1. INTRODUCTION

Relational database management systems are widely available in the commercial market. Currently available systems run on a variety of hardware, ranging from DEC minicomputers (e.g., Informix, Oracle, Unify) to IBM mainframes (e.g., Adabas, Datacom/DB, DB2, IDMS/R). These systems have been successful because of the merits of the relational model, as first illustrated by two research prototypes, INGRES [STON76] and SYSTEM-R [ASTR76]. INGRES and SYSTEM-R not only illustrated the feasibility of the relational model, but their respective query languages, QUEL [HELD75b] and SQL [CHAM76], also showed that it is possible to ask queries without explicitly specifying access paths. The ability to support these non-procedural query languages is a result of sophisticated query optimization algorithms. INGRES introduced a technique known as query decomposition [WONG76], while SYSTEM-R employed an exhaustive search algorithm [SELI79]. Largely due to the success of these algorithms, relational systems were made efficient. Therefore, coupled with the simplicity and uniformity of the relational model, it is not surprising that relational databases have established a formidable presence in the commercial market.

The relational model, however, has been criticized for its impoverished semantics [KENT79], [ZANI83] and inability to provide strong support for non-business applications [HASK82]. In recent years, researchers have been investigating the possibility of extending query languages in relational systems to support new application areas as well as better semantics. Examples include:

- a proposal to support abstract data types (ADTs) and operators in INGRES to improve the semantics of applications [FOGG82], [ONG82]
- a new language, QUEL*, to support the transitive closure operations required in artificial intelligence applications [KUNG84]
- a proposal to support QUEL as a data type to increase the data modeling power of relational systems [STON84], [STON85b]

- a proposal to support rules and triggers in a relational system to provide inference and forward chaining needed in expert system applications [STON85a].

The ideas behind these proposals are being incorporated into POSTGRES (‘‘POSTinGRES’’), a next-generation relational database system being built at the University of California, Berkeley [STON86b]. Providing better support for engineering design and artificial intelligence applications are among the goals of POSTGRES. To meet these goals, POSTGRES will support extendible and user-defined access methods [STON86a] as well as abstract data types, transitive closure queries, procedural data fields, triggers, and rules. The query language for the system will be called ‘‘POSTQUEL.’’

POSTGRES is still in its preliminary implementation phase. However, a query optimizer for the system has been built. Although the basic optimization algorithm is modeled after the SYSTEM-R approach, there are many other issues that the optimizer must contend with given the novel features of POSTGRES. Section 2 will introduce these features. Section 3 will then discuss design decisions that were made in formulating optimization algorithms. Section 4 will discuss implementation decisions made, and finally section 5 will evaluate the performance of the POSTGRES optimizer by comparing it with the query optimizer of another relational system.

2. POSTQUEL

The next two subsections will motivate the different issues that the optimizer must consider through several examples. Then, it will indicate how these new features affect the optimizer.

2.1. An Extendible Type System

One of the features that POSTGRES supports are abstract data types (ADTs). An ADT facility allow users to define their own data types, simplifying representation of complex information. For example, a user who must store box coordinates in his database can define a data type called ‘‘boxcoordinates.’’ From here, he can define a relation BOX with a coordinates field of type ‘‘boxcoordinates.’’ The unit square box shown in figure 2.1, therefore, would be represented as shown in figure 2.2. This is in contrast to figure 2.3.

POSTGRES also allows users to define operators to be used in conjunction with user-defined data types. By defining an operator ‘‘AREA,’’ a query to compute the area of the above box would be expressed as:

retrieve (a = AREA(BOX.coordinates)) **where** BOX.boxid = 1

rather than:

retrieve (a = sqrt (sqr (BOX.x1 - BOX.x2) + sqr (BOX.y1 - BOX.y2)) +
sqrt (sqr (BOX.x1 - BOX.x4) + sqr (BOX.y1 - BOX.y4)))
where BOX.boxid = 1.

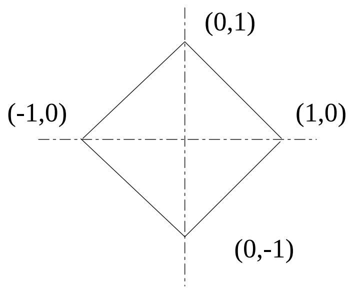


Figure 2.1

BOX	boxid	coordinates
	1	((-1,0), (0,1), (1,0), (0,-1))

Figure 2.2
Relation with user-defined type ‘‘boxcoordinates’’

BOX	boxid	x1	y1	x2	y2	x3	y3	x4	y4
	1	-1	0	0	1	1	0	0	-1

Figure 2.3
Relation without user-defined types

In addition, an operator ‘‘AREAEQ’’ (area equals) can be defined to find all boxes whose area is equal to some desired value. For example,

> **retrieve** (BOX.all) **where** BOX.coordinate AREAEQ 10

finds all boxes whose area equals ten. Similarly, operators like ‘‘AREALT’’ (area less than) and ‘‘AREAGT’’ (area greater than) can also be defined.

The operators, AREAEQ, AREAGT, and AREALT, are quite similar to the conventional relational operators, =, >, and <. Therefore, in the same way that users can specify access methods like B-trees and hash tables to provide efficient access when used with conventional operators, users should also be able to specify access methods to efficiently scan relations when non-standard operators appear within restriction clauses. One solution, which POSTGRES supports, is to allow users to extend existing access methods so they can be used with a new set of operators. For example, by maintaining coordinate records within a B-Tree sorted in ‘‘AREALT’’ order rather than a more conventional ‘‘<’’ or ‘‘>’’ order, a user has an access method that provides efficient access to queries whose restrictions contain the AREAEQ, AREAGT, or AREALT operators. Another example is a hash table that can be constructed using the operator ‘‘AREAEQ’’ rather than ‘‘=.’’

The technique just described is suitable provided there exists an appropriate access method upon which extensions can be made. However, suppose a user defines an operator contained-in, ‘‘≪,’’ that returns true if the left operand is spatially contained within the right operand. To provide efficient access to this operator, two-dimensional access methods are required, e.g. an R-tree [GUTT84] or a K-D-B Tree [ROBI81]. Since conventional databases do not support two-dimensional access methods, an extension of an existing access method is not possible. To alleviate this problem, POSTGRES allows users to define their own access methods. Details on user-defined access methods are discussed in [STON86a].

In summary, with an extendible type system, users can build data types to suit their application needs, operators to manipulate the new types, and access methods to provide efficient access to queries containing these new operators.

2.2. Procedural Data Fields

Existing relational databases do not provide good support for storage of complex objects. For example, if a complex object consists of a single box, circle, and triangle, this information would be represented as shown in figure 2.4. As a result, three join queries must be executed to retrieve all information about subobjects within this complex object. In the more general case where a complex object is composed of up to *n* different subobject types, a user would have to execute *n* join queries. Without extra information indicating which subobject types are actually contained within a desired object, the user has no choice but to execute all *n* queries. This is quite inefficient, particularly when *n* is large, because as indicated in the previous sentence, many of the join queries are unnecessarily executed.

BOX (bid, bdata)
CIRCLE (cid, cdata)
TRIANGLE (tid, tdata)

COBJECT	coid	objtype	oid
	1	box	2
	1	circle	3
	1	triangle	4

retrieve (BOX.all) **where**
    BOX.bid = COBJECT.oid **and**
    COBJECT.objtype = ‘‘box’’ **and**
    COBJECT.coid = 1

retrieve (CIRCLE.all) **where**
    CIRCLE.cid = COBJECT.oid **and**
    COBJECT.objtype = ‘‘circle’’ **and**
    COBJECT.coid = 1

retrieve (TRIANGLE.all) **where**
    TRIANGLE.tid = COBJECT.oid **and**
    COBJECT.objtype = ‘‘triangle’’ **and**
    COBJECT.coid = 1

Figure 2.4
Storage of complex objects in a relational system

The basic problem here is that the relational model is not well-suited for representing hierarchical relationships. As a solution, Stonebraker has proposed embedding queries within data fields and using these queries to express the hierarchical relationship between the corresponding tuple and information elsewhere in the database [STON84]. Using this idea, which POSTGRES supports, our complex object example is now represented as shown in figure 2.5. To retrieve information executed by the queries embedded within this data field, the user would issue the following query:

execute (COBJECT.components) **where** COBJECT.coid = 1.

Thus, *n* join queries reduce to a single **execute** query. In addition, users can selectively retrieve information linked through these hierarchies by nesting attributes in a manner similar to the proposal in GEM [ZANI83]. For example, to retrieve triangle information for a particular complex object, a user would nest ‘‘tdata’’ within ‘‘components’’ as shown below:

retrieve (COBJECT.components.tdata) **where** COBJECT.coid = 1.

In general, attributes can be nested to an arbitrary number of levels.

2.3. The POSTGRES Optimizer

Query optimization decisions are made based upon the characteristics of operators appearing within queries as well as the index types defined on relations. In a conventional optimizer, information about operators and access methods can be ‘‘hardwired’’ into the optimization code because there are only a

COBJECT	coid	components
	1	**retrieve** (BOX.all) **where** BOX.bid = 2 **retrieve** (CIRCLE.all) **where** CIRCLE.cid = 3 **retrieve** (TRIANGLE.all) **where** TRIANGLE.tid = 4

Figure 2.5
Storage of complex objects with procedural data fields

fixed number of operators and access methods within the system. Such a solution would not suffice in a system like POSTGRES where an arbitrary number of operators and access methods are at a user's disposal. Consequently, this was an issue that had to be considered when designing the POSTGRES optimizer.

The optimizer also must consider queries containing nested attributes. As section 3.5 will describe, there is a clean and simple solution to this problem, which only requires that the optimizer apply the basic planning algorithm once for each nesting level.

Rules and triggers will be processed using query modification [STON75]. This aspect of POSTGRES will not be discussed in this report because query modification is being implemented in a module separate from the optimizer. For details, see [STON86d].

Sophisticated algorithms have been proposed to optimize transitive closure queries as well as sets of queries. This is done by transforming sequences of database operations to equivalent sequences that execute more efficiently. This report, however, will not discuss these techniques any further because the topic is outside the scope of this project. For details, see [SELL85], [SELL86].

3. DESIGN OF THE OPTIMIZER

This section will first describe the optimization algorithm chosen for POSTGRES, focusing on features specifically incorporated to handle extendibility in POSTGRES. The rest of the section will then indicate how this algorithm is used in optimizing nested-attribute queries. Algorithms are described in high-level detail with special attention given to the design rationale behind various features. Plans produced by these algorithms are also described to indicate how the query processor interprets optimizer plans.

