THE DESIGN OF XPRS

Michael Stonebraker, Randy Katz, David Patterson, and John Ousterhout
EECS Dept.
University of California, Berkeley

Abstract

This paper presents an overview of the techniques we are using to build a DBMS at Berkeley that will simultaneously provide high performance and high availability in transaction processing environments, in applications with complex ad-hoc queries and in applications with large objects such as images or CAD layouts. We plan to achieve these goals using a general purpose DBMS and operating system and a shared memory multiprocessor. The hardware and software tactics which we are using to accomplish these goals are described in this paper and include a novel "fast path" feature, a special purpose concurrency control scheme, a two-dimensional file system, exploitation of parallelism and a novel method to efficiently mirror disks.

1. INTRODUCTION

At Berkeley we are constructing a high performance data base system with novel software and hardware assists. The basic goals of XPRS (eXtended Postgres on Raid and Sprite) are very high performance from a general purpose DBMS running on a conventional operating system and very high availability. Moreover, we plan to optimize for either a single CPU in a computer system (e.g. a Sun 4) or a shared memory multiprocessor (e.g a SEQUENT Symmetry system). We discuss each goal in turn in the remainder of this introduction and then discuss why we have chosen to exploit shared memory over shared nothing or shared disk.

1.1. High Performance

We strive for high performance in three different application areas:

1) transaction processing
2) complex ad-hoc queries
3) management of large objects

Previous high transaction rate systems have been built on top of low-level, hard-to-program, underlying data managers such as TPF [BAMB87] and IMS/Fast Path [DATE84]. Recent systems which are optimized for complex ad-hoc queries have been built on top of high-function data managers (e.g. GAMMA [DEWI86] and DBC/1012 [TERA85]); however such systems have used custom low-level operating systems. Lastly, applications requiring support for large objects (such as images or CAD geometries) have tended not to use a general purpose data manager because of performance problems.

This research was sponsored by the Defense Advanced Research Projects Agency under contract N00039-84-C-0089, the Army Research Office under contract DAAL03-87-G-0041, and the National Science Foundation under contract MIP-8715235.

The first goal of XPRS is to demonstrate that high performance in each of these areas can be provided by a next generation DBMS running on a general purpose operating system without unduly compromising performance objectives. Clearly, this will be a major advantage as it will bring the benefits of ease of application construction, ease of application migration, data independence and low personnel costs to each of these areas. Specifically, we are using a slightly modified version of POSTGRES [STON86] as the underlying data manager and the Sprite network operating system [OUST87]. Our concrete performance goals for XPRS in each of the three application areas are now described.

We feel that general purpose CPUs will obey Joy's law of:

MIPS = 2 ** (year - 1984)

at least for the next several years. As such, we are confident that single processor systems of 50-100 MIPS will appear by 1991 and that shared memory multiprocessors of several hundred MIPS will also occur in the same time frame. Consequently, we expect that CPU limitations of current transaction processing engines will become less severe in the future. Therefore we strive only for ''good'' performance on TP1 [ANON85] which has come to be the standard benchmark by which transaction processing performance is measured, i.e:

100,000 instructions per transaction
or
XACTS/sec = 10 * MIPS

For example, a 100 MIPS system should be capable of 1000 TP1s per second.

Achieving this goal requires system tuning, the removal of critical sections in the code, and avoiding lock contention on so-called **hot spots,** i.e. records or blocks with high traffic. To alleviate this latter problem, it is necessary for the DBMS to hold locks for shorter periods of time, and we are planning to use two tactics to help in this regard. First, we plan to run the various DBMS commands in a transaction in parallel where possible to shorten the amount of time they hold locks. In addition, we are using a special purpose concurrency control algorithm which avoids locking altogether in certain situations. These tactics are described later in this paper.

Some researchers believe that high performance is achieved easily by using a low-function high-performance DBMS such as IMS/Fast Path. Others question the propriety of removing DBMS services such as query optimization and views and suggest utilizing only high level interfaces. Clearly, the elimination of function from the path length of high traffic interactions is a possible optimization strategy. Although we are not enthusiastic about this strategy because we feel that the potential problems outweigh the advantages, we are going to allow exploitation of this concept by implementing a novel **fast path** scheme to achieve a **variable speed interface.** We also discuss this tactic later in this paper.

Although transaction processing systems typically perform short tasks such as TP1, there are occasional longer running transactions. Such transactions will be a clear locking bottleneck, unless they can be parallelized. Parallel execution of single commands has been addressed in [DEWI85, DEWI86, RICH87]. In this paper we sketch a slightly different approach that focuses on a multi-user environment, the presence of shared memory and the necessity of carefully exploiting a large amount of main memory. Our performance goal is to outperform recent systems such as GAMMA and the DBC/1012 on ad-hoc queries using comparable amounts of hardware.

Lastly, high performance is necessary in engineering environments where large objects are stored and retrieved with great regularity. A typical image might be several megabytes and an application program that processes images requires retrieval and storage of such objects at high bandwidth. Current commercial systems tend not to store such objects at all, while prototype extendible systems (e.g. POSTGRES [STON86] and EXODUS [CARE86]) have been designed with object management needs in mind. The current design of both systems will limit the speed at which large objects can be retrieved to the sequential reading speed of a single disk (about 1.5 mbytes/sec for current low-end disks). Hence, a 64 mbyte object will require about 43 seconds of access time. Especially if a supercomputer is the one making the request, it is unreasonable to require such delays. Our last goal of XPRS is an order of magnitude improvement in access times to large objects. We plan to achieve this goal with a variable-speed two-dimensional file

system which is described below.

