-query parallelism to obtain high performance. As will be explained presently, a single user query will be decomposed into multiple parallel queries, each exploring a portion of the search space required to solve the query. These plans may be doing sequential or random reads; however, the important issue is to ensure that parallel plans do not collide for the use of disk arms.

In order to explain the alternative chosen, it is useful to visualize storage as the two dimensional space indicated in Figure 1. To obtain high sequential read and write performance, it is appropriate to scatter disk tracks from a single file over a large number of disks. Similarly, in order that multiple parallel subquery plans not collide for disk arms, it is again desirable that disk tracks in a single file occur on a large



The FTD Storage Model
Figure 1

number of drives.

Each file in FTD is a composed of an arbitrary collection of **extents.** The options available to FTD for extent allocation are:

1) Allocate each extent as a single disk track, i.e. a rectangle of height 1 and width 1.
2) Allocate each extent as a single, properly aligned RAID stripe, i.e a rectangle of height 1 and width N.
3) Allocate each extent as a horizontal rectangle of height 1 and width D.
4) Allocate each extent as a horizontal rectangle of height H and width D.
5) Allocate each extent as a vertical rectangle of height H and width 1.
6) Allocate each extent as a RAID vertical rectangle of height H and width N.
7) Allocate each extent as a general rectangle of height H and width W.
8) Allocate extents to storage areas that are not rectangular.

A detailed analysis of these options is presented in [SELT88]; here we simply make a few comments. First, if H is moderately large, the vertical schemes (5 and 6) will store the data for each file on a modest number of different disks, and the resulting arm contention under heavy load will compromise performance. Second, general rectangles present a significant optimization problem in placing new extents in available memory. Hence, (7) was discarded. Furthermore, non rectangular storage (8) was felt to entail a significant bookkeeping hassle and was discarded.

The remaining options differ significantly in how many extents a large file will require For option 1, each extent contains one 16K disk track, and hence a 10 gigabyte file requires 625,000 extents. Assuming disk track addresses require 4 bytes, then 2.5 mbytes are required to describe the storage for this file. Clearly, a hierarchical system of indirect tracks, such as present in UNIX would be required to efficiently access a structure of this size. On the other hand, if the user requested ten 1 gigabyte rectangles in option (4), then 10 extents would be sufficient, each requiring a starting track and a height. The description of this storage requires less than 100 bytes.

We assume that compact file description is valuable and quantify this notion with the **file-value** of a file as a function G of the number of extents, E. G is assumed to be a monitonically decreasing function of E. Furthermore, G is assumed to have a positive second derivitive (i.e. concave upward). Hence, increasing the number of extents by 1 will result in decreasing the file-value for larger value of E by lesser and lesser amounts.

In order to facilitate files with high file-values, we are implementing a variation on option (4) above that allows variable length extents. Our scheme can be explained most easily by numbering the disk tracks in Figure 1 from 1, ... , D * T so that track i and i + D are on the same logical drive. In other words, the logically next disk track is on the logically adjacent drive. The jth extent in a file is a logically contiguous chunk of storage specified by a starting track number Sj and a length Lj. As such, an extent begins at a track on a specific disk and and is then striped across all D disks |Lj/D| number of times where |x| is the largest integer less than x. The remainder of the variable length allocation continues sequentially on the next several disks so that the rectangle is one higher on some disks than on others.

Files smaller than one track will be supported by a **short** file construct which does block-at-a-time allocation from a collection of scratch tracks. Short files will not be considered further in this paper.

FTD will maintain a free space list in main memory. This free space is a collection of "holes" each of which has a starting address, A, and a length, L. FTD maintains the free space list in order of descending hole size.

Users of FTD can request new extents of specific lengths or additions to old extents of specific lengths. In either case, FTD receives a request for a new extent of B tracks and executes an algorithm to be presently described to find a hole to accommodate the request and then allocates the correct amount of storage from this hole. If no hole of sufficient size can be found, then FTD returns the largest hole available.

Consequently, we assume that maintaining large blocks of contiguous free space is desirable. Specifically, we assume that the **hole-value** of a chunk of free space is a function F(L) of its length L and that F has positive second derivitive F''.