3.1. Choice of Optimization Algorithm

In selecting an optimization algorithm to work with, there were two choices — query decomposition [WONG76] or exhaustive search. Query decomposition is a heuristic ''greedy'' algorithm that proceeds in a stepwise fashion. If a query has three or more variables, heuristics are first used to subdivide the query into two smaller subqueries. This process is applied recursively to any subqueries that contain at least three variables. For subqueries containing less than three variables, tuple substitution is used to process the join, while a component known as the one-variable query processor determines the path used to scan individual relations.

Once the first step of a query plan has been constructed using this decomposition algorithm, the step is executed. By doing this, the optimizer has information on the sizes of intermediate results, which can be used to its advantage in making subsequent decisions. Furthermore, the search space is reduced substantially because only a single path is considered. However, as a result, potentially good plans are ignored during early stages of the algorithm.

The SYSTEM-R designers took a dramatically different approach by essentially doing an exhaustive search of the plan space. All possible ways of scanning each relation appearing within a query are found. Using these paths, all plans for processing two-way joins are considered. Then, single relations are joined to form three-way joins, and from here, the algorithm iterates in a similar manner. The cost of executing each of these paths is estimated, and the cheapest is selected as the desired plan.

Although exhaustive search inevitably requires more planning time, good plans are not overlooked. This is especially important when optimizing complicated queries because for these queries the difference in the amount of processing required by two plans can be quite significant. Thus, the extra planning overhead is more than compensated by savings that result from executing a better plan. For simple queries, although the selected plan may not be significantly better than another, the extra overhead is likely to be inconsequential. For queries embedded within data fields, the extra overhead of enumerative planning is especially unimportant because these queries will be preexecuted in background mode and POSTGRES will cache the execution results as well as the compiled query plans. In these cases, the time spent optimizing will be amortized over several executions.

The SYSTEM-R optimizer only considers linear joins, e.g.,

((A join B) join C) join D

for a 4-way join. The optimizer could be improved to consider joins between pairs of composite relations, e.g.,

(A join B) join (C join D).

This would allow the optimizer to examine further plans, and on occasion, these plans may be significantly better than plans that only utilize linear joins.

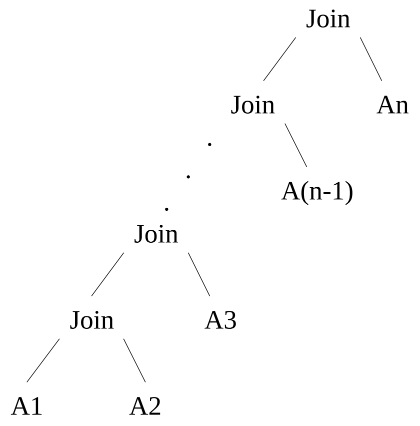
Another enhancement to the SYSTEM-R optimizer is to consider plans that will dynamically create indices on join attributes if they are not already available. If the cost of building the index is small compared to the savings that result from using the index in a join, such a strategy can be advantageous.

The POSTGRES optimizer does not consider these enhancements either. Although they would undoubtedly result in a better optimizer, the main goal of POSTGRES is to illustrate the feasibility of the novel features that it will support. Therefore, for simplicity, the POSTGRES optimizer will adhere fairly closely to the algorithm as described in [SELI79].

3.2. Pipelining of Tuples

Ignoring nested attributes for the moment, the query plan created by the optimizer is a tree of scan and join nodes. Relations are either scanned sequentially or via primary or secondary indices. Joins are processed using nested iteration, merge-sorts, or hash-joins. Each of these join strategies will be discussed further in section 3.3.3.

Every join involves two components. The component that is scanned first will be referred from hereon as the ‘‘outer join relation,’’ while the other component is the ‘‘inner join relation.’’ For a query containing *n* variables, the plan is constructed in such a way that the *n*-way composite join appears at the top of the tree. (See figure 3.1.) The left subtree is a plan for processing the outer join relation, and the right subtree corresponds to a plan for processing the inner join relation. Because the optimizer only considers linear joins (see section 3.1), the right subtree is always a scan node while the left subtree is a plan for an *(n-1)*-way join, or scan if *n* = 2. These characteristics apply recursively to the rest of the tree.



(((A1 join A2) join A3) join ... A(n-1)) join An

Figure 3.1
Plan tree for an *n*-way join

To process such a plan, the query processor walks through the tree starting at the root. If the node is a join, depending on the join strategy, calls are made on either the left or right subtrees to retrieve tuples from either the outer or inner join relations. If the node is a scan node, the appropriate relation is scanned using the selected strategy. When scanning a relation, restriction clauses specified on the relation are examined. Once a single tuple has been found satisfying these qualifications, the tuple is returned to the node that initiated the scan. If the higher node was a join node and this tuple originated from the left subtree of the join node, then a call is made on the right subtree. Otherwise, this tuple originated from the right subtree and thus can be joined with the tuple passed back by the left subtree. Provided all corresponding join clauses are satisfied, a composite tuple is formed. If the join clauses are not satisfied, calls are made on either the right or left subtrees until a qualifying composite tuple can be constructed. Once this composite tuple is formed, it is passed upward to the node that called this join node, and the process is repeated.

If tuples must be sorted or hashed prior to join processing (see figure 3.2), all tuples returned from a lower node must first be stored in a temporary relation. Once the lower node has passed all relevant tuples into the temporary, the sort or hash is performed. From here, the temporary relation is scanned like any other relation, and its tuples are also pipelined upward. In summary, calls are made on lower nodes in the tree when tuples are needed at higher nodes to continue processing, and tuples originating from scan nodes at the leaves of the plan tree are pipelined bottom-up to form composite tuples, which also are pipelined upward.

As an alternative to pipelining, the query executor could have processed nodes to completion, stored the intermediate subresults in temporary relations, and then passed groups of tuples upwards rather than a tuple at a time. This may be advantageous when there are many duplicate tuples in a subresult. The duplicates could be removed from the temporary, reducing the time required to process later joins. However, for simplicity, we chose to focus on a pipeline processing scheme for now. Implementation of temporaries will be reserved for future extensions.

3.3. Generating Possible Plans

The SYSTEM-R optimizer decides which plans should be generated based upon the types of indices defined on relations appearing in a query as well operators that also appear in the query. For example, suppose a B-tree index is defined on a relation, and a query contains the following restriction clause:

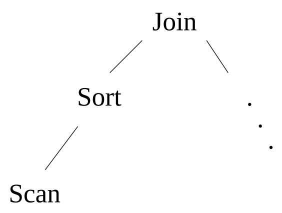


Figure 3.2
Sort node in a plan tree

relation.field OPR constant.

Using Selinger's notation, clauses of this form will be referred from hereon as ''sargable'' clauses [SELI79]. If *relation.field* within a sargable clause happens to match the key of the B-tree index and the restriction operator, *OPR*, is anything but $\neq$, then an index path should be considered because a B-tree provides efficient access when used with the following operators:

$=, <, \leq, >, \geq.$

The criteria for decisions like this can easily be incorporated into the SYSTEM-R optimization code because a conventional database only has a fixed number of operators and access methods; so there are only a fixed number of possibilities. Clearly the POSTGRES optimizer cannot use such a strategy due to POSTGRES's provision for extendibility. Therefore, we resorted to storing operator and access method characteristics in database system catalogs, and we coded the optimizer to access and use this information in generating possible plans. The rest of this subsection will discuss in greater detail the steps taken in creating access plans in order to focus on the type of information the optimizer and other relevant parts of the system will need in making decisions.

3.3.1. Transforming Queries Into Standard Form

Prior to optimizing a query, the parser must take the user's ascii request, parse it for valid syntax, perform semantic checks, and finally generate a tree representing information within the query. To make optimization simpler and to produce more efficient query plans, the parser must place the qualification portion of every query in conjunctive normal form. This entails pushing all ''or'' clauses to the innermost levels of a qualification using the following distributive rule:

a or (b and c) $\equiv$ (a or b) and (a or c).

The optimizer also requires that ''not's'' be pushed to the innermost levels using DeMorgan's law:

not (a and b) $\equiv$ not (a) or not (b)
not (a or b) $\equiv$ not (a) and not (b)

and if possible, removed from the query. For example, the qualification in figure 3.3 is equivalent to the qualification in figure 3.5, which is in conjunctive normal form with ‘‘not’s’’ removed.

Removing ‘‘not’s’’ from a qualification requires substituting operators with their respective negations. For example, ‘‘=’’ would be replaced by ‘‘≠,’’ while ‘‘AREAGT’’ would be replaced by ‘‘AREALE.’’ For the parser to make these substitutions, users must specify an operator’s corresponding negation, in addition to other information, when defining a new operator. The information is specified as follows:

define operator (**opname is** =, · · · , **negator is** ≠, · · ·)

and is stored in an operator catalog, accessible by the parser.

There are, however, problems associated with this requirement. First of all, this *forces* users to define operators corresponding to negators. In other words, having specified ‘‘AREANEQ’’ as a negator, it is also necessary to define an operator called ‘‘AREANEQ.’’ Although this definition is not difficult, since a negator is the logical opposite of an already defined operator, users may have no need for the negator, and therefore would rather not have defined the extraneous operator. Secondly, because every

---

not (r.f = 1) or (not (r.f2 > 1 or r.f2 < 3))

Figure 3.3
Qualification in non-standard form

(not (r.f = 1) or not (r.f2 > 1)) and (not (r.f = 1) or not (r.f2 < 3))

Figure 3.4
Equivalent qualification in conjunctive normal form

(r.f ≠ 1 or r.f2 ≤ 1) and (r.f ≠ 1 or r.f2 ≥ 3)

Figure 3.5
Equivalent qualification in conjunctive normal form with ‘‘not’s’’ removed

---

negator is also a user-defined operator, an operator's negator may be specified before the negation operator has been defined. In other words, depending on whether the operator ''AREAEQ'' or ''AREANEQ'' is defined first, the other operator will not have been defined yet when the negator of the first is specified. This means there is no guarantee that the specified negator is actually a valid operator; a user could specify the negator of ''AREAEQ'' to be ''foobar,'' or he may specify the correct negator, ''AREANEQ,'' but forget to define the *operator* ''AREANEQ.''

We have addressed both issues by implementing the optimizer so it allows ''not's'' to appear within qualifications. Therefore, if a user defines a negator that happens to be a nonexistent operator (e.g., ''foobar'') or doesn't specify one, the parser has no choice but to push the ''not's'' as far as possible into the qualification without actually removing them. (See figure 3.4.) The only problem with this is that the optimizer may not produce optimal plans when ''not's'' are left within clauses. For example, the following query:

(1) **retrieve** (foo.bar) **where not**(foo.f1 AREANEQ 1)

is equivalent to this query:

(2) **retrieve** (foo.bar) **where** foo.f1 AREAEQ 1

because **not**(AREANEQ) ≡ AREAEQ. If an area B-tree index is defined on the field ''f1,'' then the optimizer would definitely consider an index scan because the operator, ''AREAEQ,'' in query (2) can be used with an area B-tree. However, if a user had not specified the negation of ''AREANEQ,'' then the transformation from (1) to (2) would not have been possible, and the optimizer could not have considered an index scan. In this case, the index scan probably would have been the optimal path. Therefore, it is to the user's advantage to specify and define negators.