1.2. High Availability

A second goal of XPRS is to make data base objects unavailable as infrequently as possible. There are two common causes of data unavailability, errors and locking problems, and we discuss each in turn. Errors have been classified by [GRAY87] into:

hardware errors (e.g. dish crashes)
software errors (e.g. OS or DBMS crashes)
operator errors (e.g. accidental disk erasure)
environment errors (e.g. power failure)

Because we are designing an I/O system, our goal is to make a contribution to improved availability in this area in the presence of hardware errors. Our initial ideas have resulted in an efficient way to mirror disks at much reduced storage costs [PATT88]. On the other hand, in the case of CPU failures, we assume that XPRS would be part of a distributed data base system such as Non-stop SQL [GRAY87A], INGRES/STAR [RTI87] or ORACLE/STAR. Availability in the presence of CPU or memory failures is provided in such systems by traditional distributed DBMS multi-copy techniques. Hence, this area is not discussed further in this paper.

Software errors are soon likely to be the dominant cause of system failures because hardware and operator errors are declining in frequency and environment errors are largely power problems which can be avoided by uninterruptable power supplies [GRAY87]. It should be noted that many current techniques for achieving high availability (such as process-pairs for non-stop operation and mirrored disks) are vulnerable to software errors. Obviously, an errant DBMS will write bad data on each disk in a mirrored pair, and the backup process may write the same bad data that caused its mate to fail. Hence, our assumption in XPRS is to realize that software errors will happen. Therefore, our goal in XPRS is to recover from software crashes in seconds. The tactics which we have in mind are described later in the paper and depend on the characteristics of the storage system built into POSTGRES [STON87].

Operator errors are best avoided by having no operator. Hence, another goal of XPRS is to perform all utility functions automatically, so that an operator-free environment is feasible. Among other things an operator-free environment requires that the system self-adapt to adding and deleting disks, automatically balance the load on disks arms, and create and drop indexes automatically.

In summary our goals for XPRS are improved I/O system availability in the event of hardware errors, ensuring that data is not lost as a result of software errors, instantaneous recovery from software errors, and a system capable of running with no operator.

A second availability goal in XPRS is never to make a data base object unavailable because of locking problems. These result from running large (usually retrieval) commands which set read locks on large numbers of data objects, and from housekeeping chores performed by the data manager. For example, in current systems it is often advisable to reorganize a B-tree index to achieve physical contiguity of index pages after a long period of splits and recombinations of pages. Such a reorganization usually makes the index unavailable for the duration of the reorganization and perhaps the underlying relation as well. In XPRS this and all other housekeeping chores must be done incrementally without locking a relation or taking it off line.

1.3. Shared Memory

In contrast to other recent high performance systems such as NONSTOP SQL [GRAY87A], GAMMA [DEWI86] and the DBC 1012 [TERA85] which have all used a shared-nothing [STON86A] architecture, we are orienting XPRS to a shared memory environment. The reason for making this choice is threefold. First the memory, bus bandwidth and controller technology are at hand for at least a 2000 TP1/sec system. Hence, speed requirements for advanced applications seem achievable with shared memory. Moreover, a 2000 TP1/sec. system will require around 8000 I/Os per second, i.e. about 250 drives. The aggregate bandwidth of the I/Os, assuming 4K pages, is about 32 mbytes/sec. Current large

mainframes routinely attach this number of drives and sufficient channels to deal with the bandwidth.

Second, the reason to favor shared memory is that it is 25 to 50 percent faster on TP1 style benchmarks than shared nothing [BHID88] and has the added advantage that main memory and CPUs are automatically shared and thereby load balanced. Hence, we avoid many of the software headaches which are entailed in shared nothing proposals.

Lastly, some people criticize shared memory systems on availability grounds. Specifically, they allege that shared nothing is fundamentally more highly available than shared memory, because failures are contained in a single node in shared nothing system while they corrupt an entire shared memory system. Certainly this is true for hardware errors.

On the other hand, consider software errors. With conventional log-based recovery a shared nothing system will recover the failed node in a matter of minutes by processing the log. A shared memory system will take at least as long to recover the entire system because it will have a larger log, resulting in lower availability. However, suppose the operating system and the data manager can recover instantaneously from a software or operator error (i.e. in a few seconds). In this case, both a shared memory and shared nothing system recover instantly, resulting in the same availability.

In summary, we view a shared memory system as nearly equally available as an equivalent shared nothing system. In addition, we view XPRS as having higher availability than any log-based system because it recovers instantly from software errors. Moreover, a shared memory system is inherently higher performance than shared nothing and easier to load balance.

In the rest of this paper we discuss the solutions that we are adopting in XPRS to achieve these goals. In Section 2 we present our tactics to provide high performance on transaction processing applications. Specifically, we discuss our **fast path** mechanism that is being added to POSTGRES to cut overhead on simple transactions. Moreover, we discuss inter-query parallelism which can cut down on the length of time transactions hold locks. Finally, we consider a specialized concurrency control system which avoids locks entirely in some situations. Section 3 then turns to performance techniques applicable to complex commands. We present our approach to query processing in this section which attempts to achieve intra-query parallelism to improve response time as well as make excellent use of large amounts of main memory. Then, in Section 4 we indicate how to achieve high performance when materializing large objects. We first argue that traditional file systems are inadequate solutions in our environment and suggest a novel two-dimensional file system that can provide a **variable speed** I/O interface.

Section 5 continues with our ideas for achieving high availability in the presence of failures. We briefly discuss hardware reliability and suggest a novel way to achieve ultra-high disk reliability. Then, we turn to software techniques which can improve availability and indicate how we expect to achieve instantaneous recovery. Section 6 then closes with our algorithms to avoid data unavailability because of locking problems.

2. TRANSACTION PROCESSING PERFORMANCE

This section explores three tactics relevant to transaction processing performance.

2.1. Fast Path

It is common knowledge that TP1 consists of 4 commands in a query language such as SQL [DATE84] or POSTQUEL [ROWE87] together with a begin XACT and an end XACT statement. If an application program gives these commands to the data manager one at a time, then the boundary between the data manager and the application must be crossed in both directions 6 times. Moreover, if the application runs on a workstation and the DBMS runs on a host, the messages must go over the network, increasing the cost of the boundary crossing. This overhead contributes 20-30 percent of all CPU cycles consumed by a TP1 transaction. To alleviate this difficulty, several current commercial systems support procedures in the data base and POSTGRES does likewise. Hence, TP1 can be defined as a procedure and stored in a relation, say

COMMANDS (id, code)

and then later executed as follows:

execute (COMMANDS.code with check-amount, teller#) where COMMANDS.id = “TP1”

In this way the TP1 procedure is executed with the user supplied parameters of the check amount and the teller number, and the boundary between POSTGRES and an application will be crossed just once.