Upon an allocation request of size E for a file with K extents, FTD will try to fulfill this request by maximizing the sum of all hole-values plus all file-values. There are only three candidates that can maximize this function.

1) Allocate the extent to the hole which has the minimum size, S1, larger than E, i.e. the "best fit" hole.

It is easy to show that any other hole will have its hole-value decreased by a greater amount than this one. Hence, among the holes that result in an additional extent being created, the smallest one is optimal and no others need be considered. In this case the total value of the storage system changes by:

delta-1 = F(S1) - F(S1 - E) + G(K) - G(K + 1)

2) Allocate the extent at the end of one which logically precedes it.

In this way, the previous extent is enlarged by allocating the new extent in the adjacent hole, H of size S2, and the number of extents in the file remains at K. In this case the total value changes by

delta-2 = F(S2) - F(S2 - E)

If space is not available adjacent to the previous extent because it is occupied by another extent X in a different file, then X must be moved to its "best fit" hole to make room. The computation of the value change in this case is modestly more complex than those above and is omitted for brevity.

3) Allocate the extent in the hole before the extent that follows it.

The calculation of value is analogous to (2) above.

Among these three options, FTD chooses the one which lowers the total file value the least. In the case that the function G is not present, this reduces to a classic best fit algorithm.

Moreover, it will often be desirable to rearrange storage to increase the size of free chunks or lower the number of extents. We accomplish this task and several other objectives through the use of an asynchronous storage compacter (called the **broom)** which is described in the next section.

2.3. The Broom

The broom has several functions. First, it can improve the total value of the storage system by moving extents. An extent of size E has three candidate better locations:

1) In the smallest hole larger than E

If the extent has holes of size X and Y on either side and is moved to a hole of size Z, then the total value of the storage system is improved by:

delta-3 = F(X + Y + E) - F(X) - F(Y) + F(Z - E) - F(Z)

and the improvement per track of disk movement is:

delta-3 / E

2) next to its predecessor extent.

In this way, the two extents can be merged into a single extent. If an extent must be moved out of the way, then it will be moved to the best hole available. The calculation for value change is straight forward and is omitted.

3) next to its successor extent.

We envison the broom as an asynchronous demon running at lowest priority which makes sweeps through all extents calculating the value of each of the three options above. The highest value is compared against a threshold, T. Moves which are above threshold are consumated; the remainder are discarded. This threshold is the average value plus one standard deviation of the best candidates during the preceding pass through the storage system. In this way, only the high profit moves are actually done.

If a candidate reallocation is performed, then the broom can, if necessary, move the allocated storage one track at a time, locking only the single track it is moving. In this way, the broom need not interfere with high availability.

The second purpose of the broom is to support addition and deletion of RAID disk systems. When a new RAID disk system of N logical disks is added to FTD, several actions must take place. For each of the T tracks of the disk system, FTD must inspect the adjacent track below the inserted disk system as well as the one above the disk system. If either is free, then the size of the appropriate hole can be expanded as the only FTD action. If the tracks on both sides belong to different extents, then a new hole can be inserted between them of size N. However, if the track on either side belong to the same extent E, then E must be split into two extents one to the right of the new disk system and one to the left. The file control block must be changed to reflect this split.

This sweep of the tracks can be done without taking the disk offline with the aid of a bit array. This array will have one bit per track indicating whether this track is "long", i.e. includes the new disk system or "short".

To take a RAID disk system offline is slightly more complex. FTD can modify the free space list to reclaim immediately all the unused storage on the selected disk system. Then, the broom can make a sweep to push allocated tracks onto other RAID devices. By keeping a track bit map for bookkeeping as above, allocations and deallocations can continue in parallel. When the bit map has all zeros the disk system is empty and can be removed.