Another desirable transformation is that variables in qualifications appear on the left-hand side of an operator and constants on the right (e.g., *r.field OPR constant*). To make this transformation, operands must be reversed and operators must be replaced with their respective ''commutators,'' which again the user must specify. For example, the commutator of ''<'' is ''>,'' while the commutator of ''AREAEQ'' is also ‘‘AREAEQ.’’ The issues and solution discussed in the previous paragraphs in reference to negators also apply here, and again, it is to the user’s advantage to specify commutators. The reasoning behind this will be discussed further in section 3.3.2. Basically, it enables the optimizer to consider index paths it could not have considered had a variable appeared on the right hand side of an operator.

3.3.2. Index Scans

Once a query tree has been transformed as close as possible to standard form, it is passed to the optimizer. The first step the optimizer takes is to find all feasible paths for scanning each relation appearing in the query. Relations can always be scanned sequentially; therefore, a sequential scan path is always considered. Generally when sargable clauses (i.e., clauses of the form *relation.field OPR constant*) appear within queries, indices will restrict the amount of search required. Therefore, if a user has a primary index or secondary indices defined on a relation, all viable index paths are also considered.

For an index to be considered, its keys must match variables that appear within sargable restrictions. The optimizer also needs to insure the usability of the operator within the sargable clause with the index under consideration. For example, an area B-tree, whose records are sorted in ‘‘AREALT’’ order, can be used with sargable clauses for which *OPR* is:

AREAEQ, AREAGT, AREAGE, AREALT, or AREALE,

while a standard B-tree can be used with sargable clauses containing:

$=, >, \geq, <,$ or $\leq$.

To distinguish the two cases, POSTGRES introduces the notion of a ‘‘class.’’ Associated with every class is a particular access method and the set of operators that can be used with it. For example, the class ‘‘intops’’ refers to a standard B-tree. The operators associated with it are:

$=, >, \geq, <,$ and $\leq$.

The class ‘‘areaops’’ is an area B-tree. Therefore, the operators associated with it are:

AREAEQ, AREAGT, AREAGE, AREALT, and AREALE.

Moreover, another class ‘‘hashops’’ is a standard hash table that can only be used with ‘‘=’’. This information is stored as specified in table 3.1. To determine the usability of an index, whose key matches the variable within a sargable clause, table 3.1 is scanned using the class of the index and the operator within the sargable clause as a search key. If the pair is found, then the index can be used; otherwise, it cannot.

Determining index usability, therefore, involves matching operators that appear within sargable restriction clauses with the operators associated with an index’s class. By requiring that qualifications be transformed so variables are on the left-hand side and constants are on the right-hand side, the optimizer is insured that an operator appearing within a clause is semantically equivalent to the actual operator that appears in table 3.1. For example, suppose the operator ‘‘foo’’ is usable with index ‘‘foobar,’’ but its commutator ‘‘bar’’ is not. If we have the following restriction:

10 foo relation.field,

the optimizer should not consider using index ‘‘foobar’’ because the above restriction is equivalent to:

relation.field bar 10.

AMOP	access-method	class	operator	· · · other information · · ·
	B-tree	intops	=	
	B-tree	intops	<	
	B-tree	intops	≤	
	B-tree	intops	>	
	B-tree	intops	≥	
	B-tree	areaops	AREAEQ	
	B-tree	areaops	AREALT	
	B-tree	areaops	AREALE	
	B-tree	areaops	AREAGT	
	B-tree	areaops	AREAGE	
	hash	hashops	=	

Table 3.1
Index and operator classes

Currently, only a single clause can be used with an index defined on a single key. For example, if the following two clauses are contained in a query:

r.foo > 100 **and** r.foo < 150,

and a B-tree index is defined on the field ‘‘foo,’’ then only one of the two clauses can be used with the index. In other words, either *all* values foo > 100 are located using the index, or *all* values < 150, but *not* both. Had both clauses been used with the index, only those values between 100 and 150 would have to be examined. This has the possibility of reducing the scope of an index scan substantially. However, such an optimization is being reserved for future extensions of POSTGRES. It will not only require a a redefinition of the access method interface but also extra information from the user establishing which operator should be used at the low end of a scan (e.g., >) and which corresponds to the high end (e.g., <). The first requirement is necessary because currently the access method interface is only designed to work with one clause per index key.

In the case where an index is defined on multiple keys, the ideas discussed above simply generalize. In other words, for every key of an index, there must be a corresponding sargable restriction clause, and in addition to all operators in these clauses being usable by the index, they all must be identical. A more flexible approach would have allowed partial matching of keys and multiple operators to be used in a single scan. This would have required that POSTGRES store further information about operators and access methods in system catalogs. Namely, for every access method, the optimizer would need an indication as to whether partial key matching is possible, and it also would need some way of establishing that the following clause:

r.field1 = 1 **and** r.field2 > 5

can be used with a B-tree index defined on ‘‘field1’’ and ‘‘field2,’’ but the following cannot:

r.field1 > 1 **and** r.field2 < 10.

The benefits of this extra information is not significant enough to justify the extra complexity that would be required when defining new operators and access methods; therefore, POSTGRES does not implement these features.

One optimization that the POSTGRES planner does support is use of multiple indices to process ‘‘or’’ clauses. Normally, it would not be possible to use an index scan with the following clause:

r.field = 1 **or** r.field = 2

because there are two key values, 1 and 2. However, it is possible to use an index scan keyed on 1 followed by another index scan keyed on 2. Since two index scans may be much less expensive than a single sequential scan, the optimizer will consider using multiple index scans.

In addition to restricting the scope of a search, index paths are also considered for another reason. During query processing, it may be necessary to sort an intermediate result prior to a merge-sort join (see figure 3.2), or the user may specify that the results of a retrieve query be sorted on certain fields. However, these sorts do not have to performed explicitly at all times. Some access methods maintain their tuples sorted on the keys used to define the structure. Thus, scanning a relation via such an index may yield tuples sorted in a desired order. For example, a standard B-tree stores its tuples sorted either in ascending (<) or descending (>) order, while an area B-tree maintains its tuples sorted in either ‘‘AREALT’’ or ‘‘AREAGT’’ order.

To make use of an index with this sort characteristic, the index keys must either match variables within join clauses, which correspond to relations that will later be merge-sorted, or attribute fields on which a query’s resulting tuples will be sorted. To determine whether an index’s implicit sort order is that which is needed, POSTGRES requires that users specify an access method’s sort order (if it exists) when defining a new access method. If the implicit ordering matches a desired ordering and the keys are usable, a path that takes advantage of the index will be considered. The next two subsections will elaborate on further uses of this sort information.

3.3.3. Join Paths

Once all feasible paths have been found for scanning single relations, paths are found for joining relations. Joins are first considered between every two relations for which there exists a corresponding join clause. For example, for the following query:

retrieve (A.a, B.b, C.c) **where** A.d = B.e,

during the first level of join processing, the only pairs considered are:

A join B
B join A

All feasible paths are found for processing joins between these relation pairs. Having done this, all paths are then found for processing 3-way joins, using available 2-way join paths for the outer path and relation scan paths for the inner path. Again, the optimizer only considers those join pairs for which there is a corresponding join clause. If this heuristic results in no further relations being joined, all remaining possibilities are considered. For the above query, at the second level of join processing, no relations should be joined according to the heuristic. Therefore, the remaining possibilities are:

(A join B) join C
(B join A) join C

From here, these steps are repeated until no further join levels need to be processed.

All possible join paths are generated for every join pair considered. The simplest join strategy is nested iteration. In a nested iteration join, the inner join relation is scanned once for every tuple found in the outer join relation. All available paths on the outer join relation are possibilities for the outer path. On the other hand, since the inner join path is independent of the outer in a nested iteration join, only the least expensive path for the inner join relation is a possibility for the inner path.

Nested iteration is simple, but it can be a time-consuming join strategy, especially if the inner join relation is not indexed on join variables. A join strategy that is much more attractive in these situations is merge-sort. A merge-sort join can be used to process a join between *relation1* and *relation2*, provided there is a merge join clause of the form:

relation1.field1 OPR relation2.field2.

During the first phase of a merge-sort, each relation is sorted on appropriate join attributes. During the second phase, the merge phase, the two relations are merged together, taking advantage of the fact that both relations are ordered on join attributes.

For a merge-sort join to be advantageous, the operator within a merge join clause must be ‘‘similar to’’ an equality operator, e.g. ‘‘AREAEQ’’. Therefore, in the most ideal situation, when both join relations contain unique values in the merge join fields, the merge phase will only require a sequential scan of both sorted relations. So when defining new operators, POSTGRES requires that users indicate whether an operator is ‘‘mergesortable’’ by specifying the operator that must be used to sort the two join relations prior to the merge. For example, ‘‘=’’ is mergesortable, provided the sort is made in ‘‘<’’ order, while ‘‘AREAEQ’’ is also mergesortable, provided the sort is in ‘‘AREALT’’ order. Therefore, if a join clause in the form specified above contains a mergesortable operator, then a merge-sort join path between the two relations in the clause is considered in addition to paths that process the join using nested iteration.

As alluded to earlier, relations may not always have to be explicitly sorted prior to a merge. If the keys of an index match join attributes and the index’s implicit sort order matches the sort operator required for the merge, the relation need not be sorted; unless, of course, it is cheaper to scan a relation into a temporary via its least expensive path, sort the temporary, and then read the sort result.

A second join strategy that may be useful when indices are not available is hash-join. To process a join using this strategy, the inner join relation is first hashed on its join attributes. Then, the outer relation is scanned. For every tuple found, values within outer join attributes are used as hash keys to locate tuples in the inner join relation. Thus, to use such a strategy, the join operator also must be ‘‘similar to’’ an equality operator. Users will indicate this by simply specifying whether an operator is ‘‘hashjoinable’’ when defining a new operator.

3.3.4. Pruning The Plan Space

In generating possible plans for a query, many paths are considered. In fact, the plan space is exponential because plans at lower levels are used in creating plans at higher levels. Furthermore, when indices and multiple join strategies can be used, there are a number of ways of processing scans and joins on identical relations. Moreover, some of these paths may be redundant.

Two paths are redundant if they scan identical relations, and their resulting tuples are sorted on identical fields. The latter is determined by making use of index sort information. For example, suppose the outer relation of a join is scanned using an index that sorts its tuples in ascending order, and the relation is joined with another relation using nested iteration. This join path is equivalent to another path where the outer relation is explicitly sorted into ascending order and merge-sorted with the same inner join relation. Although these two paths are different, they will yield identical results because both outer join relations are sorted in identical order, and joins preserve the sort order of the outer relation. Therefore, after generating plans for each level of joins, if two redundant join plans are found, the optimizer can eliminate the more expensive of the two, thereby reducing the size of the plan space.

