1. Introduction

As database management systems evolved, users have made ever-increasing demands on the number, type and speed of the services provided by the DBMS implementations. Relational database systems have provided many users with the primitives they need to interact productively with their data, and recent implementations have provided them with acceptable performance. New research projects — among them, POSTGRES [STON86c] and XPRS [STON88] — have begun to address the next round of function and speed demands. This thesis describes some of the query processing techniques used in the POSTGRES database management system to meet these demands.

The organization of the thesis is as follows. Section 1 presents an overview of POSTGRES, XPRS, and related research efforts. Section 2 describes how the POSTGRES query optimizer supports several features that provide function beyond that provided by standard relation database management systems. Section 3 is a design sketch for a set of extensions to the POSTGRES query processing system that will enable POSTGRES to exploit features of the XPRS database machine in order to increase performance, as well as some experimental results that support the assumptions made in the design. Finally, Section 4 gives some directions for future work.

1.1. Overview of POSTGRES and XPRS

POSTGRES (POST inGRES) [STON86c, WENS88] is a database management system under development at the University of California at Berkeley. The original INGRES project [STON76] helped to prove that relational database management sys-

tems are practical and useful. However, existing RDBMS implementations and semantics have proven deficient in certain applications. As the follow-on to INGRES, the POSTGRES project has had the dual goals of providing additional *function* that the relational model alone lacks, as well as *speed* beyond that provided by current RDBMS implementations. Among the advanced features provided by POSTGRES are a rules system capable of supporting data integrity enforcement and artificial intelligence applications, query-language procedures for modelling complex data, and a powerful and extensible data model (including a data structure and method inheritance mechanism). The POSTGRES design includes interfaces for programmers to access system internals at different levels, enabling those who wish to sacrifice some safety and abstraction for raw speed to do so.

XPRS (eXtended Postgres, Raid and Sprite) [STON88] is a database machine in construction at Berkeley. The goal of the XPRS project is to build a high-performance transaction-processing engine with as few full-custom parts as possible; that is, the general philosophy is that as much as possible should be done in software and relatively inexpensive off-the-shelf hardware. Cost-effective transaction-processing performance requires a balance of I/O and computational power. XPRS will achieve its I/O performance goals by exploiting the parallelism available through use of redundant arrays of inexpensive disks (RAIDs) [PATT88] and a new log-based file system [DOUG89] built into the Sprite network operating system [OUST88]. The CPU power will be provided by a commercially-available shared-memory multiprocessor computer and a version of POSTGRES modified and tuned for this kind of tightly-coupled parallel architecture.

Since the current POSTGRES prototype has been constructed solely as a uniprocessor system, a great deal of the high-level query processing code will have to be restructured to deal with the control of multiple processors. In particular, the *query optimizer* and *query execution* modules must be modified to take parallelism into account.

1.2. Query Optimization

A database management system might require its users to query the data in a manner which they know to be efficient. For example, if a file has an indexing structure built over it which allows the file to be probed efficiently, the user could be forced to refer explicitly to the indexing structure in order to achieve good performance. Data models which allow queries to be expressed in a manner independent from the physical data layout, such as the relational model, do not have this ‘‘luxury.’’ Such systems must have a *query optimization* module, which, given some representation of the user’s query (which refers only to entities described by the data model), returns some representation of an efficient scheme for accessing the physical data that underlies the data model. In short, in order to save the user from having to understand the physical data layout and provide efficiency hints, the system itself must be able to figure out the equivalent information.

A general introduction to query optimization techniques in relational database systems can be found in [JARK84]. Briefly, a typical approach is for the query language parser to generate some kind of binary *query tree* structure that represents the user’s query and pass this to the query optimizer, which generates another tree of

query processing operators (scans of individual relations and joins between two relations). This *query plan tree* is then either interpreted at runtime or compiled into machine-language instructions that can be directly executed. The selected plan tree is chosen by applying some kind of cost function to each tree in the search space of possible trees. The cost functions typically take into account an estimate of the number of disk I/Os and processor cycles required to evaluate the query using the given plan tree.

Picking out the optimal tree from the entire space of tree topologies, node types, and node labellings is known to be an NP-complete problem. Hence, query optimization is a matter of choosing a set of heuristics that work for the expected workload. Consider the RTI INGRES optimizer [KOOI82] and its predecessor from the Univer-

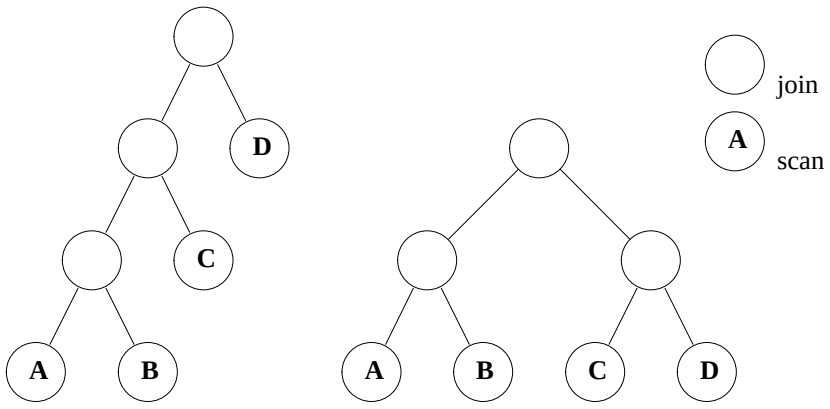

Figure 1. ‘‘Left-deep’’ vs. ‘‘bushy’’ query plan trees.

sity of California version of INGRES [WONG76]. Both INGRES optimizers produce *full* or *bushy* query plan trees; that is, no arbitrary set of tree topologies is ignored. However, because of the intractability of searching the entire topology space, both systems limit the search time. University INGRES uses a greedy search heuristic known as ‘‘decomposition’’ and commercial INGRES uses a (varying) limit on the time spent optimizing the query. As another example, consider the System R optimizer described in [SELI79]. Unlike the INGRES optimizers, the System R optimizer reduces the topology space by considering only *left-deep* query plan trees, that is, trees in which the right subtree is always a leaf node representing a base-relation scan. Limiting the tree topologies in this way reduces the query plan search space (and hence optimizer execution time) from $O(8^N)$ to $O(2^N)$ in the number of relations being joined [GRAE87]. While this strategy seems highly restrictive, experimental evidence from [GRAE87] suggests that planning left-deep trees is nearly as effective as planning bushy trees for a small number of join relations (e.g., 10 or less) but that this heuristic becomes much less effective as the number of relation (and, of course, the optimization search space) increases. Examples of trees with the two topologies may be seen in Figure 1. The leaf (labelled) nodes represent scans of the base relations and the internal (unlabelled) nodes represent joins.

