THE DESIGN OF XPRS

by

Michael Stonebraker, Randy Katz,
David Patterson, and John Ousterhout

Memorandum No. UCB/ERL M88/19

10 March 1988

COVER PAGE

THE DESIGN OF XPRS

by

Michael Stonebraker, Randy Katz
David Patterson, and John Ousterhout

Memorandum No. UCB/ERL M88/19

10 March 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

THE DESIGN OF XPRS

by

Michael Stonebraker, Randy Katz
David Patterson, and John Ousterhout

Memorandum No. UCB/ERL M88/19

10 March 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

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 (DoD) 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 prove that high performance in each of these areas can be provided by a next generation DBMS running on a general purpose operating system without unduely 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, and our performance goal for XPRS in transaction processing (measured in TP1 [ANON85] transaction per second) is:

XACTS/sec = 5 * MIPS

For example, a 200 MIPS system should be capable of 1000 TP1s per second. Although this corresponds to "common" performance in [ANON85], we believe that an "academic strength" prototype cannot realistically hope to do any better. Also, considerable commercial attention has been focused recently on the TP1 benchmark in Sun and VAX environments. The current performance leader in the Sun environment is INGRES [RTI88] which does about 14 TP1s per second on a Sun 3/260 and 30 TP1s per second on a Sun 4, i.e. about 4 * MIPS. Hence, highly tuned high-function commercial relational systems are somewhat short of the goal that we have set.

We expect to achieve our performance goal by a novel use of **fast path** to achieve a **variable speed interface.** We discuss this tactic later in this paper.

To achieve high performance on benchmarks like TP1, it is necessary to have more than just raw DBMS power. Most transaction processing applications have so-called **hot spots,** i.e. records or blocks with high traffic. Lock contention for hot spot objects is a limiting factor in many environments. To alleviate this 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 many situations. These tactics are also described 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]. Unfortunately, these papers indicate how to solve two-way joins in a single-user environment. On the other hand, in this paper we present a general multi-user n-variable algorithm appropriate for a high speed uniprocessor or a shared-memory multi-processor with 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). 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 software errors are more common than environment ones [GRAY87]. It should be noted that current techniques 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 will write the same bad data that caused its mate to fail. Hence, our goal in XPRS is to recognize that software errors will happen and to limit their ability to do damage. Moreover, after a software error we consider it unacceptable for software recovery times to be measured in minutes. Hence, a goal of XPRS is to recover from a crash 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.

Lastly, the most common cause of environment errors are power failures, and we view an uninterruptable power supply as a viable solution to this problem, which coincidentally provides stable main memory. Hence we have no specific additional goal in this area.

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 [GRAY87], 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. Hence we reject the often argued premise that shared memory system lack required I/O bandwidth [DEWI86].

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 most 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 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 query processing algorithm 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

2.1. Introduction

A high-function data base system has an inherent problem when asked to perform transaction processing. High function usually entails substantial overhead per command. Intuitively, a high-function system provides a "sledgehammer" rather than a pair of nutcrackers. However, the overhead of swinging the sledgehammer substantially exceeds the cost of lifting the nutcracker. In a transaction processing environment, this overhead is a dominant source of CPU overhead. To simultaneously provide high function and low overhead we explore two tactics that we are building into POSTGRES.

2.2. Fast Path

It is common knowledge that TP1 consists of 5 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 7 times. This overhead contributes 20-30 percent of all CPU cycles consumed by a TP1 transaction. To alleviate this difficulty, several current commercial systems (including INGRES, the IDM 500, and Sybase) 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 one of four ways. The easiest but slowest way is for the function to simply execute the stored procedure TP1 noted above. Of course, this provides no performance advantage relative to the application doing the same thing. The second option is for the function to have the query plans for all TP1 commands as a data structure inside the function. It can substitute the parameters into the plan and then pass each plan to the POSTGRES executor. The only requirement for this approach to work is that the parser and optimizer be functions that the designer of the TP1 function can use, e.g:

parse (string)
optimize (parse-tree)

Compiling a plan in advance of execution has been used by conventional preprocessors for many years. Again, little if any advantage is gained relative to executing a stored procedure because stored procedures are also compiled in advance of execution.