To go even faster POSTGRES makes use of user-defined functions. For example, a user can write a function OVERPAID which takes an integer argument and returns a boolean. After registration with the data manager, any user can write a query such as:

retrieve (EMP.name) where OVERPAID (EMP.salary)

Of course it is also legal to execute a simpler query such as:

retrieve (result = OVERPAID (7))

which will simply evaluate OVERPAID for a constant parameter.

Fast Path is a means of executing such user defined functions with very high performance. Specifically, we have extended the POSTQUEL query language to include:

function-name(parameter-list)

as a legal query. For example,

OVERPAID(7)

could be submitted as a valid POSTGRES query. The run-time system will accept such a command from an application and simply pass the arguments to the code which evaluates the function which is linked into the POSTGRES address space without consuming overhead in type checking, parsing or query optimization. Hence, there will be 200 or fewer instructions of overhead between accepting this command and executing it. Using this facility (essentially a remote procedure call capability) TP1 can be defined as a POSTGRES function and a user would simply type:

TP1(check-amount, teller-name)

The implementation of the TP1 function can be coded in several ways. The normal implementation would be for the function to simply execute preconstructed query plans for the four commands which make up TP1. This will result in a somewhat faster implementation of TP1 than executing a stored procedure because of the reduced overhead in the procedure call.

Alternately, an application designer can more directly control the low level routines in POSTGRES. POSTGRES has user defined access methods, which correspond to a collection of 13 functions described in [WENS88]. These functions can be directly called by a user written procedure. As a result, high performance can be achieved by coding TP1 directly against the access method level of POSTGRES. Although this provides no data independence, no type checking and no integrity control, it does allow the possibility of excellent performance. Moreover, there are also interfaces to the buffer manager as well as higher level interfaces in the query execution system.

Using this technique of user written procedures, POSTGRES can provide a **variable speed interface.** Hence, a transaction can make use of the maximum amount of data base services consistent with its performance requirements. This approach should be contrasted with “toolkit” systems (e.g. EXODUS [CARE86]) which require a data base implementor to build a custom system out of tool-kit modules.

2.2. Inter-query Parallelism

There is no reason why all commands in TP1 cannot be run in parallel. If any of the resulting parallel commands fails, then the transaction can be aborted. In the usual case where all commands succeed, then the net effect will be that locks are held for shorter periods of time and lock contention will be reduced. Since an application program can open multiple **portals** to POSTGRES, each can be executing a parallel command. However, we expect high performance systems to store procedures in the system which are subsequently executed. A mechanism is needed to expedite inter-query parallelism in this situation. Although it is possible to build a semantic analyzer to detect possible parallelism [SELL86], we are

following a much simpler path. Specifically, we are extending POSTGRES with a single keyword **parallel** that can be placed between any two POSTGRES commands. This will be a marker to the run-time system that it is acceptable to execute the two commands in parallel. Hence, TP1 can be constructed as four POSTQUEL commands each separated from its neighbor by **parallel.**

The reason for this approach is that it takes, in effect, a theorem prover to determine possible semantic parallelism, and we do not view this as a cost-effective solution.

2.3. Special Purpose Concurrency Control

We make use of a definition of **commutative** transactions in this section. Suppose a transaction is considered as a collection of atomic actions, a1, ..., am; each considered as a read-modify-write of a single object. Two such transactions T1 and T2, with actions a1, ...,am and b1,..., bn will be said to **commute** if any interleaving of the actions of the two transactions for which both transactions commit yields the same final data base state. In the presence of transaction failures we require a somewhat stronger definition of commutativity which is similar to the one in [BADR87]. Two transactions will be said to **failure commute** if they commute and for any initial data base state S and any interleaving of actions for which both T1 and T2 succeed, then the decision by either transaction to voluntarily abort cannot cause the other to abort.

For example, consider two TP1 withdrawals. These transactions commute because both will succeed if there are sufficient funds to cover both checks being written. Moreover, both withdrawals failure commute because when sufficient funds are present either transaction will succeed even if the other decides to abort. On the other hand, consider a $100 deposit and a $75 withdrawal transaction for a bank balance of $50. If the deposit is first, then both transactions succeed. However, if the deposit aborts then the withdrawal would have insufficient funds and be required to abort. Hence, they do not failure commute.

Suppose a data base administrator divides all transactions in XPRS into two classes, C1 and C2. Members of C1 all failure commute, while members of C2 consist of all other transactions. Moreover, he provides an UNDO function to be presently described. Basically, members of C1 will be run by XPRS without locking interference from other members of C1. Members of C2 will be run with standard locking to ensure serializability against other members of C2 and also for members of C1.

To accomplish this POSTGRES must expand the normal read and write locks with two new lock types, C1-read and C1-write. Members of C1 set these new locks on objects that they respectively read and write. Figure 1 shows the conflict table for the four kinds of locks: Obviously, C1 transactions will be run against each other as if there was no locking at all. Hence, they will never wait for locks held by other members of C1. Other transactions are run with normal locks. Moreover, C1-read and C1-write locks function as ordinary locks with regard to C2 transactions.

	R	W	C1-R	C1-W
R	ok	no	ok	no
W	no	no	no	no
C1-R	ok	no	ok	ok
C1-W	no	no	ok	ok

Compatibility modes for locks.
Figure 1.

Because multiple C1 transactions will be processed in parallel, the storage manager must take care to ensure the correct ultimate value of all data items when one or more C1 transactions aborts. Consider three transactions, T1, T2 and T3 which withdraw respectively $100, $50 and $75 from an account with $175 initially. Because the POSTGRES storage system is designed as a no-overwrite storage manager, the first two transactions will write new records as follows:

initial value:	$175
next value:	$75 written by T1 which is in progress
next-value:	$25 written by T2 which is in progress

The transaction T3 will fail due to insufficient funds and not write a data record.