The degree to which other database management systems can take advantage of underlying system parallelism varies a great deal. The simplest kind of system runs multiple copies of the data manager process and uses a central transaction manager to ensure data integrity. This technique allows multiple queries or transactions to run in parallel (*interquery* parallelism) but does not of itself speed up single queries. An

example of this is Tandem's NonStop SQL[1] systems as described in [BORR88]. Another class of machines tries to exploit *intraquery* parallelism by breaking query plans into multiple parts that can be executed concurrently. We will no longer consider the first class of systems, since our goal is improved response time for individual queries.

While there are many papers that propose query processing algorithms, there is a remarkable dearth of literature on real query optimizers. This dearth is even more acute in the area of parallel database machines. In fact, only the Gamma project appears to have published any details of its query optimization techniques. Gamma [DEWI86] employs a query optimizer that is limited in two significant ways. First, the optimizer appears to have only one join tactic, hash join, which makes the system incapable of handling anything but equijoins. Second, according to [GRAE87], the Gamma optimizer produces left-deep query plan trees. In addition, the Gamma prototype appears to be essentially a single-user system, so considerations of resource allocation and scheduling (e.g., memory usage, processor load balancing between queries) appear to be largely ignored.

1.3. The POSTGRES Query Optimizer

The version of the POSTGRES query optimizer described in [FONG86] is basically an adaptation of the design from [SELI79] with the following extensions:

---

[1] Tandem, NonStop, and NonStop SQL are trademarks of Tandem Computers, Inc. As a side note, Tandem's NonStop systems do in fact have a parallel sort operator [TSUK86], but the system was not designed with intraquery parallelism as a design goal.

(1) Incorporation of improvements in query optimization technology since 1979 (e.g., the use of multiple index scans on OR clauses [CHAM81])
(2) Support for user-defined data types, functions, and access methods
(3) A design for procedure support [STON86a]

Since then, we have added code to support a number of additional features as well as parallelism and main-memory. These are described in the sections that follow.

In the text that follows, we will refer to the existing set of code as the ‘‘POSTGRES optimizer’’ and to the proposed design as the ‘‘XPRS optimizer’’.

2. High Function

In terms of new functionality, the recent additions to the repertoire of the POSTGRES query optimizer are support for:

hash join method
functional access methods [LYNC88]
inheritance/union/version queries [ROWE87]
rules [STON87b]

Most of the work described in this section involves simple extensions to the optimizer prototype that existed in early 1987. The chosen solutions were often not the one that would have resulted in the minimal amount of optimization time. For example, two cases involve preprocessing of query (parse) trees and multiple runs of the base optimization engine. These were software engineering decisions. The current POSTGRES system is a prototype, and minimizing the changes to the current stable code sped up implementation and reduced the debugging time required. Furthermore, the query optimization and query processing modules are to be rewritten in C in the near future, affording an opportunity to optimize the optimization code.

Support for hash join is well-understood, so we will skip on to discuss the optimizer support for the other POSTGRES features mentioned above.

2.1. Supporting User-Defined Extensions

The POSTGRES optimizer prototype described in [FONG86] contained code to support user-defined functions, operators and access methods. However, the model assumed in that code (and in [STON86c]) does not work for all possible user extensions. As an example, consider the access method proposed in [LYNC88], a generalization of the extended user-defined indexing as described in [STON86b]. Query optimizers will usually recognize query qualification clauses of the form

```
variable OPERATOR constant
```

(clauses of this form are called ‘‘sargable clauses’’ in [SELI79]) can be used with indexes. The new access method indexes the precomputed result of functions on the index keys rather than the index keys themselves; hence, to support this new access method, the optimizer must be able to recognize that query qualification clauses of the form

```
FUNCTION(variable, ...) OPERATOR constant
```

can also be used with this type of index. A more detailed account of the actual implementation may be found in [AOKI88]. Although some changes had to be made to the POSTGRES system catalogs and the associated lookup code, the total optimizer change was a matter of about 40 lines of code.

2.2. Supporting Union Operations

Several POSTGRES features require the ability to treat multiple base-relation scans as a single virtual base-relation scan; in the discussion that follows, we will call such scans ‘‘multiscans’’ for convenience. For example, consider Figure 2. One can create a group of relations that inherit attributes and methods using a standard multiple-inheritance hierarchy. In this case, the tuples in all four relations are instances of ‘‘PERSON’’. When relations in an inheritance hierarchy are accessed, the base class and its descendants must be scanned; when archival data is accessed, the base relation and the archive relation must be scanned; when a version relation is accessed, the base relation and all intervening version relations must be scanned; and when relations are formed using the POSTQUEL set operations (e.g., ‘‘union’’) both relations must be scanned. Figure 3 gives some examples of queries that require mul-

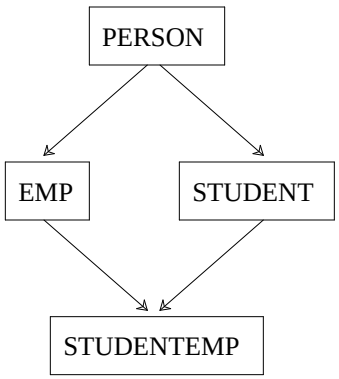

Figure 2. An example of a multiple inheritance hierarchy.

tiscan support. In summary, inheritance queries require one scan per inheriting relation, archival queries require two scans, version queries require $N+1$ scans for a version with N parent relations, and set operations require one scan per relation operated on.

The general idea of the solution is as follows. The query optimizer constructs subplans for each scan within the multiscan and links them together with a ‘‘union’’ plan node which instructs the executor to skip from subplan to subplan, changing target lists, range tables, result relations and other structures at runtime. This is necessary because the different relations have different tuple structures (attributes and attribute numbers), and these executor data structures are, unfortunately, global.

```
create person (name = text)
create emp (empid = int4) inherits (person)
create student (stuid = int4) inherits (person)
create studentemp (sponsor = text) inherits (emp, person)

/* inheritance query - scan "person" and its descendants */
retrieve (p.name) from p in person*

/* archival query - scan "person" from the present back to epoch */
retrieve (p.name) from p in person[,]

/*
 * version query - scan version person2 (implicitly, scan "person1"
 *  and "person" as well
 */
retrieve (p.name) from p in person2

/* set union query - scan "emp" and "student" as one relation */
retrieve (p.name) from p in emp union student
```

Figure 3. Examples of queries requiring multiscan support.