The third, higher performance implementation takes advantage of the fact that POSTGRES has user defined access methods. Hence, the abstraction for the access methods is 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 result in excellent performance.

The fastest implementation of TP1 directly calls the routines which manage the buffer pool. If the user is willing to give up essentially **all** data base services, he can obtain ultimate performance.

In this way, user transactions can be coded against several different interfaces of POSTGRES. Ones with critical performance objectives can be coded against the bottom level while less critical ones can take full advantage of POSTGRES services at higher levels. Using this technique POSTGRES provides 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 provide a variable speed interface by requiring a data base implementor to build a custom system out of tool-kit modules.

We plan to use fast path on TP1 only to the extent necessary to achieve our performance goals. Hence, if procedures are workable we will stop there; otherwise we will implement TP1 directly against the access methods.

2.3. 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 five 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.

2.4. 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 page. 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.

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:

	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.

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

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 POSTGRES optimization algorithm and the parallelism that it achieves.

3.1. Assumptions

We begin our discussion of query optimization with three fundamental assumptions which will radically differentiate our proposal from others that have been made.

Assumption 1: Main memory is available to hold any two data objects a user wishes to join.

The basic reasoning is 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 can join two 450 megabyte objects. It is our experience that most users restrict data base objects to smaller sizes than these before joins are performed. Put differently, the join of two 450 Megabyte objects is another 450 megabyte object, and few users are going to be interested in looking at that much line printer output.

Assumption 2: It takes about 1000 instructions to process a record.

This include a fraction of the CPU tax to get the page containing the record off the disk, find the page in the buffer pool, extract the desired record and then pull apart the required fields.

Therefore, a stream of accessed records can be sorted in main memory for marginal additional cost. Consider, for example, a 100 megabyte object consisting of 500,000 records each 200 bytes wide. Consider a main memory insertion sort that sorts these records without moving them (i.e. by sorting a list of (key, pointer) pairs). Each record can be inserted in log (500,000) comparisons. These 19 comparisons require perhaps 20 instructions each [DEWI84]. Hence, the incremental overhead to sort a record stream is 380 instructions per record, i.e. 38 percent of the 1000 instruction processing overhead.

Assumption 3: Current Optimizers are too complex

Traditional optimizers (e.g. [SELI79]) have the following bad properties. They are slow, typically requiring more than 200,000 instructions of running time. By itself this is not a severe problem; however current optimizers will be called on to participate in heterogeneous, open architecture distributed data base systems, such as INGRES/STAR [RTI87] and ORACLE/STAR. Obviously, this will require a local optimizer to be the inner loop of a distributed optimizer. If such a distributed optimizer is to have finite running time, it is necessary for the local optimizer to be fast.

Moreover, current optimizers are complex modules of code that are not amenable to large expansions in complexity, such as adding query plan parallelism. Others have suggested rule-driven optimizers as a solution [LOHM87, GRAF87]. We are skeptical of this approach as it will tend to make the performance problem worse and looks like it will make optimizers more complex, albeit easier to change.

There are several startling conclusions that result from Assumptions 1 - 3. First hash join [DEWI86] algorithms are not worthwhile. The reasoning is that they will consume the same number of I/Os as a merge sort algorithm and marginally less CPU instructions (perhaps 10 percent). Since some results must be sorted anyway (e.g. those specified by an SQL ORDER BY clause), it is not worthwhile for an optimizer to consider an alternate, inessential join tactic with marginally better performance. A similar argument suggests that join indexes [VALD87] are not worthwhile either.

Another conclusion is that main memory will be the major resource to worry about. It will clearly be possible to break a query into many parallel plans. However, performing two joins in parallel will consume twice as much main memory as doing the joins sequentially. If the prerequisite memory is available, then parallelism should be exploited; otherwise, it should be avoided. As a result, the desired amount of parallelism will be determined by the amount of available main memory.

In the remainder of this section we indicate our algorithm to plan queries and construct the desired amount of parallelism. Our query planning algorithm proceeds in two steps. First we construct a collection of **plan fragments** using a simple heuristic algorithm and construct a query plan which maximizes parallelism assuming sufficient main memory is available. Then, we schedule the desired number of parallel fragments based on available main memory.