3.4. Estimating Costs and Sizes

To prune the plan space as well as determine the optimal plan, the optimizer must estimate the cost of executing every plan it generates. In the SYSTEM-R optimizer, both CPU and I/O time are accounted for in estimating costs. Every cost factor is of the form:

$$\text{cost} = P + W * T,$$

where P is the number of pages examined at runtime by a plan and T is the number of tuples examined. P reflects I/O cost, T reflects CPU cost, and W is a weighting factor that indicates the relative importance of I/O to CPU in terms of processing cost. Thus, for a sequential scan of a relation, since every page and tuple must be examined, P equals the number of pages in the relation and T equals the number of tuples. For a secondary index scan, the number of pages and tuples touched also depends on the number of pages and tuples in the index relation because index pages and tuples must be read first to determine where to scan in the main relation.

For every index, the number of pages and tuples touched is also determined by the fraction of tuples in a relation that one would expect to satisfy restriction clauses specified on the relation. This fractional quantity is called a ‘‘selectivity.’’ Selectivities are functions of a variety of parameters, including the operator in the restriction clause, the restriction constant, the number of records in an index, and the maximum and minimum values of entries stored in an attribute. In SYSTEM-R, every possible selectivity factor is hardwired into the cost estimation code. See table 3.2 for a sampling of selectivity factors.

Cost estimation formulas for all join strategies are functions of the sizes, in pages and tuples, of the outer and inner join relations. To estimate the size of either an outer or inner join relation, the optimizer simply multiplies the original size of the relation by the selectivity of every restriction clause applicable to the relation. If the clause can be used with an index, the selectivity is computed as described earlier. If it cannot, SYSTEM-R resorts to an ‘‘else case,’’ associating constants with these factors. The SYSTEM-R designers justify this simplification by stating that if an index is not defined on a relation, this implies that the relation is small; so if the selectivity is not accurate, the difference is insignificant. If the outer join relation is a composite relation, the desired selectivity is that of a join operation. Such a selectivity indicates the fraction from among the cross product of an outer and inner join relation one would expect to satisfy a join clause. Again, SYSTEM-R hardwires this information into the optimizer

qualification	selectivity factor
r.field = value	1/(number of tuples in the index relation defined on r.field)
r.field > value	((high value of r.field) - value) / ((high value of r.field) - (low value of r.field))
r.field < value	(value - (low value of r.field))/ ((high value of r.field) - (low value of on r.field))

Table 3.2
Examples of selectivity factors for SYSTEM-R

code. For a summary of the different cost formulas required, see table 3.3.

The POSTGRES optimizer uses this same basic idea in estimating costs. As in SYSTEM-R, the system stores statistical information that cost formulas depend upon in database system catalogs. However, POSTGRES takes the novel approach of updating certain statistics, like the number of pages and tuples in relations, using demons, which execute in background mode. These demons are implemented

Cost of Scans		
	P	T
Sequential Scan	NumPages	NumTuples
Primary Index Scan	NumPages*F	NumTuples*F
Secondary Index Scan	NumPages*F + ITuples*F	ITuples*F + NumTuples*F

where

NumPages = the number of pages in a relation
NumTuples = the number of tuples in a relation
ITuples = the number of tuples in an index relation
F = combined selectivity factor of applicable restriction clauses

Cost of Joins	
Nested Iteration	$C_{outer}+N_{outer}*C_{inner}$
Merge-sort	$C_{outer}+C_{sortouter}+C_{inner}+C_{sortinner}$
Hash-join	$C_{outer}+C_{createhash}+N_{outer}*C_{hash}$

where

C_{outer} = the cost of scanning the outer join relation
C_{inner} = the cost of scanning the inner join relation
$C_{sortouter}$ = the cost of sorting the outer join relation into a temporary †
$C_{sortinner}$ = the cost of sorting the inner join relation into a temporary †
$C_{createhash}$ = the cost of hashing the inner join relation into a temporary
C_{hash} = the cost of a single hash
N_{outer} = the size of the outer join relation

† equals 0 if sort is not required

Table 3.3
Summary of cost formulas

using triggers, a feature POSTGRES supports. Another new idea is that other statistics, e.g., the high and low values of an attribute, are stored in the form of queries embedded within data fields. These queries will retrieve appropriate information from elsewhere in the database, and the system will cache the result so the optimizer need not repeatedly execute the same query. These two provisions alleviate problems other systems encountered when updating database statistics. INGRES updated statistics immediately, resulting in loss of concurrency because it blocked other users access to information in system catalogs. SYSTEM-R chose not to update statistics at runtime, instead requiring that a data base administrator run a special command to explicitly do the update. This, however, meant that statistics were not always up-to-date, possibly yielding incorrect cost estimations.

To account for all possible selectivities, the approach of storing information in catalogs is used again. One difference between selectivity information and other operator and access method information seen thus far is that selectivities are parameter dependent, as illustrated in table 3.2. An earlier approach suggested storing simple formulas within data fields and writing a parser to interpret the formula, substituting appropriate values for a fixed set of possible parameters [STON85a]. This is fairly straightforward, but not very flexible. Instead, our optimizer capitalizes on an inherent feature of POSTGRES. As already mentioned, POSTGRES supports embedding of queries within data fields. Generalizing on this idea, POSTGRES also allows users to define arbitrary procedures written in general purpose programming languages, like C and LISP, which can then be registered into the system and stored within data fields [STON86c]. Therefore, in POSTGRES every selectivity formula is expressed using a parameterized procedure. Each procedure accepts as its arguments the relations, attributes, and constants appearing within restrictions and the index identifier and number of index keys, if an index is used. The routine then retrieves necessary information from elsewhere in the database and returns a computed selectivity.

As discussed in reference to SYSTEM-R selectivities, selectivities come in three flavors. Therefore, there will be a selectivity routine associated with every feasible operator-class pair shown in table 3.1, and every operator will have two selectivitiy routines associated with it — one for ordinary restrictions, which are not used with index scans, and the other for joins. Each of these procedures is stored within appropriate tuple entries.

Thus, by executing the appropriate procedure with the appropriate parameters, selectivity computation is flexible and simple. A procedure written in pseudo C code that computes the selectivity of the operator ‘‘>’’ for a B-tree index is shown in figure 3.6.

3.5. Nested-attribute Queries

The last several subsections have described optimization of simple queries, i.e. those without nested attributes. Figure 3.7 summarizes information the optimizer uses in generating possible query plans.

From here on, the module implementing the algorithms just described will be called the ‘‘subplanner,’’ while the entire optimizer will be labeled the ‘‘planner’’. To create access plans for queries containing nested attributes, the planner simply applies the subplanner algorithm once for each nesting level of attributes in a query. In other words, for any query, the number of times the subplanner is called is equal to the maximum nesting of attributes in the query. Once all subplanner calls have completed, the planner then builds a final plan that indicates how these subpieces fit together. Thus, given a query, the planner first modifies it to consider only top level attributes. This new query is passed to the subplanner to create a *subplan*. The planner then modifies the original query to consider only nested attributes. This is recursively processed by the planner to create a *plan*, and the resulting access plan simply indicates which attributes from *subplan* and *plan* should be returned to the user.

An example will illustrate these ideas more clearly. Suppose we have the following relation:

EMP (name, dept, hobbies),

where hobbies contains POSTQUEL queries to retrieve information about the different hobbies each employee participates in. One of these relations may be:

SOFTBALL (empname, position, batting-history),

where batting-history contains a POSTQUEL query retrieving information about an employee’s past batting averages from the relation:

```
/*
 *      Procedure to compute the selectivity of the ‘‘>’’ operator
 *      when it is used with a B-tree index defined on integer fields.
 */

float
greater_btree (opid, relid, attnos, values, flags, indexid, nkeys)
int     opid;           /* contains unique id of operator ‘‘>’’ */
int     relid;
int     attnos[];
int     values[];
int     flags[];        /* equals 1 if clause is of the form ‘var > constant,’
                         * else clause is of the form ‘constant > var’
                         */
int     indexid;        /* parameter isn’t used by this particular routine */
int     nkeys;
{
        int     i;
        int     high;
        int     low;
        float   s;

        s = 1.0;
        for (i = 0; i < nkeys; ++i) {
                high = retrieve high value of attribute ‘attnos[i]’ in relation ‘relid’;
                low = retrieve low value of attribute ‘attnos[i]’ in relation ‘relid’;
                /*
                 * the selectivity of multiple clauses is the product of the
                 * selectivity of each individual clause
                 */
                if (flags[i] == 1)
                        s = s * (high - values[i]) / (high - low);
                else
                        s = s * (values[i] - low) / (high - low);
        }
        return(s);
}
```

Figure 3.6

Code to compute selectivity

OPER	operator	negator	commutator	msortop	hash	selectivity	join-selec
	=	≠	=	<	yes	*procedures*	*procedures*
	<	≥	>	no	no	*to compute*	*to compute*
	≤	>	≥	no	no	*selectivity*	*selectivity*
	>	≤	<	no	no	*of*	*of*
	≥	<	≤	no	no	*1-variable*	*join*
	AREAEQ	AREANEQ	AREAEQ	AREALT	yes	*clauses*	*clauses*
	AREALT	AREAGE	AREAGT	no	no	*containing*	*containing*
	AREALE	AREAGT	AREAGE	no	no	*‘‘operator’’*	*‘‘operator’’*
	AREAGT	AREALE	AREALT	no	no		
	AREAGE	AREALT	AREALE	no	no		

AMOP	access-method	class	operator	selectivity	strategy †
	B-tree	intops	=	*procedures*	=
	B-tree	intops	<	*to compute*	<
	B-tree	intops	≤	*selectivity*	≤
	B-tree	intops	>	*of clauses*	>
	B-tree	intops	≥	*containing*	≥
	B-tree	areaops	AREAEQ	*‘‘operator’’*	=
	B-tree	areaops	AREALT	*when used*	<
	B-tree	areaops	AREALE	*with index*	≤
	B-tree	areaops	AREAGT	*‘‘class’’*	>
	B-tree	areaops	AREAGE		≥
	hash	hashops	=		=
	B-tree	intops	<		sort
	B-tree	areaops	AREALT		sort

† used in determining which operator corresponds to each generic operation (e.g., sorting)

Figure 3.7
Summary of optimizer information stored in system catalogs

BATTING-HIST (empname, year, avg).