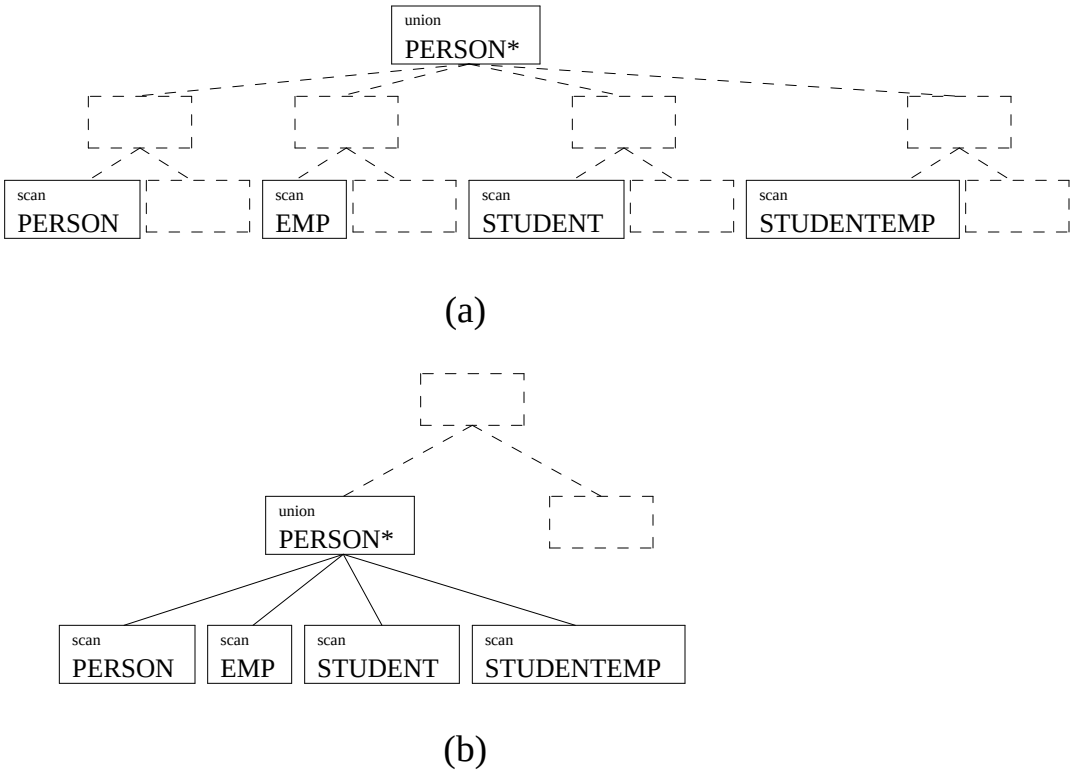

Figure 4. Query plans resulting from two methods of evaluating multiscans.
(a) Reoptimize query for each scan.
(b) Optimize multiscan as an aggregate of scans.

Within this general context, there are two ways to solve the optimization problem. One can use a frontend to the entire optimizer and replan the entire query N times with just the scan relation information changed each time. This means that each multiscan within a query generates a cross product for optimization (e.g., an archive scan of the example inheritance hierarchy would cause the optimizer to run eight times). A plan that might result from this method is shown in Figure 4(a). The query is a join with the inheritance relation **person***, and the resulting plan is simply

four separate join plans connected by a ‘‘union’’ node. The inner join relation is the same in all cases. Another way to solve this problem would be to try to optimize common subexpressions once. That is, if an inheritance relation is at the bottom of a twenty-way join, why reoptimize the join several times? This is the approach shown in Figure 4(b). In this case, the inner join relation is joined to the inheritance relations after they have been unioned together. This would mean really optimizing scans as one multiscan, with all scans producing a single ordering/partitioning, so where the system currently keeps track of each scan path and the ordering it provides, it would have to keep track of the aggregate multiscan. The latter option can be more efficient, in terms of both optimization time (less redundant planning) and execution time (e.g., fewer scans and joins are activated in Figure 4(b) as compared to Figure 4(a)), but there are implementation problems. First, there is the problem of dealing with the multiscan as an aggregate. Second, the scans generate tuples with different structures (attributes, attribute numbers), and the optimizer would have to decide on a common structure for the executor to restructure them into. In any case, the current system is implemented in the first way because it was easier. If queries with multiple multiscans become common, retrofitting true multiple-scan code would not be hard.

2.3. Supporting the POSTGRES Rules System

The query optimizer was equipped with yet another query tree preprocessing module to support the POSTQUEL ‘‘define rule’’ query. As proposed in [STON87a], multiple plans are generated and stored in system catalogs, to be used if rule locks in the database are set off. There are several types of rule locks (one write,

W, and three read, R1, R2, R3).

Consider the following example:

```
define rule r1 is always
replace x (a = y.z, b = 1) where x.c < 5
```

In the following discussion, we will call the ‘‘replace’’ query in this example the *subquery* of the ‘‘define rule’’ query. Besides optimizing the subquery itself, the preprocessor causes the generation of a number of query plans by repeatedly calling the optimizer on modified versions of the subquery. When the preprocessor runs, locks are placed on particular attributes referred to in the subquery (the criteria for placing the locks are listed below). During the repeated invocations of the optimizer, the query plan tree nodes corresponding to these attributes are replaced with *parameter* nodes. When the rule lock on a particular tuple is set off, the plan corresponding to the lock is pulled from the catalogs, the parameter nodes are replaced at with attributes extracted from the tuple, and the plan is executed. Plans and locks are generated as follows:

(1) The subquery is first optimized normally; this plan is not ‘‘parameterized’’ and is used by the executor to place the locks.

(2) For W locks, a parameterized plan for a ‘‘retrieve’’ query (with what is essentially the same target list as the original subquery) is generated. A W lock is generated per update relation variable, e.g., W locks would be placed on *x.a* and *x.b*.

(3) For R1 locks, a parameterized plan for the subquery is generated. A R1 lock is generated per variable assigned to a targetlist variable, e.g., R1 locks would be placed on *y.z*.

(4) For R2 locks, the query to be optimized is always ‘‘retrieve (x = 1)’’. A R2 lock is generated per relation read in the query, e.g., R2 locks would be placed on *x.c*.

(5) No plans are generated for R3 locks.

The preprocessor/frontend simply finds what locks are required, performs some query modification, and then calls the optimizer for each plan. Again, no common-subexpression optimization is done, but since this is only done at rule definition time performance is much less likely to become an issue than with union queries.

3. High Performance

As previously described, the XPRS query processing system attempts to achieve high performance by exploiting two features of the XPRS database machine, large main memory and parallel processors. In the following chapter we describe how the XPRS query optimizer takes advantage of these features.