The last purpose of the broom is to resolve bad load balance. Assume that there are a collection of **hot** tracks, i.e. ones that are accessed with higher frequency than other tracks. Also, assume that they are not hot enough to reside permanently in the buffer pool. FTD will naturally place such tracks randomly distributed over all D disks. However, the distribution of hot tracks is binomial and not uniform. Therefore, there will be disks with an excessive number of hot tracks and resulting arm activity. If a statistics system keeps a list of hot tracks and the average arm utilization on each drive, then the broom can resolve load imbalance in the following obvious way. When it moves an extent containing a hot track HT on an arm with {higher, lower} utilization than the average, it can move it to an arm with {lower, higher} utilization. This is easily accomplished by making a sub-optimal extent allocation if necessary. As a result of broom activity XPRS can be assured that each logical disk has approximately the same load.

Using this design the files corresponding to OLTP applications are striped across all D = R * N disks. Hence, a maximum of D reads or D / 2 writes can occur in parallel. Similarly, big objects can be read in parallel from all D disks. Lastly, complex queries have data spread across all drives and arm collisions Should be rare as explained in more detail in Section 3.

3. THE XPRS OPTIMIZER

3.1. Introduction

XPRS stores data base objects (classes or relations) one per file in FTD. When a query is specified to the system, the optimization takes place in two steps. First, a heuristically optimal plan is constructed without regard to main memory or parallelism considerations. This plan is constructed when the optimizer is run, often in advance of query execution and assumes the entire buffer pool is available. Hence, this plan can be called the "single user" plan and it is divided into a collection of subplans called **fragments** which are the basis for XPRS parallelism.

The second step of optimization is performed during execution by the run-time system. At this time, available main memory and desired amount of parallelism are factored into the plan. This step is

accomplished by first determining a target amount of parallelism for each fragment. Then, each fragment is decomposed if insufficient memory is available. If decomposition is not possible, then XPRS will lower the number of batches [DEWI84] by which a hash join is processed.

We turn now to discussing each of the steps in the optimization process in detail.

3.2. Optimizer Plans

There are three popular tactics to solve join queries, iterative substitution, merge-sort and hash joins. Clearly, iterative substitution must be retained to solve non-equijoins. Merge-sort joins will not be used in XPRS because there is such a limited range of memory sizes for which they outperform hash joins. For example, to join two 10 gigabyte relations with a disk block size of 4K, one requires 6.4 mbytes of buffer space for a hybrid hash join [DEWI84] to outperform merge sort. We expect to routinely have much more buffer space than this amount; hence merge-sort will not be used by XPRS.

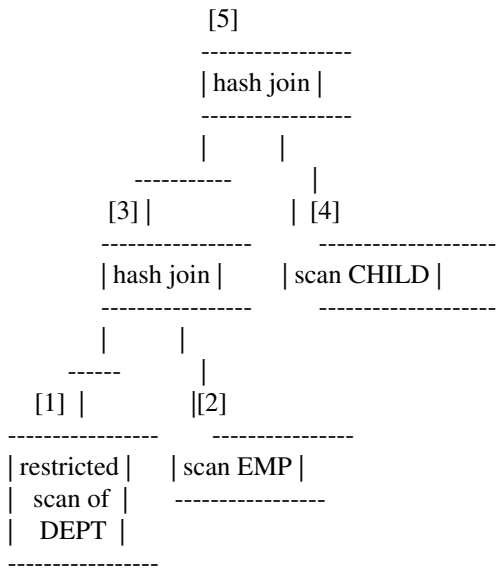
Moreover, in this paper we will not concern ourselves with aggregates or aggregate functions for simplicity. Hence, the kinds of nodes which can occur in an XPRS plan are scan nodes, restricted scan nodes, iterative substitution nodes, and hash-join nodes.

An example plan containing scan and hash-join nodes is shown in Figure 3 for the query:

```
retrieve (CHILD.name) where CHILD.parent = EMP.name and
        EMP.dept = DEPT.dname and DEPT.floor = 1
```

Our optimizer will, of course, choose the plan which minimizes a weighted cost of I/O operations and CPU cost as in [SELI79]. As noted above it will assume that the total buffer pool is available for use of this query. In addition, it must place extra information about each node into the optimal plan, namely:

1) the total number of I/Os expected to process the node
2) the expected number of CPU cycles to process the node
3) the expected size of the hash table in each hash-join node
4) an integer indicating the order of execution of the nodes
5) an integer, NBATCH, indicating how many batches are required for hash-join nodes



An Example XPRS Plan
Figure 3

This information will be used by the execution engine in its scheduling decisions as discussed in the next subsection.