Given the following query:

Q1: **retrieve** (EMP.name, EMP.hobbies.batting-history.avg) **where**
EMP.hobbies.batting-history.year = ‘‘1986’’ **and**
EMP.hobbies.position = ‘‘catcher’’ **and**
EMP.dept = DEPT.dname **and**
DEPT.floor = 1,

which finds the current batting average of employees who are catchers and work on the first floor, the planner will first consider the top level of attributes by passing the following query to subplanner:

S1: **retrieve** (EMP.name, EMP.hobbies) **where**
EMP.dept = DEPT.dname **and**
DEPT.floor = 1.

After the subplanner has optimized the above query, the planner then only considers attributes beyond the top level nesting:

Q2: **retrieve** (EMP.hobbies.batting-history.avg) **where**
EMP.hobbies.batting-history.year = ‘‘1986’’ **and**
EMP.hobbies.position = ‘‘catcher’’

and passes this query to itself recursively. This process is repeated, and in the process, the subplanner optimizes the following two queries:

S2: **retrieve** (EMP.hobbies.batting-history) **where**
EMP.hobbies.position = ‘‘catcher’’

S3: **retrieve** (EMP.hobbies.batting-history.avg) **where**
EMP.hobbies.batting-history.year =‘‘1986’’

Since the maximum attribute nesting is three, recursion ends. The final plan is constructed as shown in figure 3.8, where *Pn* is a subplan for executing subquery *Sn*. *R2* is a node corresponding to *Q2* that indicates that EMP.hobbies.batting-history.avg should be retrieved from *P3*, and *R1* corresponds to *Q1*, indicating that EMP.name should be retrieved from *P1* and EMP.hobbies.batting-history.avg originates from *R2*.

To process this plan, the query executor first executes *P1*. As soon as execution of *P1* returns a single tuple, *T*, EMP.hobbies in that tuple is materialized into a temporary relation, if that has not already been done by POSTGRES’s facility for preexecuting embedded queries. The executor then processes the subtree whose root is *R2* in a recursive manner for that instance of EMP.hobbies. Each EMP.hobbies.batting-history.avg value returned from this execution is combined with EMP.name from *T* to create a result tuple that is passed to the user. When execution of *R2* has completed, *P1* is reexecuted to retrieve another EMP.hobbies value that is used to process another instance of the *R2* subtree. This is repeated until all tuples from *P1* have been found. The subtree *R2* is processed similarly. *P2* is processed first, and *P3* is processed once for each instance of a qualifying *P2* tuple.

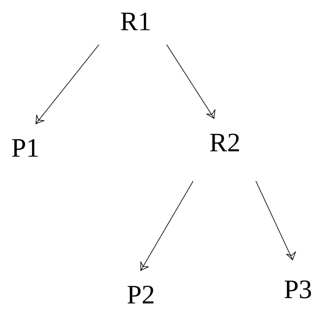


Figure 3.8
Structure of a query plan

Processing nested-attribute queries in this way is attractive because it only requires a very clean extension of the basic optimization algorithm, and nested attributes are transparent to the subplanner. As a result, the code for query processing is also simpler. Furthermore, entire query plans are generated a priori, eliminating the need to optimize subqueries at runtime.

Optimizing nested-attribute queries a priori, however, does suffer because of its simplicity. In generating plans for higher nesting levels, the contents of relations that will be materialized from embedded queries are not known in advance. Therefore, the optimizer does not know the sizes of these relations or the distribution of values within nested attributes; this information or some estimate is normally required in computing subplan costs. To work around this problem, the optimizer simply assumes that materialized relations are all of the same size, and if a nested attribute appears within a clause, rather than return a parameter-dependent value for the selectivity, the selectivity code instead returns a constant that reflects the relative selectivities of various operators. For example, ‘‘=’’ is more selective than ‘‘>.’’ So a clause containing ‘‘=’’ may have a selectivity of $\frac{1}{10}$, while a clause with ‘‘>’’ has a selectivity of $\frac{1}{4}$. Although this may be an oversimplification, usually relations materialized from embedded queries will be small. So if the resulting plan is not the overall best choice in actuality, the chosen plan will not be a bad plan.

As an alternative, pieces of nested-attribute queries can be optimized at runtime. This can be advantageous not only because relation sizes and attribute distributions are known, but also because it enables the optimizer to consider special paths. For example, it may be possible to process an embedded query using a B-tree index that sorts its records in ascending order. If the tuples materialized from this query are later merge-sorted using an ascending sort, due to the available index path, the query processor need not explicitly sort the relation. A runtime optimizer would be able to note this.

Although more intelligent query plans are generated, there is a great deal of planning overhead associated with runtime optimization. For every tuple generated by *P1*, a subquery of the form:

> **retrieve** (TEMP.batting-history.avg) **where**
> TEMP.batting-history.year = ‘‘1986’’ **and**
> TEMP.position = ‘‘catcher’’

must be optimized, where TEMP is the relation materialized from EMP.hobbies. Subsequently, for every tuple generated by the above query, the following query:

> **retrieve** (TEMP′.avg) **where**
> TEMP′.year = ‘‘1986’’

must also be optimized, where TEMP′ is materialized from TEMP.batting-history. Due to this extra overhead, the efficiency of runtime optimization is questionable.

3.6. Query Plans

The plan created by the optimizer is a tree of nodes. Each node corresponds to some scan, join, sort, or hash, or creation of a subresult. Scan and join nodes contain information indicating which attributes should be retrieved, which qualifications must be satisfied, and any other information relevant to the particular type of scan or join. Sort and hash nodes indicate which attributes should be placed into a temporary and the operator used to perform the sort or hash. A subresult node interconnects subplans and plans, indicating which attributes should be retrieved and from where. As an optimization, the topmost result node contains constant qualifications, i.e. those without variables, so these clauses can be examined prior to any other processing.

A possible (not necessarily the optimal) plan for the query introduced in the previous subsection is shown in figure 3.9. There are a few things about the tree that should be elaborated on to avoid

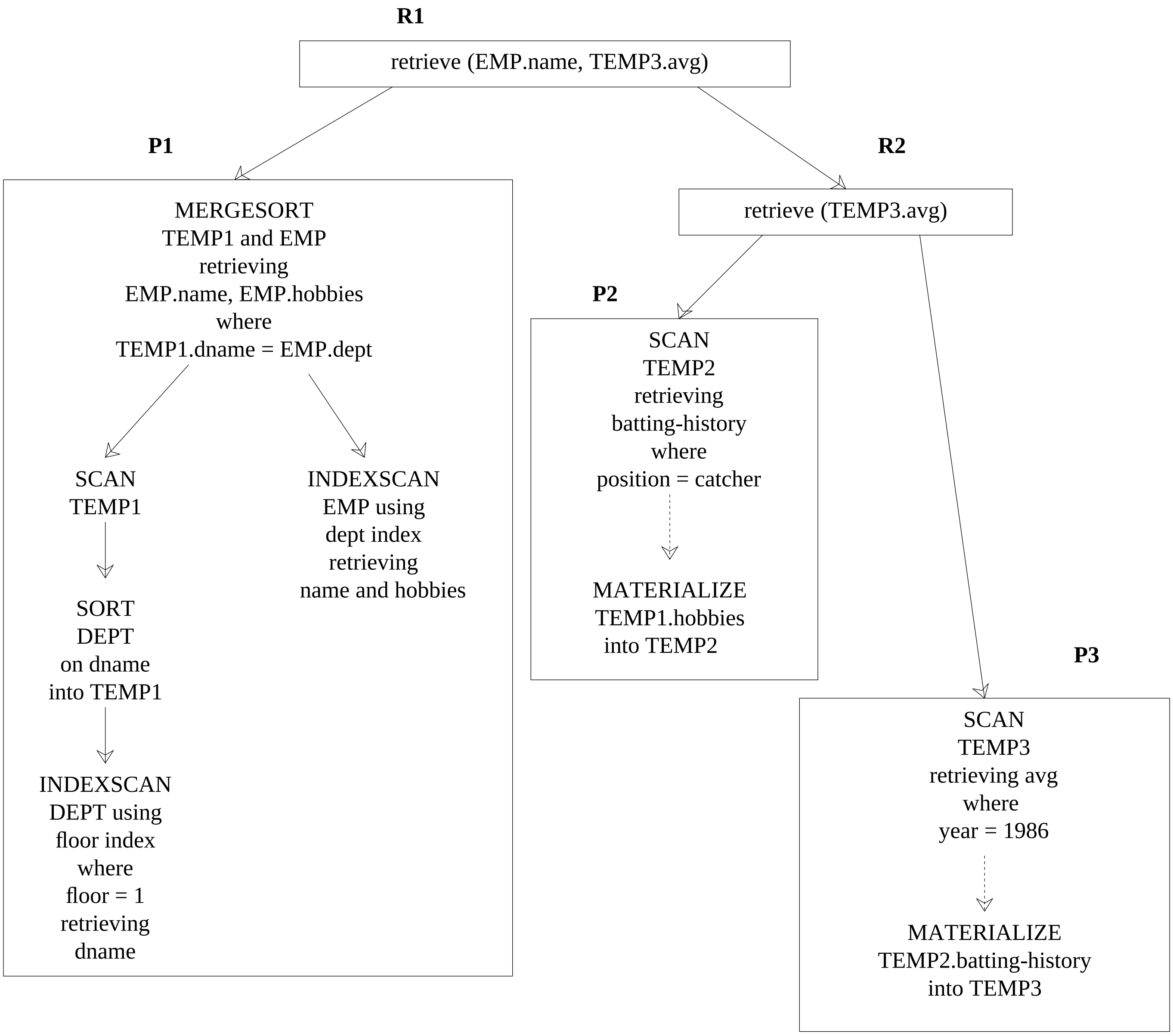


Figure 3.9
Sample query plan tree

misconceptions. First of all, the materialization steps shown are not executed at runtime if the necessary embedded queries have already been preexecuted, and their results remain valid. Furthermore, as will become apparent later, these materialization steps are actually implicit in the plan tree.

From the figure, it would appear that throughout the plan tree, relation entries are explicitly identified using relation and attribute names. This would imply that the query processor would have to match identifiers to locate values that originate from nodes elsewhere in the tree. For example, references to TEMP1 and EMP attributes in the mergesort node would be found by searching for an appropriate identifier within tuples that originate from the two nodes below the mergesort node. This, however, is not the case. Explicit references are shown only for readability. Rather, an attribute is identified by its relative position within a specified tuple. By using this relative value and a descriptor associated with tuples returned by each node, the query processor can locate a desired attribute instantaneously.

The names of temporaries associated with materialized relations, again, are shown only for readability. In the actual plan tree, these relations are identified by the attribute containing the queries that will be used to build the temporary. For example, TEMP2 would be identified by the relative attribute identifier of TEMP1.hobbies, while TEMP3 would be identified by TEMP2.batting-history. By examining the contents of these referenced attribute entries, the query executor can determine whether materialization is necessary. Thus, the materialization step is implicit in these attribute references.