3.1. Exploiting Large Main Memory

The explicit and straightforward consideration of main memory (in the form of buffer space) would simply add another variable to the already-intractable query optimization problem. Hence, some simplifying assumptions must be made in order to prevent disastrous optimization run times, at the cost of possible plan non-optimality. With respect to main memory, the XPRS optimizer assumes that plan structure (tree topology, join order, choice of join and scan operators) is essentially invariant over memory changes.

This assumption essentially eliminates main memory as an optimization degree-of-freedom, although it is still important as a means of choosing between query plan operators. Hence, a single plan, optimized with the expectation of having some large amount of main memory, should suffice for all reasonable amounts of memory. This has a certain intuitive appeal. With respect to the query plan operators

used, the expected-case differences between sequential and index scans and nested-loop and hashed joins are so clearcut that the choice of operators seems unlikely to change at all between trees optimized for two given amounts of memory.

In order to test this intuitive argument, we modified the POSTGRES optimizer to consider main memory usage as a cost function factor and examined the output of this modified optimizer as the amount of available buffer space changed. Hash joins were assumed to follow the execution model of [DEWI84] (as will be described later in an example).

Random queries were generated assuming the following parameters: 3- to 8-way joins, 1 to 10 million tuples per relation, 80% equijoins (20% other POSTGRES operators besides bit-equality), 50% 4-byte integers (50% other data types) with all integer columns indexed, 80% conjunctive connectives (20% disjuncts; no pure relational products were generated). No actual tuples were generated, but appropriate statistics were fabricated and loaded into the catalogs to make the join and qualification clause selectivities more realistic. One hundred of these random queries were optimized for each of four test databases assuming buffer pools ranging from 16 to 16384 buffers of 8KB each (the default size of a POSTGRES disk buffer) — that is, 128KB to 128MB. The databases differed in the number (10-12) and composition (number of tuples, number and types of attributes) of relations.

The results of the tests are unsurprising. The plans generated by the POSTGRES optimizer were structurally identical, with the exception of six plans that interchanged the join order of two leaf scans (i.e., a join between two scan nodes, as

opposed to, say, a scan node and a join result). In other words, the only differences seen did not affect plan cost at all — a join between a sequential scan of relation A to a sequential scan of relation B has the same cost as a join of B to A. Hence, at least for the POSTGRES optimizer, the memory-invariance assumption appears to be a reasonable one.

3.2. Exploiting Parallelism

Assume the main-memory model just given. Once a conventional query plan tree has been constructed, individual query plan operators can be parallelized. In addition, the query plan can be subdivided into subplans which can be run in parallel. We term these two types of parallelism *intrafragment* parallelism and *interfragment* parallelism for reasons that will become apparent in the discussion that follows.

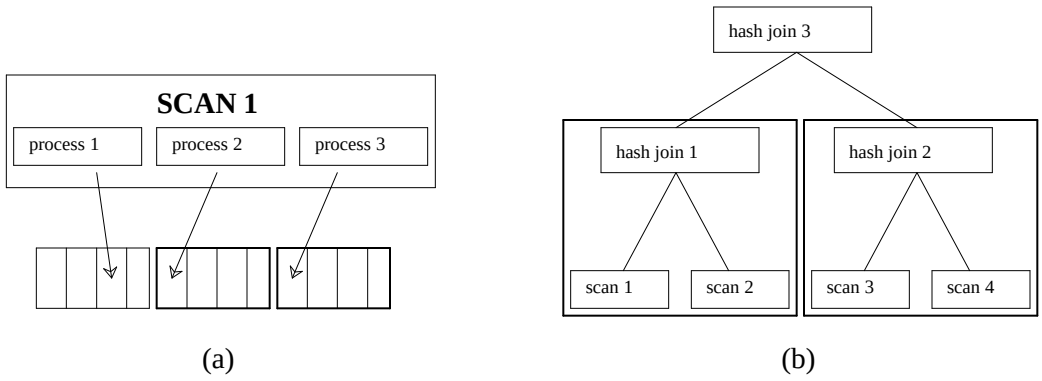

Figure 5. Two kinds of parallelism.
(a) Intrafragment parallelism. One scan is broken up into three subscans.
(b) Interfragment parallelism. A tree is broken up into independent subtrees.

Figure 5 shows examples of the two types of parallelism. For now, we define *fragment* as a unit of multiple-operator parallelism.

3.2.1. Intrafragment Parallelism

The individual query plan operators (plan nodes) can be parallelized using techniques adapted from [BITT83, MENO86, RICH87]. The operators used in XPRS (sequential and index scans, nested-loop and hash joins) are exactly those that can be trivially decomposed into an arbitrary number of independent threads or processes.

There may be several operators within a given fragment, and each operator within a fragment is decomposed into the same number of processes. This has several advantages over allowing operators with differing numbers of processes. First, no mux/demux ‘‘plumbing’’ is required, as would be required to interface operators with differing numbers of processes. Second, in terms of plan data structures, each thread within a fragment basically corresponds to a separate query tree. As a pragmatic matter, this convention permits the adaptation of the existing POSTGRES executor code, essentially eliminates the overhead of process/thread switches, and simplifies the processor scheduling problem.

As an example, we describe a parallel hash join algorithm consistent with our memory and processor models. It is basically the algorithm from [DEWI84] with each set of hash buckets handled in *NPROC* parallel parts by *NPROC* different processes (where *NPROC* is a tuning parameter). The following phases are illustrated in Figure 6.