The final step taken by the optimizer is to decompose the plan into a collection of **plan fragments.** The following simple algorithm generates a unique collection of fragments. Node 1 is included in the first plan fragment along with any iterative substitution nodes directly above it and ending with the first node that is either:

a hash join node or
an iterative substitution node that forms a temporary relation

In our example, fragment 1 contains nodes 1 and 3. Fragment 2 then contains the lowest number leaf node, in this case node 2, and all direct ancestors up through the first hash-join or iterative-substitution-to-temporary node that has not already been allocated to a plan fragment. In this case fragment 2 will contain nodes 2, 3 and 5. The last fragment will contain nodes 4 and 5.

Notice that each hash-join node appears in two fragments. The code to build the hash table will be run in the first plan fragment and the subsequent hash table lookup appears in the second fragment.

The purpose of plan fragments is as a basis for parallelism. In the case that sufficient memory is available to hold the hash tables in node 3, then fragment 1 can be processed by an arbitrary number of parallel subplans. Each such subplan will be responsible for a portion of the restricted scan of DEPT and then performing the first part of the hash join in node 3. Specifically, they must hash the qualifying DEPT records on the field dname into a main memory hash table. The allocation of DEPT tuples to parallel subplans will be discussed in the next subsection. When all the parallel fragment 1 subplans complete, then fragment 2 can commence. If sufficient main memory for the hash table in node 5 can be allocated (in addition to the space for the hash table in node 3), then fragment 2 can be decomposed into an arbitrary number of subplans each responsible for a portion of the scan of EMP. For each qualifying EMP record, the subplans can process the rest of node 3, namely looking up the dept field in the hash table built by fragment 1 and finding all matches. Next, they will hash the resulting records on the name field and construct the hash table needed in node 5. When all parallel fragment 2 subplans complete, then the space for the hash table in node 3 can be released and processing of fragment 3 can commence using an arbitrary amount of parallelism.

Clearly, plan fragments can be no larger than the ones produced by our algorithm. The last node in a plan fragment is one where subsequent processing cannot begin until **all** parallel subplans implementing the plan fragment have completed. Hence, the last node corresponds to a needed synchronization point. Of course, it is possible to construct plan fragments smaller than those in our algorithm. In a subsequent section, we will observe situations where this is required because of limited amounts of main memory. Unless required for this reason, smaller plan fragments are a bad idea. First, they introduce addition synchronization points leading to additional overhead. Second, every plan fragment ends with the construction of a temporary relation. Clearly, smaller plan fragments lead to additional temporaries and additional overhead.

3.3. The XPRS Execution Engine

The XPRS execution engine schedules plan fragments to collections of processes at run time based on several considerations, including available main memory, load average and synchronization overhead as explained below. We first treat the ample memory case then we specialize the solution to the case where restricted amounts of main memory are available.

3.3.1. Ample Main Memory

We treat in this subsection the case that available main memory can be obtained for all temporaries needed by the plan fragment with the largest memory requirements. In this case, all plans constructed by the optimizer will have NBATCH = 1.

First, the maximum parallelism desirable in a fragment is calculated. The cost of the fragment in CPU cycles is determined by adding the CPU cost in each node in the fragment. If N parallel processes execute the plan, there will be a CPU cost per process to start up the subplan, synchronize it and then

terminate it. This cost depends significantly on whether the operating system in use has a notion of light weight processes or not. Hence we leave this cost as a parameter, Q. Considering only CPU costs, the desirable maximum number of processes, M1, to allocate to a fragment is:

M1 = (1/20) * (CPU cost of the fragment) / Q

For example, if a fragment costs 10 sec of CPU time and the overhead of a subplan is 25 msec, then 20 processes should be scheduled for the fragment and each will perform 500 msec of useful work and 25 msec of overhead, i.e. 95 percent useful work.