For temporaries associated with sorts and hashes, the name of the temporary shown in the tree serves merely as an identifier. It is left to the runtime executor to create the actual name.

4. IMPLEMENTATION ISSUES

This section does not attempt to describe in detail the actual implementation of the POSTGRES optimizer. Rather, it focuses on important decisions made in building the optimizer.

4.1. Choice of Language

Query optimization requires a great deal of element manipulation. The optimizer must separate, modify, and regroup elements within a query's target list and qualification to create new components for individual nodes in the plan tree. A programming language like LISP is well-suited for this type of processing because the language contains functions and data structures specifically designed for object manipulation tasks. Consequently, for ease of implementation, we chose to write the POSTGRES query optimizer in LISP.

Franz LISP, Opus 42 [FRAN85] was the selected LISP dialect. It was chosen because it was readily available and also because it supports a foreign function interface. A foreign function interface allows LISP code to call routines written in other programming languages. This feature is of utmost importance because the optimizer must call C routines to access information from database system catalogs, given that the access method code in POSTGRES is being written in C.

Franz LISP, Opus 42 is also fairly compatible with CommonLISP [STEE84], an emerging standard LISP dialect. Therefore, in the future, if translation from Franz to CommonLISP is necessary, this will require minimal effort.

In general, compiled LISP code executes less efficiently than compiled C code. Therefore, an optimizer written in LISP will execute more slowly than an optimizer written in C. This, however, is not a problem. As discussed in section 3.1, POSTGRES compiles query plans and caches, for later use, plans and tuples resulting from query preexecution. Because of these two features, a single query plan produced by the optimizer may be used several times. As a result, the cost of optimization is amortized over several executions. This significantly reduces a query's planning cost, yielding a figure that is minimal relative to the overall cost of execution. Therefore, in terms of optimizer efficiency, the choice of language is not a major concern.

4.2. Representing Query Plans in LISP

In general, the cost of query processing constitutes the most significant portion of a query's execution cost. Therefore, the query processor must execute as cost efficiently as possible. To meet this goal, every node in the plan tree is constructed using one-dimensional arrays. These are known as ''vectors'' in LISP. Each element within a vector corresponds to some property of a node. By indexing appropriate vector entries, all properties can be accessed in constant time.

Among the properties within each plan node are the left and right subtrees of the node, target lists, and qualifications. The left and right subtrees either point to another plan node or nothing (**nil** in LISP). The target list and qualification entries respectively point to a list of individual target list elements and a list of restriction clauses. Lists are used to represent these structures because both sets of items are variable length, and random access to individual entries within these lists is not required. Each target list item consists of two items, also grouped together in a list. The first item in the list is a ''resdom'' node. It contains information about its corresponding target list entry — its type, destination, and if relevant, sort or hash information. Each resdom node is implemented using a vector. The second element, an *expr*, is an arbitrary arithmetic expression consisting of variables, constants, parameters, functions, and operators. Each of these subcomponents is also a vector, and these vectors are linked together in a list if they represent the arguments to a particular operation or function. A restriction clause is a boolean *expr*; therefore the preceding description applies to qualifications as well.

In addition, every plan node contains an empty slot that the query processor uses to store runtime-specific query information. Figure 4.1 shows the internal representation of a query plan that accesses two attributes and processes a single join clause using nested iteration.

Constructs analogous to records in Pascal and ''structs'' in C are used to build the different vector types associated with each node type. These constructs are called ''defstructs'' in LISP. With defstructs, LISP programmers can combine primitive data types to create structured items. These new data structures, in turn, can be combined and nested to create even more complex structures. After defining a defs-

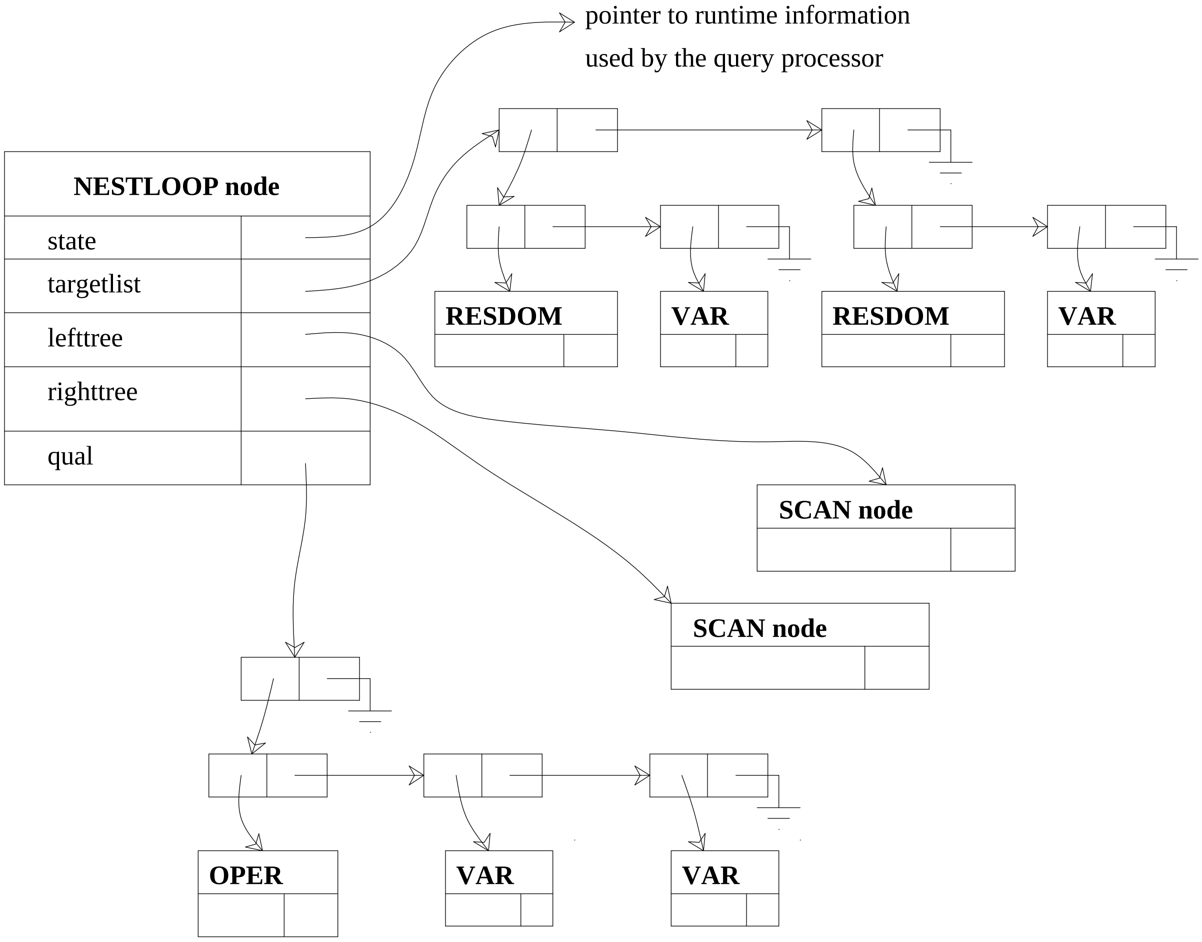


Figure 4.1
Internal representation of a nested iteration join node

truct, LISP automatically provides a set of routines to dynamically create objects of these structured types and to access information within a structure. As a result, although a vector is the underlying data type used to implement defstructs, users can access node properties by specifying field names, as opposed to indexing vector entries. Figure 4.2 shows a Franz LISP defstruct definition and associated routines for a nestloop node.

```
;       Plan information common to all plan nodes

(defstruct (plannode
            (:conc-name get_))
            (state nil)
            targetlist
            (lefttree nil)
            (righttree nil))
                  ; initialize lefttree, righttree, and state to nil

;       Nestloop node

(defstruct (nestloop
            (:include plannode)
                  ; node contains defstruct defined above
            (:conc-name get_)
            (:constructor make_nestloop (targetlist qual lefttree righttree)))
            (nodetype ''NESTLOOP'')
            qual)

;
;       LISP routines provided as a result of the above definitions:
;
;       Routines to retrieve property fields:

(get_state node)
(get_targetlist node)
(get_lefttree node)
(get_righttree node)
(get_nodetype node)
(get_qual node)

;       Routine to construct a nestloop node:

(make_nestloop targetlist qual lefttree righttree)
```

Figure 4.2
Sample defstruct definition

4.3. Internal Data Structures

In the process of creating possible query plans, the optimizer generates a great deal of information. To keep track of all this information, a more flexible structure of LISP is used — property lists. Every

LISP object may have associated with it a list of characteristics, set by the user, called a property list. A major advantage of property lists is that one does not have to preallocate space for property slots, as required for defstructs. As a result, at any given time, every object may have an arbitrary number of properties associated with it.

Property lists are implemented using linked lists. Thus, access to information within a property list requires a linear search. The inefficiency of linear search is not a problem here because generally, the optimizer does not store any more than four or five items within a single property list, and as indicated in section 4.1, efficiency is not a primary consideration in this part of the system. Therefore, because property lists are simpler to work with, they are used extensively within the optimizer.

4.4. LISP-C Interface

The foreign function interface in LISP is fairly easy to work with, provided a set of stringent rules are followed. For example, the standard way to pass structured information (e.g. a character string) from a C function is to return a pointer to the object. From here, the user can manipulate information within the object by referencing the pointer. This, however, will not work when LISP calls C because LISP cannot manipulate objects that C has allocated. It presents problems for the LISP garbage collector.

To work around this, C can return structured information by loading the data into variables that LISP has passed as parameters. Space for these return variables must be allocated by LISP prior to the C call. This is straightforward provided LISP knows the size of the returning object and can set aside a sufficient amount of memory. However, this is not always the case because tuples returned by C access method routines are variable length.

Fortunately, the optimizer never requires the contents of an entire tuple; on all occasions, it only needs a fixed set of attributes from within a single tuple. Therefore, rather than attempt to directly manipulate arbitrary tuples returned by access method routines, a layer written in C was built between the optimizer and the access method code. When the optimizer needs information from system catalogs, it calls some routine within this layer, which then calls access method routines to retrieve tuples. Desired information within these tuples are either returned explicitly as integers and floats, or they are passed back within variables allocated by LISP.

As an example, the optimizer may call a C routine, within the layer, called ‘‘retrieve_index’’ to retrieve information about a secondary index. In calling the routine, LISP passes a pointer to an integer array ‘‘indexinfo.’’ ‘‘Retrieve_index’’ then calls the access method routine ‘‘getnext’’ until an appropriate tuple from the index catalog has been located. The index identifier, the number of pages in the index, and any other relevant information are extracted from the tuple and passed back to the optimizer in the array ‘‘indexinfo.’’ Consequently, variable length tuples are handled solely by C, resulting in a cleaner and simpler LISP-C interface.