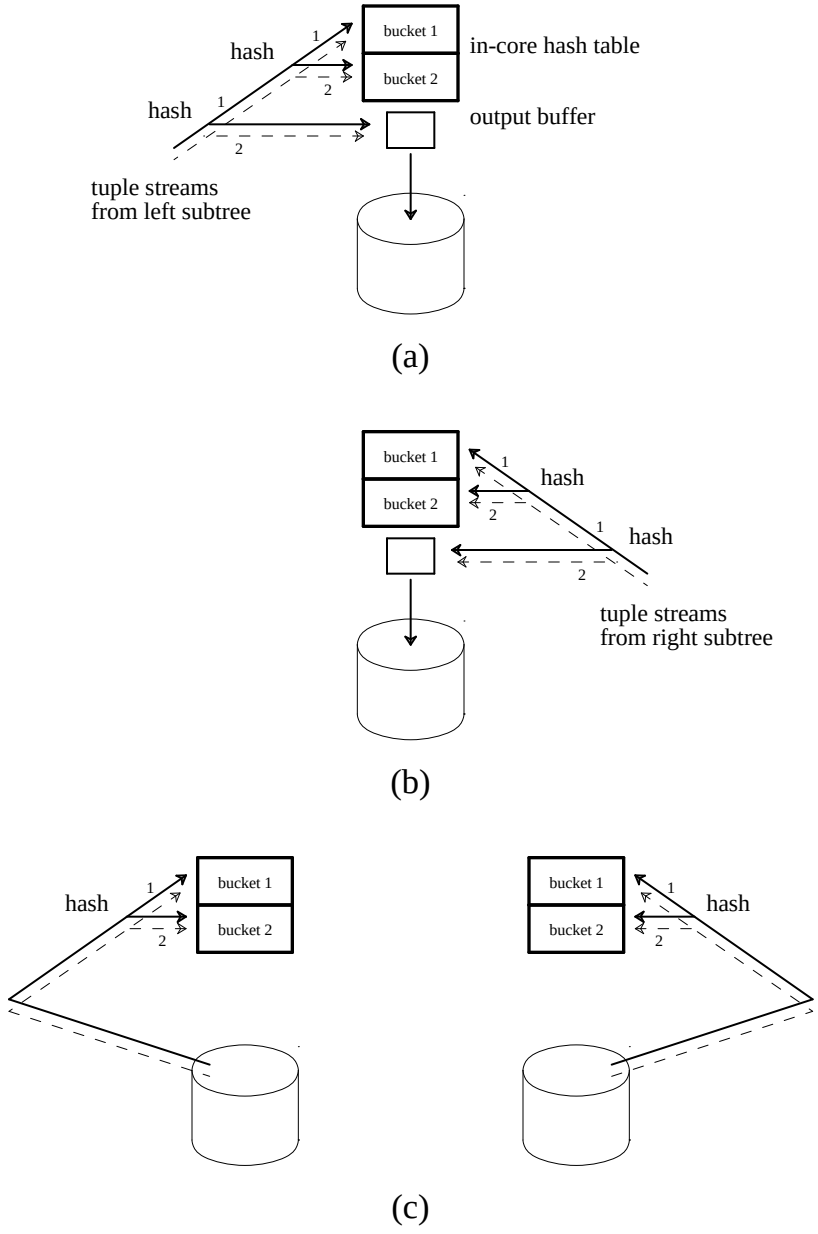

Figure 6. A shared-memory parallel hashjoin algorithm.
(a) Initial build phase (first batch hashed).
(b) Initial probe phase (first batch joined).
(c) Subsequent build and probe phases (*i* th batch joined).

(a) **Initial build phase:**
The *NPROC* tuple streams from the left child of the hash node are hashed into *NBATCH* ''batches.'' As in [DEWI84], batches are sized such that one batch will essentially fill the available buffer space. The first batch is immediately rehashed into the buckets of a main-memory hash table; the other $NPROC-1$ batches are spooled into buffers and written out to disk.

(b) **Initial probe phase:**
The *NPROC* tuple streams from the right child of the hash node are hashed into batches using the same hash function as in step 1. Tuples hashed into the first batch are rehashed into the buckets of the main-memory hash table and joined with the first-batch tuples of the left-subtree immediately. Tuples hashed into the other $NPROC-1$ batches are spooled.

(c) **Subsequent build and probe phases:**
For each batch $i > 1$, The ith left-subtree batch is read in *NPROC* tuple streams into the buffers allocated for the main-memory hash table. Once the hash table is built, the ith right-subtree batch is read in *NPROC* tuple streams. As tuples come in, they are joined with the left-subtree tuples and piped up.

The key points are as follows. First, main memory is used to hold the first batch and keep it from being written out to disk; this caching saves a number of I/Os proportional to the memory allocated for this purpose (i.e., to the size of the main-memory hash table). Second, all *NPROC* processes allocated to the fragment are kept as busy as possible.

3.2.2. Interfragment Parallelism

Two key problems must be solved if a system allows multiple portions of a query plan to be run at once. The first is activation of the portions, and the second is scheduling among them.

A system which uses left-deep query plans, e.g., GAMMA, can simply activate its operators bottom-up. XPRS must solve the activation problem in a more general way, since XPRS allows bushy query plan trees. The rules for forming fragments are as follows:

F1. A node belongs to up to two fragments, one for each (left and right) subtree.

F2. The left subtree of a node that forms a temporary (e.g., a hash join) forms the top node of a new fragment.

The basic rationale for this is the obvious one. Forming a temporary places a large bubble in the upward stream of tuples. Hence, it does not make sense to place operator nodes on either side of one that forms a temporary in the same fragment, since the two sets of non-temporary-forming operator nodes cannot run concurrently for the duration of their activation.

The rules for activating fragments are equally simple. They are:

A1. A given fragment may not be started if a fragment containing a node below the given fragment in the query plan tree has not been started. Hence, fragments are activated bottom-up.

A2. The left subtree of a hash join node must have run to completion before the hash join node itself can run, since the main-memory hash table must be formed before tuples from the right subtree can be hashed into it.

Again, the reasoning is that of dataflow dependency.

A sample left-deep query plan and its fragmentation is shown in Figure 7. The query

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

might result in the plan shown in the top of the figure and then be broken up into three fragments according to rules F1 and F2. Following rules A1 and A2, the fragments are then activated in the left-to-right order shown. When the first fragment is activated, node 3 constructs a hash table from the DEPT tuples streaming up from node 1. In the second fragment, node 3 hashes the tuples from the scan of EMP

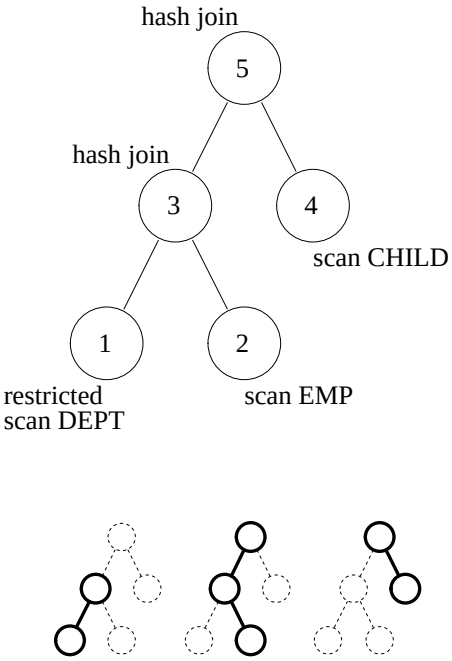

Figure 7. A sample left-deep query plan tree and its fragments.

(node 2) into its table (performing the equijoin with DEPT) and node 5 begins formation of its hash table. Finally, when the third fragment starts, tuples from the scan of CHILD (node 4) are hashed into node 5's table (performing the equijoin with EMP) and the results are returned to the user.