POSTGRES must be slightly altered to achieve this effect. It currently maintains the ''state'' of each transaction in a separate data structure as:

committed
aborted
in progress

To these options a fourth state must be added:

C1-in-progress

Moreover, POSTGRES must return data records that are written by either C1-in-progress or committed transactions to higher level software instead of just the latter. The locking system will ensure that these C1-in-progress records are not visible to C2 transactions. Using this technique C1 updates are now immediately visible to other concurrent C1 transactions as desired.

When an ordinary transaction aborts, POSTGRES simply ignores its updates and they are ultimately garbage-collected by an asynchronous vacuum cleaner. However, if a C1 transaction aborts, there may be subsequent C1 transactions that have updated records that the aborted transaction also updated. For example, if T1 aborts, the state of the account will be:

initial value:	$175
next value:	$75 written by T1 which aborted
next-value:	$25 written by T2 which is in progress

Since all the versions of an individual record are compressed and chained together on a linked list by the storage manager, this situation will occur if the storage manager discovers a record written by an aborted transaction followed by one or more records written by a transaction with status “C1-in-progress” or “commit”.

In this case, the storage manager must present the correct data value to the next requesting transaction. Consider the record prior to the aborted transaction as “old-data” and the record written by the aborted transaction as “new data.” The run time system simply remembers these values. Then, when it discovers the end of the chain, it finds a third record which we term “current data.” The run time system now calls a specific UNDO function:

UNDO (old-data, new-data, current-data)

which returns a modified current record. The run time system writes this revised record with the same transaction and command identifier as the aborted record. In this case, the UNDO function would return $125 and the account state would be changed to:

initial value:	$175
next value:	$75 written by T1 which aborted
next-value:	$25 written by T2 which is in progress
next-value:	$125 written on behalf of T1 as a correction

Hence, later in the chain of records there will be a specific “undo” record. Of course, if the run time system sees the undo record, it knows not to reapply the UNDO function.

It is possible to generalize this technique to support several classes of C1 transactions each with their own UNDO function. However, we view the result as not worth the effort that would be entailed.

3. INTRA-QUERY PARALLELISM

3.1. Introduction

Intra-query parallelism is desirable for two reasons. First, less lock conflicts are generated if a query finishes quickly, and thereby increased throughput can be achieved. For example, a recent study [BHID88] has shown that substantial gains in throughput are possible using parallel plans if a command accesses more than about 10 pages. Second, one can achieve dramatically reduced response time for individual commands. This section presents a sketch of the optimization algorithm planned for XPRS.

In supporting parallel query plans, it is essential to allocate a single relation to multiple files. We choose to do this by utilizing a **distribution criteria,** e.g:

EMP where age < 20	TO file 1
EMP where age > 40	TO file 2
EMP where age >= 20 and age <= 40	TO file 3

to partition a relation into **fragments.** Such distribution criteria will be arbitrary one-relation predicates and can include used defined functions such as hash functions.

When creating indexes on the EMP relation, say on the salary field, it is equally natural to construct three physical indexes, one for each fragment. This will ensure that parallel plans do not often collide for access to the same disk arm or collection of arms. These indexes would have the form:

index on EMP(salary) where age < 20
index on EMP(salary) where age > 40
index on EMP(salary) where age >= 20 and age <= 40

Such data structures are **partial indexes** and offer a collection of desirable features as discussed in [STON88]. Notice that such indexes are smaller than a current conventional index and should be contrasted with other proposals (e.g join indexes [VALD87], links [ASTR76] and the indexes in IMS [DATE84]) which are larger than a conventional index.

Three issues must be addressed by a parallel optimizer in the XPRS environment. First, we assume that that main memory in a 1991 computer system will approach or exceed 1 gigabyte. Obviously with 900 megabytes or more of buffer pool space, a DBMS will keep large portions of data base objects in main memory. For example, one can join two 450 megabyte objects by reading both into main memory and then performing a main-memory sort-merge.

On the other hand, in a multiuser environment much less buffer space may actually be available. Unfortunately, different query plans for the same query are optimal depending on how much buffer space is available. Consider for example the following query:

retrieve (R1.all) where R1.a = R2.b and R2.c = R3.d

and assume that R2 and R3 have a clustered index on fields c and d respectively. Further assume that all relations occupy the same number of pages, P, and that the join of R2 to R3 yields a temporary of size .1P while the one between R1 and R2 contains 1 record. Lastly, suppose the optimizer cost function includes only I/O.

In this case one can first join R1 to R2 and then join the result to R3. Alternately, one can first join R2 to R3 and then join the result to R1. If there is a large amount of memory, the first option will read R1 and R2 into main memory, compute the one record join and then read the single page from R3 which is required. On the other hand, the second option will read all three relations in their entirety, resulting in a higher cost. If only minimal main memory is available, the result is somewhat different. The first option will perform a disk-based merge-sort join of R1 and R2, at a cost of 2P * log P + 2P. The resulting one record temporary will reside in main memory where a single extra page fetch will obtain the matching values from R3. The second option will read R1 and R2 through the clustered index at cost 2P producing a

temporary which is written to disk at cost .1P. A disk based merge sort must be done between this temporary and R1 at a cost of .1P * log .1P + .1P + P log P + P. The formulas for the two cases are presented below and generally the second option will be cheaper for minimum memory.

query plan	cost with zero memory	cost with large memory
(R1 join R2) join R3	2P*Log P + 2P + 1	2P + 1
R1 join (R2 join R3)	3.2P + P*log P + .1P*log .1P	3P

In this case the first plan is superior for a large buffer pool while the second wins if little or no space is available. Consequently, even without considering possible parallelism, the XPRS query optimizer should carefully consider available main memory in its decision making.