However, I/O parallelism must also be considered. If asynchronous I/O can be performed, then there is no reason to have more than one subplan per disk arm. On the other hand, if only synchronous I/O is possible, then two plans per arm will be required to fully utilize the arm. Hence, we assume that the maximum number of desirable plans is K times the number of drives, for some parameter K. However, there is little reason to allocate such a large number of plans if each of them has only a small number of I/O operations to perform. Hence, the desirable maximum number of processes from an I/O perspective is:

M2 = min (expected number of I/Os in the fragment / 2, K * D)

Consequently, the overall desirable maximum number of processes, M, is:

M = min (M1, M2)

However, if the load average is high, M may be an excessive number of processors. For example, suppose we have a workload of 10 percent long queries and 90 percent short queries on a 20 processor computer system. Assume short queries consist of one fragment and are executed with M = 1 while long queries consist of a single fragment with M = 40. Under the assumption that each executing subplan receives an equal fraction of the total machine resources, then allocating one subplan to each fragment of each long query will result in 10 percent of system resources being allocated to long queries. On the other hand, if 40 subplans are allocated to each fragment of a long query, then 82 percent of system resources will go to long queries. Clearly, the fraction of the total machine that is allocated to the long queries is dependent on the amount of parallelism that is used on the long queries.

XPRS dynamically uses the current system load average per processor (LAP), i.e. the number of jobs in the ready queue divided by the number of processors, P to make an intelligent decision. Hence, the target fragment parallelism, TP, will be:

TP = M / (LAP + 1) ** 2

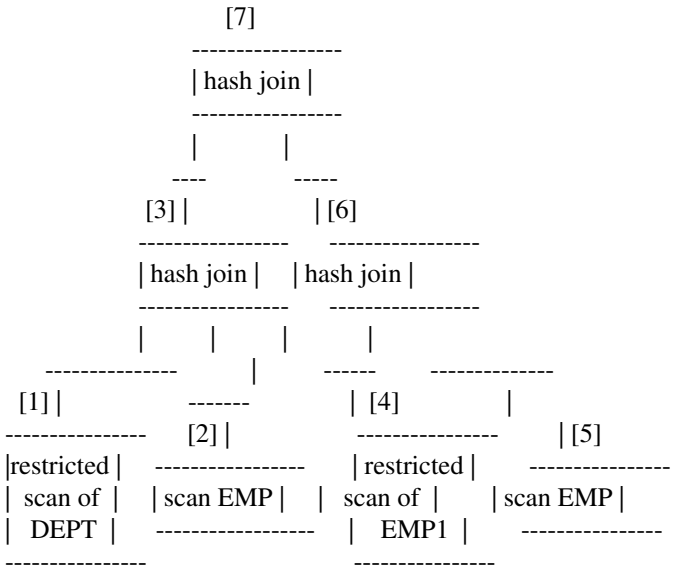
Therefore, the next fragment is scheduled as TP parallel subplans. The execution engine decomposes the fragment into TP parallel sub-fragments by decomposing the scan node at the bottom of the fragment so that each of the TP subplans reads every TP-th data track in the relation to be scanned. If the scan node involves the use of an index, then the logical scan interval is decomposed into TP nearly equal sized logical intervals by special access method calls.

There is one final consideration concerning XPRS parallelism. It is possible that TP is equal to M, in which case the actual parallelism is equal to the maximum desirable parallelism. In this case XPRS looks for another plan fragment to run in parallel with the existing one so that additional parallelism can be utilized. Two plan fragments can be executed in parallel if they have no node in common. Often no additional fragments meet this criteria; however, the following example illustrates a situation where additional parallelism is possible. Consider the query:

retrieve (EMP.name) where EMP.dept = DEPT.dname and DEPT.floor = 1
and EMP.salary > EMP1.salary and EMP1.age > 30

Figure 4 indicates a conceivable optimal plan for this query. In this case there are four fragments as follows:

fragment 1: nodes 1 and 3
fragment 2: nodes 2, 3, and 7
fragment 3: nodes 4 and 6



Another XPRS Plan
Figure 4

fragment 4: nodes 5, 6, and 7

Hence, fragments 1 and 3 can be executed in parallel if sufficient main memory is available because they have no node in common.