The previous example, being of a left-deep tree, can contain intrafragment parallelism but cannot contain interfragment parallelism. We now consider a slightly more complicated example that does require consideration of interfragment parallelism. Say the optimal query plan for

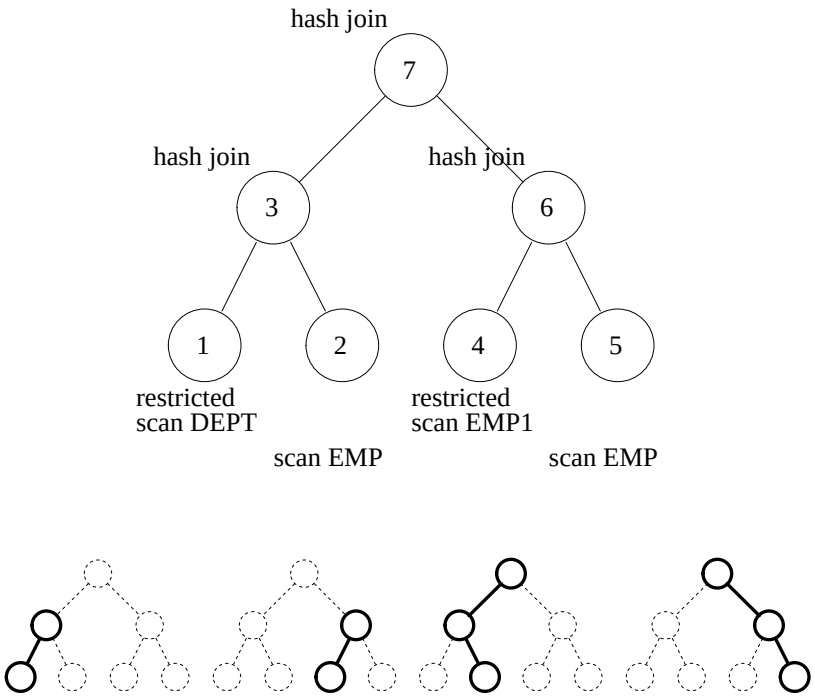

Figure 8. A sample bushy query plan tree and its fragments.

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

is as shown in the top of Figure 8. This plan is fragmented as shown in the bottom of the figure. In this case, the third fragment depends on the results of the first, and the fourth depends on the results of the first and third; however, the first and second fragments do not depend on any other fragments and can be run concurrently. Hence, the execution of this query plan would proceed in a manner slightly different from the previous example. If there are sufficient resources (i.e., memory and processors), the

first and second fragments are activated simultaneously, hashing the DEPT and EMP1 scan tuples from nodes 1 and 4 into tables constructed by nodes 3 and 6. The third fragment is then activated, hashing the tuples from the EMP scan (node 2) into the table controlled by node 3 and placing the results at the disposal of the hash operator in node 7. Finally, the fourth fragment can be activated, hashing the EMP tuples from node 5 into the node 6 hash table and then hashing these results into the hash table in node 7.

The algorithms to control this fragmentation and activation are relatively simple. Sample data structures and code for this process are shown in Figures 9 and 10. The code samples show that in order to fragment, we need only perform a simple depth-first traversal of the query plan while keeping track of the fragment numbers of the current node, its children, and the highest fragment number generated so far (in order to generate fragment numbers for the next new fragment). In addition to keeping track of the fragment numbering, the function *new-frag* also keeps track of dataflow dependencies; these dependencies are used to create the partial ordering of

```
(defstruct (fraginfo (:type :vector))
  number        ; fragment id for this fragment
  prereqs       ; fragment ids of fragments which must run before this one
  head)         ; top node of this fragment

(defstruct (fragment (:type :vector))
  left          ; fragment id for the left
  right         ;                          and right children of this node
  max)          ; highest fragment id in this node or its children
```

Figure 9. Fragmentation data structures.

```
;;; fragment-plan
;;;     Destructively marks 'node' and its subnodes with their fragment
;;;     memberships ("fragment" nodes).  Returns a dependency list (list of
;;;     "fraginfo" nodes).
(defun fragment-plan (node)
  (let ((frags
         (vector (list (make-fraginfo :number 1)))))
    (fragment-subplan node frags 1 1)
    frags))

;;; fragment-subplan
;;;     The left and right subnodes are always in fragment 'current' unless
;;;     'node' is a hashjoin, in which case 'node' and its left subnode become
;;;     members of a new fragment.  Returns the highest fragment number used
;;;     in this node or its children.
;;;
;;;     'node' is the plannode to be marked.
;;;     'frags' is a structure containing the current dependency list.
;;;     'current' is the current fragment number.
;;;     'max' is the highest fragment number known (used to generate new
;;;             fragment numbers).
(defun fragment-subplan (node frags current max)
  (assert (plannode-p node))
  (let* ((left-fragment
          (if (get_lefttree node)
              (cond ((hashjoin-p node)
                     (let ((new-max (1+ max)))
                       (new-frag (get_lefttree node) frags current new-max)
                       (fragment-subplan (get_lefttree node)
                                         frags
                                         new-max
                                         new-max)))
                    (t
                     (fragment-subplan (get_lefttree node)
                                       frags
                                       current
                                       max)))))
         (right-fragment
          (if (get_righttree node)
              (fragment-subplan (get_righttree node)
                                frags
                                current
                                (if left-fragment
                                    (fragment-max left-fragment)
                                  max))))

         (fragment
          (make-fragment :left (if left-fragment
                                   (fragment-right left-fragment)
                                 current)
                         :right current
                         :max (max max
                                   (if right-fragment
                                       (fragment-max right-fragment)
                                     max)
                                   (if left-fragment
                                       (fragment-max left-fragment)
                                     max)))))
    (set_state node fragment)
    fragment))


;;; new-frag
;;;     Adds a fraginfo node for fragment number 'new-number' to the list
;;;     of fraginfo nodes kept in 'frags'.  In addition, fragment number
;;;     'new-number' is added to the dependency list of its parent.
(defun new-frag (node frags parent-number new-number)
  (let* ((parent-frag
          (find parent-number (vref frags 0)
                :key #'(lambda (x) (fraginfo-number x)))))
    (assert (not (null parent-frag)))
    (push new-number (fraginfo-prereqs parent-frag))
    (setf (vref frags 0)
          (push (make-fraginfo :number new-number
                               :head node)
                (vref frags 0)))))
```

Figure 10. Code for fragmentation and generation of dependency information.

fragments which are, in turn, used to generate the activation order. Hence, both the fragmentation and determination of activation order are performed at once in functions that perform only one traversal of the query plan tree.

3.3. Scheduling Problems

Up to this point, we have implicitly or explicitly assumed unlimited resources. In practice, the number of processors and buffers available may be severely limited. Hence, we now turn to algorithms for the allocation and scheduling of system resources.