When parallelism is considered, the optimizer must also face the number of parallel plans into which to decompose a user query. For example, suppose a user requests:

retrieve (R1.all) where R1.a = R2.b and R2.c = R3.d and R3.e = R4.f

and suppose the best sequential plan is found to be

a) join R1 to R2
b) join R3 to R4
c) join the results of a) and b)

In this case, there may (or may not be) sufficient memory to perform steps a) and b) in parallel. Consequently, the XPRS optimizer must be cognizant of memory allocation when deciding the amount of parallelism to exploit.

Lastly, the XPRS optimizer must be able to decide between two different plans, one of which has greater parallelism while the other consumes less resources.

We feel that optimizers are becoming exceedingly complex, and we have pointed out that main memory and parallelism considerations will necessarily result in additional complexity. Although others [GRAE87, LOHM88] are trying to make optimizers extendible, we are more concerned with making them simpler so that additional optimization, such as that discussed above, can be performed. To accomplish this goal, we plan to restrict the collection of available join tactics. All optimizers must include iterative substitution to process non-equijoins. Moreover, some results must be sorted (those specified by an SQL ORDER BY clause), and therefore merge-sort comes with marginal extra complexity. However, other join tactics (e.g. hash-joins) are inessential, and we plan to not include them in XPRS.

In theory the optimizer search space includes all possible ways to parallelize all sequential plans for all possible buffer pool sizes. This space is hopeless to search exhaustively, and we plan a two step heuristic. In the first step we expect to find good sequential plans for various memory sizes. Then, in the second step we plan to explore parallel versions of only these plans. The final outcome is a collection of plans and a memory range over which each should be run.

At the time of execution, the query executor will make a call on the buffer manager to determine space availability and based on the reply will choose one of the collection of plans. It will then supervise execution of the resulting parallel algorithm. We turn now to constructing this collection of plans.

3.2. Constructing Parallel Query Plans

Our optimizer first finds a collection of good sequential plans as follows. Using a conventional disk-oriented sequential plan optimizer, we expect to produce a plan P(0, Q) for each query Q which is the best plan under the assumption of zero main memory. In addition we expect to produce a plan P (BIG, Q) which assumes as much main memory, BIG, as needed. This plan will typically read each relation accessed into main memory and do all joins in main memory with as much parallelism as possible. BIG is then calculated as the amount of memory needed by all pairs of parallel join operands and their answers. If

P (0, Q) = P (BIG, Q)

then we will stop constructing plans. If not, we will construct P (BIG/2, Q) and compare it to the previous two plans. If it is the same as either plan or has a cost within (say) 10 percent of either plan, we will assume that the optimized plan at each endpoint is, in fact, optimal for the whole interval. Then, we will subdivide the remaining interval (if any), and repeat the process. The net result is a collection of sequential plans and an interval of memory within which each is optimal.

Next we will explore all possible ways to parallelize this collection of plans. To decide between competing plans, we must use a different cost function from traditional optimizers. Specifically, our initial cost function for a query plan, Q, using X units of buffer space will be:

cost (X , Q) = (RES (X, Q) / MIN_RES (Q)) (1 + W2 (TIME (X, Q) / MIN_TIME (Q)))

where

RES (X, Q) = Ntuples (X, Q) + W1 * EIO (X, Q) ; this is the traditional optimizer cost function
Ntuples (X, Q) = number of tuples examined
EIO (X, Q) = expected number of I/O's to evaluate Q with X amount of buffer space
W1 = the traditional fudge factor relating I/O and CPU utilization
MIN_RES (Q) = the minimum cost plan for Q using the traditional function
TIME (X, Q) = expected elapsed time with X amount of buffer space
assuming all processors can be allocated to this plan
MIN_TIME (Q) = the time of the fastest plan for the query Q.
W2 = fudge factor relating response time to resource consumption

If W2 = 0, this new cost function reduces to one equivalent to the traditional one. On the other hand, choosing a large value of W2 will make response time a major criteria.

The XPRS optimizer will iterate over most of the ways to parallelize each plan. However, many leaf nodes of any candidate query plan consists of a scan of a data relation or a scan of a secondary index. Each such leaf node will be automatically decomposed into parallel subplans, one for each data fragment or partial index involved. This tactic can always be applied because it will lower TIME (X, Q) without altering main memory requirements. Moreover, if any scan is followed by a sort node, this node can also be split into the same number of parallel nodes as the scan node.

The ultimate outcome will be a collection of parallel plans with the memory requirements for each one. We expect to proceed with an optimizer on this sort. Clearly, finding good heuristics to prune the search space will be a major challenge that we plan to explore further.

4. PERFORMANCE ON MATERIALIZING LARGE OBJECTS

4.1. Introduction

We expect XPRS to run on a system with a large number of disks. As noted in [PATT88], we believe that only 3 1/2'' and 5 1/4'' drives will be attractive in a couple of years. Hence, we expect large capacity storage systems to made up of substantial numbers (say 100 or more) of such drives. Additionally, these drives do not have removable platters, so the concept of a mounted file system is not required, and we can think of the collection of drives as a two-dimensional array.

In keeping with our objective of using a conventional file system, the problem becomes one of designing a Sprite file system for this disk array which simultaneously yields good performance in transaction processing, complex commands, and materializing large objects. To simplify the discussion, we will assume that a storage system has D drives, numbered 1,...,D, the allocation unit is a disk track and the ith disk has Ti tracks. Hence, the storage system is a two dimensional array of tracks, and we will assume that the horizontal dimension is the drive number and the vertical dimension is the track number on a drive.

In many traditional file systems a user can add a new **extent** to a file which is allocated sequentially on a single drive. If necessary, it would be broken into multiple smaller contiguous extents. In our storage system an extent of size E would correspond to a vertical rectangle of width 1 and height E.

Recently, researchers have suggested striping files across a collection of disks [SALE86, LIVN85]. Striping L <= D disks entails allocating the Ith track to the Jth disk determined by:

J = remainder (I/L) + 1

In this way, a large sequential I/O can be processed by reading all disks in parallel, and very high bandwidth on sequential I/O is possible. In our model this corresponds to a rectangle of width L and height of 1 or more.