4.5. An Evaluation of Using LISP

Overall, writing the POSTGRES optimizer required about 6500 lines of LISP code and another 700 lines of C code. Having written the optimizer, using LISP was an excellent choice. There was a close match between our processing needs and the constructs and functions LISP provides. As a result, the programming effort was simplified. Had we used a language like C, we would have had to explicitly implement structures and routines equivalent to those LISP provides.

While writing the optimizer, it was also evident that other features of LISP were instrumental in simplifying code development. For instance, LISP allows you to either interpret or compile code written in the language. Naturally, compiled code is used in POSTGRES, but in developing the optimizer, the interpretive option was used. This significantly reduced development time because debugging was simpler and compilation time was eliminated.

LISP also supports dynamic allocation and implicit recollection of free space. The latter is implemented using garbage collection. As a result of these two properties, the optimizer can easily create objects of any type when needed, and LISP automatically handles memory management issues.

Last of all, LISP is a weakly typed language and because no single type is associated with variables in weakly typed languages, union types were implicit and variable declarations were unnecessary. This further resulted in simpler data structure definitions because declaration of field types was also unnecessary, as shown in figure 4.2. Another advantage of weakly typed languages is the absence of strict type checking. As a result, there is a certain degree of independence between the parameters a routine accepts and those that are actually passed. For example, if a routine accepts an identifier as a parameter but does not manipulate its actual value, then whether the identifier is an integer or string is not significant; choosing one or the other will not affect code within the routine. In many situations, this characteristic allowed us to make changes without modifying other relevant pieces of code. Changes could be made much more quickly as a result. So to briefly summarize, LISP was a simpler and much more flexible language to work with.

5. PERFORMANCE OF THE POSTGRES OPTIMIZER

This section describes how we went about validating the POSTGRES optimizer. It also presents and discusses our results.

5.1. Testing The Optimizer

To evaluate the POSTGRES optimizer's performance as well as assess the credibility of its cost formulas, we could do the following. We could measure the amount of time required to execute various query plans, and then we could compare these measurements with the optimizer's predicted costs. Ideally, the predicted cost will be identical to the actual measured cost. However, this is an unrealistic expectation since optimizer cost formulas are merely estimates. A more attainable goal is for the optimizer to select the true optimal plan in a large majority of cases. This will at least show that in almost all circumstances, a selected plan is the best plan not only according to the optimizer but also in reality.

The tests described above illustrate how we would have wanted to validate the optimizer. However, at the time when we wanted to run these performance tests, measuring the amount of time required to execute a query plan was impossible because other parts of POSTGRES had not been fully implemented. As an alternative, we compared POSTGRES plans with query plans selected by the optimizer of commercial INGRES [KOOI82], which also happens to use an enumerative planning algorithm. The assumption behind this is that the INGRES optimizer selects ‘‘correct’’ plans. Therefore, if a plan selected by POSTGRES is equivalent to a plan selected by INGRES, (for the same query under similar conditions), then the POSTGRES optimizer has also generated a ‘‘correct’’ plan. Although this may not always be a valid assumption, it is probably a very good one since the INGRES optimizer has been tested, tuned, and used widely; and tests have shown that it is ‘‘extremely good’’ [ROWE86]. Furthermore, comparing POSTGRES and INGRES query plans at least allowed us to validate our optimizer against something other than our intuition as to which plans *should* be selected.

Commercial INGRES was chosen as a basis for our comparisons because it is ideal in several respects. First of all, it has a feature that allows users to examine plans selected by the optimizer. By issuing the command ‘‘**set qep**,’’ plan trees for all subsequent join queries will be printed on the terminal monitor, easily allowing us to make comparisons between INGRES and POSTGRES plans. In addition, both POSTGRES and INGRES store within database catalogs statistical information that can be used to compute more realistic operator selectivities. In INGRES, these statistics are generated and subsequently updated using the ‘‘**optimizedb**’’ command, while POSTGRES maintains them using system demons. An example of how this extra information would be put to use is the following. Suppose a relation, *r*, has 100 tuples but only 10 distinct values in attribute, *r.f*. Because the number of distinct values is a better indicator of the distribution of data within *r.f*, a clause like $r.f = 1$ would have a selectivity of $\frac{1}{10}$ rather than $\frac{1}{100}$, as a result of using this extra information. In both systems, we made use of this and other statistical information to generate more accurate cost estimations. As a result, fair comparisons could be made between the two optimizers.

However, the POSTGRES optimizer is *not* identical to the INGRES optimizer, and consequently, in certain situations, INGRES selected a plan that POSTGRES did not even consider. In these cases, it was impossible to determine whether POSTGRES had selected the optimal path from among those plans it had considered; but as will become evident in the next subsection, we were still able to draw some interesting conclusions. One situation where this occurs relates to the manner in which both systems treat secondary indices. Figure 5.1 illustrates how a secondary index is generally used. Given the key(s) of the index, the system locates within the secondary index relation a unique tuple identifier (tid). It then uses the tid to directly retrieve an appropriate tuple from the corresponding indexed relation. The POSTGRES query plan tree corresponding to such a secondary index scan is shown in figure 5.2a, while the INGRES tree is shown in figure 5.2b. A join between the tidp field in EMPINDEX and tid in EMP is required because INGRES literally treats secondary indices as relations, given that relations are used to implement them. In this particular case, although the two trees are different in appearance, they are both processed in the manner shown in figure 5.1. The only difference is that joining of tids is implicit in the POSTGRES indexscan node.

It will not always be the case that there will be a POSTGRES tree equivalent to an INGRES tree because by treating secondary indices as relations, further strategies are available to the INGRES optimizer. In figure 5.2, the index relation, EMPINDEX, is joined directly with its corresponding indexed relation, EMP. POSTGRES will only use a secondary index in this manner. INGRES, however, may choose

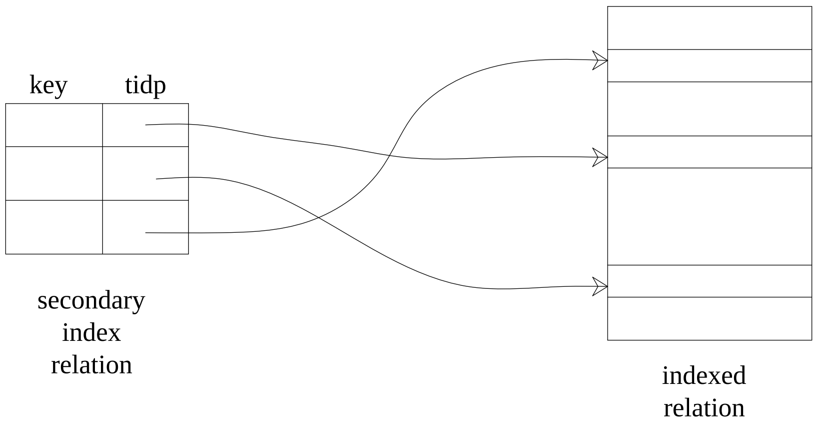


Figure 5.1
Using secondary indices

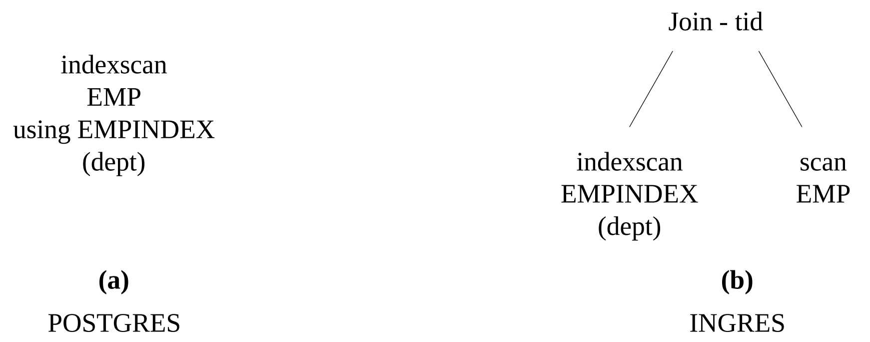


Figure 5.2
Plan trees for secondary index scans in POSTGRES and INGRES

to sort the result of a secondary scan, or it may join the result with a relation other than the corresponding indexed relation. Figure 5.3 illustrates these two situations. Examples where this generality is advantageous will be discussed in the next subsection.

Another disadvantage of testing our optimizer against INGRES's is that INGRES is a conventional database system that does not support user-defined operators and access methods or nested-attribute queries. As a result, we could only test standard operators, access methods, and queries. Fortunately, this

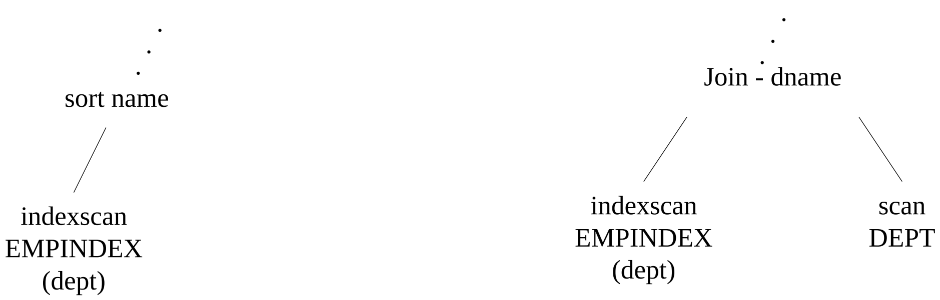


Figure 5.3
Other processing strategies using secondary indices in INGRES

drawback was insignificant. In POSTGRES, costs are computed using formulas as well as operator selectivities. The latter is supplied by the user. In other words, it is a parameter that is not inherent within the optimizer and thus cannot be manipulated or tuned (except by the user who supplied the routines). Consequently, provided selectivities relevant to a single operator and storage structure are accurate, one of each was sufficient for testing purposes. It would have been nice to illustrate the generality of our optimizer by using non-standard operators in our tests, but even standard operators and access methods in POSTGRES are implemented as if they were user-defined. Thus, no generality was lost in using a conventional operator and access method in our tests.

The single operator and access method used were the equality operator, since it is a mergesortable operator, and an ISAM storage structure [HELD75a]. To build an ISAM storage structure, tuples must first be sorted into data pages. Then, a multi-level directory is constructed indicating the high key value stored within each page. Such a structure provides quick access when used with the following operators:

$=, <, \leq, >$, and $\geq$.