Note that the availability of system resources is known only at the time at which the query is run, and is not known at query optimization time. One result of the query optimization algorithms described above is that resource allocation and scheduling can be practically ignored during the optimization phase; hence, the running time of the XPRS optimizer is basically what it would be in a system using a conventional optimizer, since all allocation and scheduling problems are left for the executor.

3.3.1. Number of Concurrent Processes Per Fragment

Because operating system processes have certain types and amounts of overhead associated with them, it is clear that there should be some control over the number of threads or processes into which the operators of a given fragment are divided. The following factors should be considered in calculating the number of processes into which a fragment should be divided:

Disk parallelism

If there are N_{drives} disk drives and each drive can handle N_{I/Os_per_drive} I/O requests (e.g., a drive with two sets of heads running under an operating system that permits only synchronous I/O can be assigned 4 simultaneous I/O requests), then the maximum number of useful processes considering only the number of drives is

$$MAXPROC_{disk} = N_{I/Os\_per\_drive} N_{drives}$$

Limiting process overhead

Say the time spent starting, controlling, and terminating a process within a fragment is $T_{process}$ and the total runtime of a fragment is $T_{fragment}$. If we put a ceiling on the process overhead time by limiting the number of processes to no more than $MAXPROC_{CPU}$, the total time wasted will be

$$fragment\ overhead = MAXPROC_{CPU} T_{process}$$

so in order to limit process overhead to some fraction $P_{overhead}$ of the fragment runtime, we should limit the number of running processes to no more than

$$MAXPROC_{CPU} = \frac{P_{overhead} T_{fragment}}{T_{process}}$$

Limiting I/O overhead

Similarly, since each I/O has some amount of overhead, it makes little sense to start a huge number of processes each of which do one or less I/Os. If there are an estimated $N_{I/Os}$ in the fragment, additional processes should only be started if each can get at least $P_{I/Os\_per\_process}$ I/O requests.

$$MAXPROC_{I/O} = \frac{N_{I/Os}}{P_{I/Os\_per\_process}}$$

Current load

In order to smooth the degradation of system performance, the current system load should be considered before starting more processes. In BSD UNIX, the available metric for system load is the *load average*, or average number of processes in the run queue. We will refer to this number as *avenrun*.

Considering all of these factors, one equation for the number of processes to start for a given fragment is:

$$MAXPROC = min(MAXPROC_{disk}, MAXPROC_{CPU}, MAXPROC_{I/O})$$

$$NPROC = \frac{MAXPROC}{(\frac{avenrun}{processors} + 1)^2}$$

Again, all operators in the fragment are divided into $NPROC$ parallel portions and so $NPROC$ copies of the runtime data structures must be created.

3.3.2. Number of Concurrent Fragments

In queries where interfragment parallelism is possible, it may turn out that $NPROC \leq MAXPROC$. In this case, XPRS immediately attempts to find fragments that are (1) independent of the currently-running fragment and (2) capable of being run given the available amount of buffer space.

3.3.3. Buffer Space

In the ideal memory case, which we have assumed until now, there is always enough memory to run whatever query plan operator method or number of processes desired. This, of course, will not always be the case. If fragments with arbitrary

buffer usages are started, they will compete for buffers and buffer-pool thrashing may result. This subsection describes how XPRS manages its buffer space and attempts to prevent thrashing.

Memory usage calculations are of course totally dependent on the implementation of the query plan operators. For the sake of concreteness in the following discussion, we will assume that query plan operators use essentially no buffer space with the exception of hash join, which will behave as described in the hash join example given above. That is, in the initial phases, hash join requires space for the main-memory hash table as well as a spool buffer for all batches but the first. Subsequent phases require enough memory for the hash table and for a buffer to pull tuples from the current spooled batch. Hence, the minimum memory requirement for this hash join algorithm if a total of $MAXBUF$ buffers are available is

$$\frac{MAXBUF}{NBATCH} + NBATCH$$

The following algorithms make no particular assumptions about the basic buffer manager replacement mechanisms, although we expect to replace the current POSTGRES buffer manager with a more intelligent buffer manager along the lines of [CHOU85] On top of the basic buffer manager, we implement a buffer *reservation* system. That is, before running, a fragment must be able to reserve an amount of buffer space appropriate for its expected needs, and after running, it must remove its reservation for the buffer space. If the buffer manager cannot provide this amount of space from the buffers that are still unreserved, the fragment cannot run. Note, however, that the reservation is in fact a *hint* for the buffer manager rather than a memory

usage *quota* for the fragment. Hence, thrashing may still occur if the executor grossly underestimate the memory usage of some fragment(s).

If a fragment cannot run, the query executor has two options. It will first attempt to change the original fragment's resource requirements so that it can run; tactics for doing so are described below. Should these tactics fail, the executor must simply go on and try to find a fragment that it can run in the amount of memory it has available to it.

How can one reduce the memory usage of a fragment? Since hash joins are assumed to be the only operators that use nontrivial amounts of memory in XPRS, the tactics used simply concentrate on reducing the memory usage of hash joins in different ways.

The first tactic is simply to break the fragment into two subfragments. If a fragment contains multiple hash join nodes, the executor can insert a spool node just below the topmost hash join node in the fragment. Everything above this spool node becomes part of a new fragment. The memory requirements of the (newly beheaded) original fragment can then be reevaluated and re-passed to the buffer manager, given that the memory requirement of supporting one of the fragment's large main-memory hash tables has just been eliminated.

The second tactic assumes that the first tactic has been applied until there is only one hash table in the current fragment. Now the problem may simply be thought of as trying to reduce the amount of memory used by a particular hash join. This can be accomplished by increasing *NBATCH*, thereby reducing the effectiveness of the batch

cache and slowing down the join.

Again, it must be noted that the tactics used to reduce memory usage are dependent on the implementation of the query plan operators. In the original proposal found in [STON89], it was assumed that each process in a hash join would require its own bucket. This created a further memory-usage opportunity, i.e., reducing the fragment parallelism. By simplifying the hash join algorithm and reducing its buffer usage, this optimization has been eliminated.

4. Conclusions

4.1. Summary of Design Points

In this thesis, we have discussed some of the techniques used in the POSTGRES query processor that are used to support new functionality and exploit the features (multiple processors, large main memory) of the other components of the XPRS database machine. In terms of increased functionality, we described the algorithms for handling generalized extended user-defined indexing, union relations and the definition of rules. We have also described the modifications made to increase performance, which include algorithms for fragmentation and activation of fragments, control of fragment parallelism, control of fragment memory usage, and (an example of) parallel hash join.