3.3.2. Limited Memory Case

In this situation limited main memory is available, and there are two consequences. First, the XPRS optimizer may have set NBATCH to a number larger than one because some temporary exceeds the size of the buffer pool. Second, concurrently executing commands from other users may require a query to run with less memory than it would use in a single-user environment. Hence, fragments must sometimes compress their memory demands in order to efficiently run with available main memory.

When a plan fragment wishes to run it must explicitly RESERVE sufficient main memory for its needs with a call to the buffer manager. If it makes a successful memory request, it begins execution with its chosen degree of parallelism. Otherwise, the buffer manager returns an indication of the amount of memory currently available. In this event, the plan fragment has two choices:

1) wait for additional memory to become available
2) alter the plan fragment to consume less memory

This decision is made based on the composition of the plan fragment and on the amount of memory currently RESERVED by the plan of which this fragment is a part as presently explained.

When a plan fragment completes, it FREEs the space for any temporaries no longer needed. The buffer manager must maintain the sum of all successful requests and ensure that this amount does not exceed available memory. In this manner, it will ensure that two large joins do not thrash against each other for memory. Notice that the buffer manager is free to implement any replacement algorithm it wishes, e.g. [CHOU85, SACC86] and need not guarantee that each plan fragment always has its exact quota of buffer pool pages.

We now indicate the computation made to determine the the size of a fragment's memory request. If the top node is a hash join node, no other nodes are hash joins, and NBATCH = 1, then the plan fragment

must request an amount of memory equal to the size of the hash table. On the other hand, to construct a hash table of size SIZE using TP processes with NBATCH > 1 requires NBATCH output buffers. However, if the remainder of the hash-join is executed with TP processes, then they will require buffer space of:

TP * SIZE / NBATCH

Hence, the initial plan fragment will request:

max (NBATCH - TP, TP * SIZE / NBATCH)

The second term is the memory for TP batches while the first is output buffers for the remaining NBATCH - TP batches.

In the case that both the top node and one or more interior nodes are hash join nodes, then memory must be reserved for the top node as above. However, memory will have been previously reserved for each intermediate hash join node in a previous plan fragment. Suppose the parallelism assumed by the previous fragments are TP1, ..., TPn. If the current fragment wishes to run with parallelism TP, then space must be reserved or released for the interior nodes. The current plan fragment can alter its space request in the obvious way to deal with this situation.

We now turn to the strategy to follow if a memory request fails. Different actions are desired depending on whether the the plan of which this fragment is a part is currently holding any buffer space. First, if no memory is held by the plan, then this fragment will wait and make a memory request later. The wait interval is set to be 1/10 of its expected processing time. If it fails in three successive attempts, it will then alter the plan fragment as noted below. If the plan is currently holding memory, then XPRS will always choose to alter the plan to consume less memory as explained below.

There are three ways to alter the plan fragment to lower memory and they will be tried in order using the first one that succeeds. First, the fragment can be **decomposed** into two sequential fragments. This strategy entails cutting the plan fragment just beneath the final node and then running the lower part first. Hence, an N node plan fragment is decomposed into an N-1 node initial piece and a 1 node final piece. Since the N-1 node initial piece needs no memory, it can be run immediately and can write output tuples to a disk temporary. When this temporary has been constructed, any memory used by the N-1 nodes in the plan fragment can be released. The last node of the plan fragment now makes a new memory request.

For example, consider fragment 2 from the plan mentioned in Figure 4. If fragment 1 was successful, then a main memory hash table exists for node 3. If space cannot be obtained for the hash table in node 7, then fragment 2 can be cut into two pieces, namely nodes 2 and 3 go to the lower piece while node 7 becomes the upper piece. Nodes 2 and 3 can be run immediately producing a disk temporary with no extra memory required. Once this piece is complete, the memory used by node 3 can be released and the second half of the plan (node 7) can be attempted. This sequential decomposition will ensure that a plan never requires two main memory hash tables during execution. Decomposition is always attempted on a plan fragment as the preferred tactic.