In the next subsections we argue that both horizontal (striped over all D drives) and vertical (i.e. traditional) allocation schemes are undesirable in our environment and that a **two-dimensional** file system which allocates extents as general M by W rectangles is the best alternative. Then, we close this section with a few comments on the design of **FTD** (Files -- Two Dimensions).

4.2. Horizontal Allocation

Define the **width** W of a rectangle of storage as the number of drives it is striped over. The choice:

W = D

will result in tricky problems in the area of high availability and space management.

In all real environments the number of disks changes over time. Hence, infrequently, the value of D will change, usually to a bigger value. When a disk is added or dropped, one must restripe all remaining disks. This is a bulk reorganization that will result in the file that is restriped being unavailable during the reorganization. Any incremental restriping algorithm must result in two different widths (namely old D and new D) being supported during the reorganization.

In addition, if the various disks have different capacities, then space management will be problematic because there will be no way to use extra space on larger drives.

Lastly, it is unlikely that hot spots will develop in a striped file system. However, in the unlikely event that they do occur, there is absolutely nothing that can be done about them, because the allocation algorithm is fixed.

4.3. Vertical Allocation

One might be led to consider vertical allocations, i.e. W = 1. Unfortunately this solution fails to achieve our performance goal on large objects.

A large object will occur in a single tuple of a single data base object. For example, the following relation might store an image library:

IMAGE (name, description)

Here, the description field would be several megabytes in size. In a system with W = 1, a description would be stored on a single drive, and therefore the bandwidth available to return it to an application program would be limited to the sequential read speed of the drive (about 1.5 mbyes/sec depending on the drive selected). Clearly, XPRS would fail to achieve its performance goal on these applications with vertical allocation.

4.4. The Design of FTD

The clear conclusion is that neither horizontal nor vertical allocation is a desirable solution, and the file system, FTD, of XPRS must be able to support extents which are arbitrary rectangles. Consequently each extent, Ei, of a file is a data structure:

DRi: the drive number on which the extent starts
Wi: the width of the extent in disks
Si: the size of the extent in tracks
{TRj: 1 <= j <= Wi} the track number on the jth disk on which the extent starts

Hence, each extent is allocated to a contiguous collection of disks and contains Si tracks on each disk.

However, the starting location of the portion of the extent can be different for each disk, thus easing the space allocation problem. In addition, addressing in an extent is striped. Hence, track 1 is allocated to drive DRi, track 2 to DRi+1, etc.

In the remainder of this section we discuss the choice of Wi. There are at least two considerations that would cause one to increase Wi and at least two that would cause it to be lowered. First, bandwidth on large read operations will be proportional to Wi. Hence, in supercomputer access to a DBMS, one should choose a large Wi. In addition, a larger choice of Wi will tend to minimize the impact of ''hot spots'' in the disk system, i.e. drives on which there is contention for blocks from multiple transactions. If a single file is spread over a larger number of drives, one would expect contention for blocks in that file to decrease. However, there can also be contention for blocks in different files on the same drive. Reducing such contention is a file placement problem. The amount of such inter-file contention will decrease as the Wi for both files is increased. As a result, concern for hot spots would cause one to increase Wi.

On the other hand, there are two considerations which would cause one to choose lower values for Wi. First, space management will probably be easier with lower values. This will clearly be true if some disks have different capacities from others. We expect to demonstrate this conjecture with a simulation study which has already started.

Second, there are many environments where the extra bandwidth from a large Wi cannot be utilized by the DBMS. As noted earlier, parallelism will be obtained by splitting data relations into multiple files and allocating parallel query plans to process each file. Suppose the CPU processing one of these parallel plans is (say) 15 MIPS. Furthermore, suppose we assume typical records of (say) 100 bytes and a query which requires (say) 500 CPU instructions per record. In this case, we require 5 instructions per byte of data and a 15 MIPS CPU can keep up with at most 2 disks. If the CPU cost per record is cut in half, then four disks can be supported. Hence, the benefit of striping more than a few disks will be lost because of CPU saturation.

In conclusion we expect to design a file system where files can be extended an extent at a time and application software can optionally suggest the value of W that would be appropriate for the extent. Larger values of W may result in higher bandwidth and less problems with hot spots. On the other hand, lower values may result in equal effective bandwidth and less problems with space management.

5. HIGH AVAILABILITY IN THE PRESENCE OF ERRORS

5.1. RAID

The I/O system in XPRS will be based on RAIDs (Redundant Arrays of Inexpensive Disks) [PATT88]. The underlying premise is that small numbers of large expensive disks can be replaced by very large numbers of inexpensive disks to achieve substantially increased transfer bandwidth at a comparable system cost. The major problem with disk arrays is the drastically reduced mean time to failure (MTTF) because of the large numbers of additional system components.

RAIDs are only of interest if they can be made fault tolerant. At one extreme, each data disk can have an associated “mirror” disk, which is comparable to Tandem’s mirrored disk approach. However, 50% of the available disk capacity is dedicated to redundant data storage, a rather high price to pay.

We take an alternative approach and assume that each FTD “logical drive” is, in fact, made up of a **group** of N physical disks. On N-1 of these disks normal data blocks are stored, while on the Nth disk, we store the parity bit for the remaining drives. Blocks on different drives can be read independently; however, writes require (up to) four physical I/Os:

(1) read original data block
(2) read its associated parity block
(3) write the updated data block
(4) write the updated parity block

Intelligent buffer management and/or read-modify-write transactions can eliminate one or two of these I/Os in many cases.

To avoid the hot spot on the Nth drive during write operations, parity blocks are actually interleaved across all N disks. Consequently, up to N/2 writes can be serviced simultaneously.

Note that the parity blocks represent much reduced overhead compared to the fully mirrored approach. For N = 8, one in every eight blocks is a parity block. This represents only a 12.5% capacity overhead.

When the controller discovers that a disk has a hard failure, processing continues in a degraded fashion as follows. A **hot spare** is allocated to the group, replacing the failed disk. A read to the failed disk is mapped into parallel reads of the data and parity blocks of the remaining disks, and the lost data is reconstructed on the fly. Writes are processed as above, and are written through to the spare.