4.2. Current Status and Future Work

The functionality extensions are in place and have been tested over the course of the last year. The implementation of union relations in particular has recently been

made more robust (if no more efficient). Implementation of the XPRS extensions is awaiting completion of other features of POSTGRES. Hence, it is likely that the design proposal described here will be reworked yet again in order to reflect the interface and architectural changes pending to the base POSTGRES system.

There are still a number of features yet unimplemented in POSTGRES that will require extensive changes to the optimizer. The most significant of these is query-language procedures, which will almost certainly require a great deal of rewriting. In addition, making the optimizer generate bushy query plan trees will likely require a significant amount of time (though not coding) since the assumption that the inner join relation is a scan result is implicit throughout the optimizer code. The more efficient implementation of union relations explained in Section 2 should be carried out if archival, version, or inheritance queries are expected to be heavily used. Finally, when a hashjoin algorithm is settled upon, the cost functions, catalog entries, and hashjoin-handling code in the optimizer will probably have to change.

References

[AOKI88] P. M. Aoki, ‘‘Implementation of Generalized Extended User-Defined Indexing in POSTGRES’’, CS292N Project Report, Univ. of California, Berkeley, CA, Dec. 1988.

[BITT83] D. Bitton, H. Boral, D. J. DeWitt and W. K. Wilkinson, ‘‘Parallel Algorithms for the Execution of Relational Database Operations’’, *Trans. Database Systems 8*, 3 (Sep. 1983).

[BORR88] A. Borr and G. Putzolu, ‘‘High Performance SQL Through Low-Level System Integration’’, *Proc. 1988 ACM-SIGMOD Conf. on Management of Data*, Chicago, IL, June 1988.

[CHAM81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu and B. W. Wade, ‘‘A History and Evaluation of System R’’, *Comm. of the ACM 24*, 10 (Oct. 1981).

[CHOU85] H. Chou, *Buffer Management of Database Systems*, Ph.D. Thesis, Univ. of Wisconsin, Madison, WI, May 1985. (Also available as Computer Sciences Tech. Rep. 597).

[DEWI84] D. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker and D. Wood, ‘‘Implementation Techniques for Main Memory Database Systems’’, *Proc. 1984 ACM-SIGMOD Conf. on Management of Data*, Boston, MA, June 1984.

[DEWI86] D. J. DeWitt, G. Graefe, K. B. Kumer, R. H. Gerber, M. L. Heytens and M. Muralkrishna, ‘‘Gamma: A High-Performance Dataflow Database Machine’’, *Proc. 12th VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[DOUG89] F. Douglis and J. Ousterhout, ‘‘Log-Structured File Systems’’, *IEEE Spring Compcon 1989*, San Francisco, CA, Feb. 1989, 124-129.

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

[GRAE87] G. Graefe, *Rule-Based Query Optimization in Extensible Database Systems*, Ph.D. Thesis, Univ. of Wisconsin, Madison, WI, Nov. 1987. (Also available as Computer Sciences Tech. Rep. 724).

[JARK84] M. Jarke and J. Koch, ‘‘Query Optimization in Database Systems’’, *Computing Surveys 16*, 2 (1984).

[KOOI82] R. Kooi and D. Frankforth, ‘‘Query Optimization in INGRES’’, *IEEE Database Engineering*, Sep. 1982.

[LYNC88] C. A. Lynch and M. Stonebraker, ‘‘Extended User-Defined Indexing with Application to Textual Databases’’, *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[MENO86] J. Menon, ‘‘Sorting and Join Algorithms for Multiprocessor Database Machines’’, IBM Research Report RJ5049, IBM Research Laboratory,

San Jose, CA, Feb. 1986.

[OUST88] J. K. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, ‘‘The Sprite Network Operating System’’, *Computer 21,* 2 (Feb. 1988).

[PATT88] D. Patterson, G. Gibson and R. H. Katz, ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID)’’, *Proc. 1988 ACM-SIGMOD Conf. on Management of Data*, Chicago, IL, June 1988.

[RICH87] J. P. Richardson, H. Lu and K. Mikkilineni, ‘‘Design and Evaluation of Parallel Pipelined Join Algorithms’’, *Proc. 1987 ACM-SIGMOD Conf. on Management of Data*, San Francisco, CA, May 1987.

[ROWE87] L. A. Rowe and M. R. Stonebraker, ‘‘The POSTGRES Data Model’’, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987.

[SELI79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, ‘‘Access Path Selection in a Relational Database Management System’’, *Proc. 1979 ACM-SIGMOD Conf. on Management of Data*, Boston, MA, June 1979.

[STON76] M. Stonebraker, E. Wong, P. Kreps and G. Held, ‘‘The Design and Implementation of INGRES’’, *Trans. Database Systems 1,* 3 (Sep. 1976).

[STON86a] M. Stonebraker, ‘‘Object Management in POSTGRES Using Procedures’’, UCB/ERL Tech. Rep. M86/59, Univ. of California, Berkeley, CA, July 1986.

[STON86b] M. R. Stonebraker, ‘‘Inclusion of New Types in Relational Data Base Systems’’, *Proc. 2nd IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1986.

[STON86c] M. R. Stonebraker and L. A. Rowe, ‘‘The Design of POSTGRES’’, *Proc. 1986 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, June 1986.

[STON87a] M. Stonebraker, E. Hanson and S. Potamianos, ‘‘A Rule Manager for Relational Database Systems’’, UCB/ERL Tech. Rep. M86/85, Univ. of California, Berkeley, CA, June 1987.

[STON87b] M. Stonebraker, E. Hanson and C. Hong, ‘‘The Design of the POSTGRES Rules System’’, *Proc. 3rd IEEE Data Engineering Conf.*, Los Angeles, CA, Feb. 1987.

[STON88] M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, ‘‘The Design of XPRS’’, *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[STON89] M. Stonebraker, P. M. Aoki and M. Seltzer, ‘‘Parallelism in XPRS’’, UCB/ERL Tech. Rep. M89/16, Univ. of California, Berkeley, CA, Feb. 1989.

[TSUK86] A. Tsukerman *et al.*, ‘‘FastSort: An External Sort Using Parallel Processing’’, Tandem Tech. Rep. TR86.3, Tandem Computers, Cupertino, CA, 1986.

[WENS88] S. Wensel, editor. ‘‘The POSTGRES Reference Manual’’, UCB/ERL Tech. Rep. M88/20, Univ. of California, Berkeley, CA, Mar. 1988.

[WONG76] E. Wong and K. Youssefi, ‘‘Decomposition: A Strategy for Query Processing’’, *Trans. Database Systems 1*, 3 (Sep. 1976).