The second way to lower space consumption is to increase the value of NBATCH for the hash table being constructed. Specifically, if there are TP processes executing a hash-join, then the number of batches, V, which minimizes memory consumption is:

V = min (NBATCH - TP, SIZE * TP / NBATCH)

i.e.

V ˜ sqrt (SIZE * TP)

Consequently, if B is the size of available main memory and

B > sqrt (SIZE * TP)

then raise NBATCH to

NBATCH = SIZE * TP / B

and leave TP output buckets in main memory during the build phase of the hash join.

The third alternative is to lower the parallelism of the plan fragment. Consequently, if

B < sqrt (SIZE * TP)

then lower TP until memory consumption is less than B. Of course, if

B < sqrt (SIZE)

then the fragment cannot be run at all, and a wait for more memory cannot be avoided.

4. CONCLUSIONS

This paper has indicated how parallelism in XPRS will be exploited. The number of parallel plans is determined by the load average on the system being used and by main memory considerations. Each parallel plan executes a portion of a scan together with all possible nodes above it in the query tree. Because FTD spreads files maximally across disks, there should be no collisions for arms in exploiting this parallelism. Hence, load balance among processors and disk arms should be achieved.

Lastly, there is one synchronization necessary at the end of each each fragment which entails local messages only. This should be contrasted with shared nothing proposals which have more difficulty with processor and arm load balance and which must fundamentally pay more messages. Hence, we expect XPRS to be faster than shared nothing systems with equivalent amounts of total hardware. The simulations in [BHID88] show the potential win to be as much as a factor of 2. We expect to verify this experimentally with a working system shortly.

REFERENCES

[ANON85] Anon et. al., "A Measure of Transaction Processing Power," Tandem Computers, Cupertino, CA, Technical Report 85.1, 1985.

[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.

[BITT83] Bitton, D. et. al., ''Parallel Algorithms for the Execution of Relational Database Operations,'' ACM-TODS, Sept. 1983.

[BORR88] Borr, A. and Putzolu, F., ''High Performance SQL Through Low-level System Integration,'' Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Il., June 1988.

[CHOU85] Chou, H. and Dewitt, D., ''An Evaluation of Buffer Management Strategies for Relational Database Systems,'' Proc. 1985 VLDB Conference, Stockholm, Sweden, August 1985.

[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.

[DEWI88] Dewitt, D. et. al., ''A Performance Analysis of the Gamma Database Machine,'' Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Il, June 1988.

[GRAY87] Gray, J. et. al., "NON-STOP SQL," Proc. 2nd International Workshop on High Performance Transaction Systems, Asilomar, CA, Sept. 1987.

[LIVN86] Livny, M. et. al., "Multi-disk Management Algorithms," IEEE Database Engineering, March 1986.

[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, Ill., 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.

[ROTH80] Rothnie, J., et. al., ''Introduction to a System for Distributed Databases (SDD-1),'' ACM-TODS, March 1980.

[SACC86] Sacco, G. and Schkolnick, M., ''Buffer Management in Relational Database Systems,'' ACM-TODS, Dec. 1986.

[SALE86] Salem, K. and Garcia-Molina, H., "Disk Striping," Proc. 1986 IEEE Data Engineering Conference, Los Angeles, CA, February 1986.

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

[SELT88] Seltzer, M., ''Analysis of Extent Allocation Policies in File Systems,'' Masters Report, EECS Dept., University of California, Berkeley, Ca., Dec. 1988.

[STON83] Stonebraker, M., et. al., ‘‘Performance Analysis of a Distributed Data Base System,’’ Proc. 3th Symposium on Reliability in Distributed Software and Data Base Systems, Clearwater, Fla, Oct. 1983

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

[STON86b] Stonebraker, M. and Rowe, L., “The Design of POSTGRES,” Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

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

[TERA85] Teradata Corp., “DBC/1012 Data Base Computer Reference Manual,” Teradata Corp., Los Angeles, CA, November 1985.

[WENS88] Wensel, S. (ed.), “The POSTGRES Reference Manual,” Electronics Research Laboratory, University of California, Berkeley, CA, Report M88/20, March 1988.