3.2. Infinite Memory Query Planning

We are required to process a query which we assume is represented as a **directed query graph.** Each relation is considered as a node in this graph and each join clause is represented by two arcs, one in each direction between the pairs of nodes. Restriction clauses and statistics are used to generate the following information for each node, Ni, using traditional techniques.

COST(i):	The expected cost of reading qualifying tuples from the ith relation into main memory using the best access path
NUMB(i):	The expected number of qualifying records from the ith relation, i.e. the size of the main memory temporary

For each arc pointing from node i to node j in the graph, we require the following:

TSUB(i,j):	The cost of accessing the jth relation to find qualifying records that match a single tuple in the ith relation. Hence, this is the cost of the inner loop of a tuple substitution join.
SEL(i,j):	The join selectivity of the clause represented by the arc. Any one of the standard definitions can be used. We do not require that SEL(i,j) = SEL(j,i).

Each node and each arc will have a status field, N-STATUS and A-STATUS, with possible values {available, used}. Initially all nodes and arcs are unused. Moreover, each node will have a field CAME-FROM which will be used to detect loops. The initial value of CAME-FROM is the node itself. Last, a list of nodes, LIST, is required. The algorithm uses this data as follows:

```
(1) Choose the available node N(l) with minimum cost and add to LIST.

(2) Delete l from LIST and for each outgoing arc, A(l,m), compare COST(m) to
NUMB(l) * TSUB(l,m). If COST(m) is smaller then do nothing else
    if N-STATUS[l] = available then:

            set CAME-FROM = l
            set N-STATUS[l] = used
            set A-STATUS[l,m] = used
            set COST(m) = COST(l) * TSUB(l,m)
            set NUMB(m) = SEL(l,m) * NUMB(l)
            add m to LIST
    if N-STATUS[l] = used and CAME-FROM != l then:
            set CAME-FROM = l
            set A-STATUS[l,m] = used
            set COST(m) = COST(l) * TSUB(l,m)
            set NUMB(m) = SEL(l,m0 * NUMB(l)
            find the other incoming arc with STATUS of used and change it to available
            add m to LIST
    if N-STATUS[l] = used and CAME-FROM = l then:
            do nothing
(3) Repeat step 2 on each member of LIST. Continue until LIST is empty

(4) Repeat steps (1) - (3) until there are no available nodes.
```

At the end of this process, partition all nodes into groups with common values of CAME-FROM. Add to each group the connecting arcs with an N-STATUS of used. The resulting data structure for each value of CAME-FROM, m, is guaranteed to be a tree, and will constitute a **plan fragment,** P(m). In realistic queries, we expect between one and four such plan fragments, and each plan fragment will be executed in parallel. A single plan, P(m), will be executed by reading the relation corresponding to node m into main memory using the best access path previously determined. For each outgoing arc, A (m,j) tuple substitution will be used to fetch records from relation j into main memory. This process continues until the plan fragment is exhausted. The last step will be to do a main memory insertion sort on one of the fields in the resulting temporary as presently discussed.

The optimizer should break P(m) into one physical plan per physical index for the relation being accessed. As will be seen in Section 6.2, several such indexes may be used to access a single relation. If there are multiple nodes in the plan fragment, then there should be an additional plan for each arc. This has been called pipelining in [DEWI86]. For each plan fragment, the required main memory MEM(m) is calculated by the optimizer in a straight forward way along with the size S(m) of the resulting temporary.

Now collapse the pairs of arcs in the query graph that correspond to the same join clause into a single bi-directional arc and delete any arcs with STATUS of used. The remaining arcs must be used to join plan fragments together. Pick a random plan fragment and join it to the smallest plan that it is connected to by performing an insertion sort of each temporary in main memory on the field in the join clause with smallest selectivity. The merge process can be done in a separate plan. Continue choosing plans and performing a parallel merge-sort until the plan fragments are exhausted.

The result is a collection of main memory temporary relations, one for each pair of plan fragments. Now repeat the above step on pairs of temporaries to connect more of the plans and continue until the final answer is produced.

3.3. Dealing with Finite Memory

The above procedure will yield a query plan which requires approximately twice the sum of MEM(m) over all plan fragments. By assumption there is available main memory for any two plans. Hence, we require a procedure for dealing with the situation where there are three or more plan fragments and not enough memory is available for all of them. In this case one or more partial results will have to written out to disk.