Just as in the case of fully mirrored disks, a second failure renders the group unavailable. Thus it is also important to reconstruct the contents of the failed disk onto the spare drive expeditiously. Two strategies are possible: stop and reconstruct, or reconstruct in the background. In the former, access to the group is suspended while the reconstruction software runs flat out to rebuild the lost disk. Sequential access can be used to advantage to keep the reconstruction time to a minimum, but assuming a group of 8 100 Mbyte 3 1/4'' disks, this is a computationally intensive task which will take at least

100 mbytes / 1.5 mbytes per second = 67 seconds

assuming that the I/O processor doing the reconstruction can keep up. This approach does not satisfy the high availability goals of XPRS.

The alternative is to spread the reconstruction over a longer period, interleaving reconstruction and conventional I/O. We assume the actual elapsed time to reconstruct the disk thereby increases by a factor of 400 to four hours.

The drawback of this approach is that a longer recovery period will adversely affect the MTTF because a second physical failure will cause data loss during the longer reconstruction period. To be specific, assume the average time to a physical disk failure is 30000 hours, and therefore the failure rate, λ, is 1/30000. Assume that the average repair time is 4 hours, and therefore the repair rate, μ, is 1/4. The mean time to failure of a group of N disks is:

$$MTTF = \frac{\mu}{N(N-1)\,\lambda^2}$$

Thus, MTTF decreases linearly with increasing repair time (decreasing repair rate). For N = 8 and a 4 hour repair interval, the MTTF exceeds 3.5 million hours. Put differently, one can have a disk array of 20 of these groups containing 160 drives, and be assured that the MTTF of the entire system is 175,781 hours, a little over 20 years.

In XPRS we will consequently assume that the disk system is perfectly reliable.

5.2. Software Errors

The POSTGRES storage manager is discussed in [STON87] and has the novel characteristic that it has no log in the conventional sense. Instead of overwriting a data record, it simply adds a new one and relies on an asynchronous vacuum cleaner to move "dead" records to an archive and reclaim space. The POSTGRES log therefore consists of two bits of data per transaction giving its status as

committed
aborted
in progress
C1-in-progress

To commit a transaction in POSTGRES one must:

move data blocks written by the transaction to ''stable'' memory
set the commit bit

To abort a transaction one need only set the abort bit. To recover from a crash where the disk is intact, one need only abort all transactions alive at the time of the failure, an instantaneous operation. Since RAID has

an infinite MTTF for disk errors, there are no crashes which leave disk data unreadable.

To achieve higher reliability one must be able to recover from software errors caused by the DBMS or the OS writing corrupted disk blocks. In this section we sketch our design which has the side benefit of making the buffer pool into "stable" storage. This will make committing POSTGRES transactions extremely fast. We base our design on two assumptions:

Assumption 1: The OS ensures that each main memory page is either GUARDED or FREE. Any guarded page is assumed to be physically unwritable and its contents obtainable after any crash.

We expect to implement GUARDED and FREE by setting the bit in the memory map that controls page writability. With a battery back-up scheme for main memory and the assumption that memory hardware is highly reliable, Assumption 1 seems plausible.

Assumption 2: The DBMS and the OS consider the operation of GUARDING a page as equivalent to "I am well." Hence, issuing a GUARD command is equivalent to the assertion by the appropriate software that it has not written bad data.

Although there is no way to ascertain the validity of Assumption 2, we expect to attempt to code routines near GUARD points as "fail fast."

Our buffering scheme makes use of the fact that the OS has one copy of each block read and the DBMS has a second in its buffer pool. Moreover there are 6 system calls available to the DBMS:

```
G-READ (X,A)      :Read disk block X into main memory page A leaving A GUARDED
READ (X,A)        :Read disk block X into main memory page A leaving A FREE
G-WRITE (A,X) :Write main memory page A to disk block X leaving A GUARDED
WRITE (A,X)       :Write main memory page A to disk block X leaving A FREE
GUARD (A)         :GUARD main memory page A
FREE (A)          :FREE main memory page A
```

The OS implements a G-READ command by allocating a buffer page, B, in its buffer pool and performing the following operations:

```
FREE (B)
physical read of X into B
GUARD (B)
FREE (A)
copy B into A
GUARD (A)
```

The READ command is nearly the same, omitting only the last GUARD (A). The OS implements G-WRITE (X,A) by using its version of the page, B, as follows:

```
GUARD (A)
FREE (B)
copy A into B
GUARD (B)
```

The WRITE command is the same except it adds a FREE (A) at the end. The OS can write pages from its buffer pool to disk at any time to achieve its space management objectives.

Each time the DBMS modifies a data page, it must perform a WRITE or a G-WRITE command to move the OS copy into synchronization. Moreover, it must assert that it has not written invalid data. According to Assumption 2, it would perform a GUARD command preceding the WRITE or G-WRITE command. For efficiency purposes, we have combined the two calls together; therefore a WRITE or G-WRITE command is equivalent to a "wellness" assertion by the DBMS.

If a crash occurs, then the OS takes the initiative to discard all unguarded pages in its buffer pool as well as in the DBMS buffer pool. All other buffer pool pages are preserved. Moreover, the code segments

of the OS and DBMS are automatically guarded, so they are intact.

Lastly, it should be noted that both the DBMS and OS copies of a page are never simultaneously unguarded. Hence, if the DBMS page is discarded, it will be refreshed from the OS page. If the OS page is discarded, it will be rewritten from the DBMS page. Moreover, since the number of unguarded pages at any one time is small, the two copies can be brought into synchronization quickly during recovery time.

It is acceptable for multiple transactions to have the same page simultaneously FREE. In this case, a GUARD operation by one transaction requires the OS to perform the obvious bookkeeping leaving the page FREE. Only when the last transaction GUARDS the page can the actual page be guarded. Of course, a transaction must delay committing until all the pages it has written have become physically guarded. It will be useful to periodically delay transactions which wish to FREE a ''hot spot'' page so that the current writers of the page can finish and the page can be GUARDED. This is similar in concept to action consistent checkpoints discussed in [GRAY81].

There are additional details that concern how to preserve the data structure which holds the mapping of disk pages to buffer pages. However, space precludes an explanation here. Also, assuming that the I/O system does not write blocks to the wrong place along with Assumptions 1 and 2 above, our scheme does not lose data and recovers essentially instantly.

6. AVOIDING DATA UNAVAILABILITY DUE TO LOCKING

In this section we indicate the approach taken by XPRS to avoid data unavailability on large user reads and on storage reorganizations.

6.1. User Reads

POSTGRES automatically supports access to a relation as of some time in the past. For example, a user can obtain the names of employees as of January 15th as follows:

```
retrieve (EMP.name)
using EMP@''January 15, 1988''
```

All retrieve commands can be run as of some time in the past. Because no locks are set for such commands, they cause no data unavailability. In addition, our technique does not require a user to predeclare his transaction to be read-only as required by some other techniques, e.g [CHAN82].

6.2. Storage Reorganization Without Locking

We now illustrate how partial indexes can be used by an automatic demon to achieve incremental index reorganization.

To convert from a B-tree index on a key to either a rebuilt B-tree index or a hash index on the same key, one can proceed as follows. Divide the key range of the index into N intervals of either fixed or varying size. Begin with the first interval. Lock the interval and construct a new index entry for each tuple in the interval. When the process is complete, unlock the interval. The new index is now valid for the interval

```
key < VALUE-1
```

where VALUE-1 is the low key on the next index page to be examined. The old index can be considered valid for the whole key range or it can be restricted to:

```
key >= VALUE-1
```

In this latter case, the space occupied by the index records of the first interval can be reclaimed. If the intervals are chosen to be the key ranges present in the root level of the old B-tree, then this space reclamation can occur without destroying the B-tree property for the old index.

The query optimizer need only be extended to realize that the two indexes together cover the key range. Hence, if a query must be processed with a qualification of the form:

```
where VALUE-3 < key < VALUE-4
```

it is necessary to construct two query plans, one for each index. There is little complexity to this optimizer extension. At one's leisure, the remaining N-1 intervals can be processed to generate the complete index.

All storage reorganizations to achieve alternate access paths or arm balance can be similarly coded as incremental operations using distribution criteria and partial indexes. We expect to embed these techniques into a collection of asynchronous demons that will run in background, thereby relieving the operator of manual (and error prone) operations.

It would also be possible to build an index without setting any locks and then process the log to correct the index afterwards. This would be similar to incremental techniques for dumping relations, so called ''fuzzy dumps''. Our technique is superior because portions of the index can be utilized as soon as they have been constructed. There is no need to wait for the end of the entire build procedure.

7. CONCLUSIONS

We have described the design of a hardware and software system to support high performance applications. This entails modifying POSTGRES to support fast-path and partial indexes, writing a collection of demons to provide housekeeping services without the presence of a human, building a controller for RAID, and providing parallel query plans.

The hardware platform utilized will either be a large Sun machine or a SEQUENT Symmetry system. The construction of RAID is in progress and we expect an initial prototype by late 1988. The fast-path feature of POSTGRES is nearly operational and we are tuning up the system to achieve our TPS performance goal. During 1989 we will concentrate on partial indexes and parallel plans.

REFERENCES

[ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.

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

[BADR87] Badrinath, B. and Ramamritham, K., ''Semantics-Based Concurrency Control: Beyond Commutativity,'' Proc. 1987 Data Engineering Conference, Los Angeles, CA, February 1987.

[BAMB87] Bamberger, F., "Citicorp's New High Performance Transaction Processing System," Proc. 2nd International Workshop on High Performance Transaction Systems, Asilomar, CA, Sept. 1987.

[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.

[CARE86] Carey, M. et. al., "The Architecture of the EXODUS Extensible DBMS," Proc. International Workshop on Object-oriented Data Bases, Pacific Grove, CA, Sept. 1986.

[CHAN82] Chan, A. et. al., "The Implementation of an Integrated Concurrency Control and Recovery Scheme," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fl., June 1982.

[DATE84] Date, C., "An Introduction to Database Systems, 3rd Edition" Addison-Wesley, Reading, Mass., 1984.

[DEWI85] Dewitt, D. and Gerber, R., "Multiprocessor Hash-based Join Algorithms," Proc. 1985 VLDB Conference, Stockholm, Sweden, Sept. 1985.

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

[GRAE87] Graefe, G. and Dewitt, D., “The EXODUS Optimizer Generator,” Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, CA, May 1987.

[GRAY81] Gray, J. et. al., ‘‘The Recovery Manager of the System R Database Manager,’’ Computing Surveys, June 1981.

[GRAY87] Gray, J., (private communication).

[GRAY87A] 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.

[LOHM87] Lohman, G., “Grammar-like Functional Rules for Representing Query Optimization Alternatives,” IBM Research, San Jose, CA, RJ5992, Dec. 1987.

[OUST87] Ousterhout, J. et. al., “The Sprite Network Operating System,” Computer Science Division, University of California, Berkeley, CA, Report UCB/CSD 87/359, June 1987.

[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.

[ROWE87] Rowe, L. and Stonebraker, M., “The POSTGRES Data Model,” Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[RTI87] Relational Technology, “INGRES/STAR Reference Manual, Version 5.0” Relational Technology, Inc., Alameda, CA, June 1986.

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

[SELL86] Sellis, T., “Global Query Optimization,” Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 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.

[STON86] Stonebraker, M. and Rowe, L., “The Design of POSTGRES,” Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86A] Stonebraker, M., “The Case for Shared Nothing,” IEEE Database Engineering, March 1986.

[STON87] Stonebraker, M., “The POSTGRES Storage System,” Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON88] Stonebraker, M., “The Case for Partial Indexes,” Electronics Research Laboratory, University of California, Berkeley, CA, Report ERL M88/62, June. 1988.

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

[VALD87] Valduriez, P., “Join Indices,” ACM-TODS, June 1987.

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