A simplistic algorithm for this situation is to start with a random plan fragment and join it to the smallest plan fragment to which it is connected. Continue joining pairs of plan fragments until they are exhausted or until available memory is exhausted. If memory is exhausted, write the temporaries to disk. Redo the algorithm on the remainder of the query graph. We expect to improve on this simplistic algorithm in the future.

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 unworkable in our environment and that a **two-dimensional** file system which allocates extents as general M by W rectangles is the only viable 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. Moreover, although hot spots are unlikely in this organization, there is nothing that can be done about them if they happen. Lastly, it will be seen that a DBMS has little liklihood of taking advantage of the resulting bandwidth. We now discuss each point in turn.

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. Since the goal of XPRS is to avoid such software-induced unavailability, the choice of W = D is unacceptable.

In our disk system, RAID, a double disk failure is required to make data unavailable as noted in Section 5.1, and the mean time to such an event is measured in tens of years. However, a cautious system administrator might still be concerned about data loss in this situation. If W = D, then each disk will have some data from all files. In environments where large objects are

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.

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

In supporting parallel query plans, it is essential to allocate a single relation to multiple files. We choose to do this 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
```

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. 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 reclaimation 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.

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 operational and we are tuning up the system to achieve our performance goal of 5 * MIPS. During 1988 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.

being stored, this will mean that the loss of a single drive will make all large DBMS objects unavailable until recovery actions can be completed. Although this is an extremely unlikely scenario, it might be a concern for a cautious system administrator.

Moreover, 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, although the DBMS can take advantage of high bandwidth when accessing large objects, this will generally be the exceptional case. Consider a normal environment in which a query is executed which sequentially reads a single relation. Such queries are generated by users, for example producing a large batch report, or by the data base sort program, which might be executing a merge-sort to perform a join with no indexes present.

A striped file system can, in theory have each drive read data a track at a time sequentially. In main memory, the resulting tracks can be correctly interleaved. Consequently, the bandwidth that is achievable on a sequential query is about:

D * 1.5 mbtes/sec

However, much less than this maximum bandwidth can be effectively used by a DBMS. Since we have assumed 1000 CPU instructions are required to process a typical record, of (say) 100 bytes, 10 CPU instructions per byte of data are required. Therefore a 15 MIPS CPU can keep up with only one disk. Hence, the benefit of striping more than a few disks will be lost on sequential commands because of CPU saturation.

To achieve better performance on sequential transactions, one must resort to breaking a sequential query into multiple query plans as noted in Section 3. The easiest way to manage the bookkeeping and synchronization of multiple plans is to allocate individual relations to multiple files and construct one query plan per file. This technique is discussed in Section 6.2.

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 workable 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
TRi: the track number of which the extent starts
Si: the size of the extent in tracks
Wi: the width of the extent in disks

Hence, each extent is a physically contiguous collection of tracks beginning at TRi and continuing to to TRi + Si -1 on each of disks DRi through DRi + Wi -1. Moreover, addressing in the rectangle 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 for particular applications and suggest that there are several dramatically different cases to consider. Consider first large object retrieval where high availability is not a consideration. This might be the case, for example, in supercomputer access to a DBMS. Clearly, in this situation one should usually choose W = D and provide maximum bandwidth to the application. On the other hand if availability is an issue, then the system administrator must know the required bandwidth of the applications using these objects. He should then set Wi to be the minimum value which meets this bandwidth requirement. Any choice of a larger Wi will result in lower availability if a catastrophic drive failure occurs or restriping must be done when the number of drives changes.

On the other hand, in a transaction processing environment where only small commands are run, one should choose W as large as possible to minimize problems with hot spots. The only reason not to choose W = D is because space management will be difficult and drive unavailability will result when disks are added and dropped.

Lastly, in an ad-hoc query environment where query parallelism is desired, the DBMS should partition each relation into a collection of files. There is no advantage to spreading such files over more than a few disks due to plan saturation. Hence, small values of W will work just fine.

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.

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 five minutes. This strategy does not meet the high availability requirements 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 fourty-eight 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.

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

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

[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.

[RTI88] Relational Technology, Inc., "INGRES Reference Manual, Version 6.0," Relational Technology, Alameda, Ca., February 1988.

[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, Feb. 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 M88/76, March 1988.