The directory is analogous to the internal nodes of a B-tree except that once it has been built, the directory remains static. Therefore, when a data page is filled, rather than splitting nodes to maintain a balanced tree, overflow pages are created and linked to the filled data page. If a large number of overflow pages are created, finding a tuple within a page requires an inefficient linear search through the primary page as well as its overflow pages. So, given a choice between an ISAM storage structure and a B-tree, a user would probably choose a B-tree. However, we could not use B-trees in our tests because the version of INGRES that we used only supported ISAM and hash access methods. Forced to choose between the two, we chose ISAM because there is a greater overhead associated with searching through an ISAM directory, making them more interesting than hash tables.

Using an ISAM access method does have its disadvantages, though. Although tuples in an ISAM structure are initially sorted when the structure is built, the tuples are not maintained in sort order. As a result, we could not test the POSTGRES optimizer feature that takes advantage of access methods like

B-trees to eliminate sort steps required to order a user-specified sort result or tuples for a merge-sort join. However, although merge-sorting on a tuple by tuple basis is not possible, a variation of merge-sort can be performed on a page by page basis since the range of values within each page is known. INGRES, in fact, does this. In contrast, the POSTGRES optimizer does not, and as a result, differences arose in our performance tests. We chose not to account for partial index sorts because few access methods have this unusual characteristic. Moreover, as already alluded to, users will likely opt for access methods like B-trees, which always maintain their records in sort order. In other words, this feature would probably not be employed very often, had we implemented it. This should be kept in mind when differences arise between POSTGRES and INGRES plans in the next subsection.

With respect to nested-attribute queries, not being able to test these either is also of minimal importance. As discussed in section 3.5, relations materialized from queries embedded within data fields will generally be small, and as a result, the cost of executing any subplan corresponding to a nested portion of a query will also be small. Therefore, it is less important if the true optimal subplan is not selected while optimizing this portion of a query.

In testing the optimizer, we used version 2.1 of INGRES running on a VAX 11/780. To simulate the INGRES optimizer as closely as possible, we had to determine values for two system-dependent parameters that affect cost estimations. One of these is the page size used by the database. This is required because in estimating the cost of a sort, the optimizer must determine the number of pages occupied by the tuples to be sorted. By multiplying the number of tuples in an INGRES relation by the width of a tuple (including space for internal page pointers) and dividing by the number of pages in the relation, it turns out that INGRES uses 2K pages.

Determining a value for the second parameter, *W*, which relates I/O to CPU cost, was less straightforward. Assuming that it takes 30 milliseconds and about 2000 instructions of operating system overhead to read an I/O page, while manipulating a tuple only consumes about 200 CPU instructions, *W* is in the range of 0.03 to 0.1 [DEWI84]. For the queries used in our tests, using various values within this range did not affect the plan choice except in a few situations where a merge-sort join was chosen over nested iteration. This occurred when *W* was close to 0.03, a value that apparently minimized the cost of CPU-intensive tasks like sorting. In these situations, the INGRES optimizer always selected a nested iteration join. Thus, to insure compatibility with INGRES plans, we assigned *W* the value 0.065, arbitrarily choosing a value in the middle of our original range.

5.2. Tests Performed

For testing purposes, we used the following three relations: EMP, DEPT, and WATER. Schemas and relevant statistics for these relation are shown in figure 5.4. First, we ran the following four two-way join queries on EMP and DEPT:

A: **retrieve** (EMP.name, EMP.age, DEPT.all) **where**
EMP.dept = DEPT.dname

B: **retrieve** (EMP.name, EMP.age, DEPT.dname, DEPT.budget) **where**
EMP.dept = DEPT.dname **and**
DEPT.floor = 1

C: **retrieve** (EMP.age, EMP.dname, DEPT.floor, DEPT.budget) **where**
EMP.dept = DEPT.dname **and**
EMP.name = ‘‘Diamond’’

D: **retrieve** (EMP.age, DEPT.dname, DEPT.budget) **where**
EMP.dept = DEPT.dname **and**
DEPT.floor = 1 **and**
EMP.name = ‘‘Diamond’’

To illustrate that the optimizer chooses appropriate strategies under varying conditions, we ran the four queries for non-indexed relations as well as relations with primary and secondary indices defined on relevant attributes. Indices were selected so as to illustrate the combined effects of relation sizes, distribution of values, and secondary index overhead in determining a query’s optimal plan. The results of running queries A-D using various indices are shown in table 5.1. For the fourteen executions shown, when indices were defined, in all but one case both POSTGRES and INGRES made similar decisions as to whether or not the indices should be used, and except in cases where INGRES took advantage of a par-

RELATIONS :

EMP (name = c10, salary = i4, manager = c10, age = i4, dept = c10)
 # pages : 600 (unindexed)
 # tuples : 30,000
 # distinct values in name : 30,000
 # distinct values in dept = 1000

DEPT (dname = c10, floor = i4, budget = i4)
 # pages : 10 (unindexed), 14 (primary ISAM index on floor)
 # tuples : 1000
 # distinct values in dname : 1000
 # distinct values in floor : 9

WATER (cid = i4, floor = i4)
 # pages : 1 (unindexed)
 # tuples : 50
 # distinct values in cid : 9
 # distinct values in floor : 9

SECONDARY INDICES :

EMPINDEX (ISAM on dept) :
 # pages : 253
 # tuples : 30,000

DEPTINDEX (ISAM on floor) :
 # pages : 8
 # tuples : 1000

DEPTINDEX (ISAM on dept) :
 # pages : 11
 # tuples : 1000

Figure 5.4
Schemas and statistics for test relations

A: **retrieve** (EMP.name, EMP.age, DEPT.all) **where**
EMP.dept = DEPT.dname

B: **retrieve** (EMP.name, EMP.age, DEPT.dname, DEPT.budget) **where**
EMP.dept = DEPT.dname **and**
DEPT.floor = 1

C: **retrieve** (EMP.age, EMP.dname, DEPT.floor, DEPT.budget) **where**
EMP.dept = DEPT.dname **and**
EMP.name = ‘‘Diamond’’

D: **retrieve** (EMP.age, DEPT.dname, DEPT.budget) **where**
EMP.dept = DEPT.dname **and**
DEPT.floor = 1 **and**
EMP.name = ‘‘Diamond’’

Storage Structure	Query A	Query B	Query C	Query D
no indices	Plan (a) †	Plan (a)	Plan (c)	Plan (c)
secondary index on DEPT (floor)		Plan (a)		
primary index on DEPT (floor)		Plan (b)		Plan (d)
secondary index on EMP (dept)	Plan (c)		Plan (e)	
primary index on DEPT (floor) & secondary index on DEPT (dname)		Plan (b)		Plan (g)
secondary index on EMP (dept)	Plan (f)		Plan (c)	
secondary index on DEPT (dname) & secondary index on EMP (dept)	Plan (f)			

† See figure 5.5

Table 5.1
Results of executing queries A-D using various storage structures

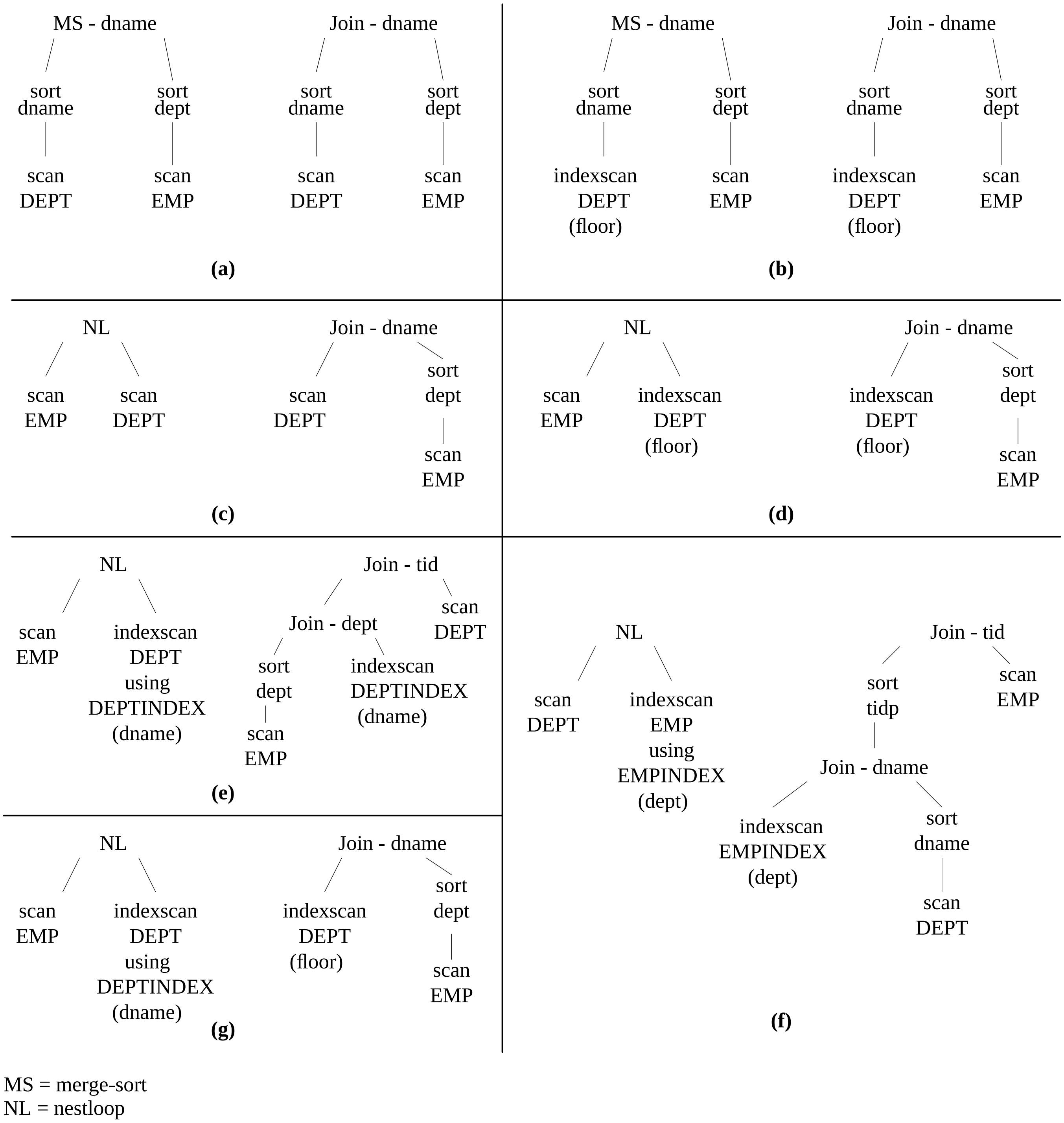


Figure 5.5
Comparison of selected POSTGRES and INGRES plans

tial ISAM sort, both selected equivalent join strategies.

Figure 5.5 gives a pictorial representation of the seven different corresponding plan trees mentioned in table 5.1. For each pair, the POSTGRES tree is shown on the left while the INGRES tree is on the right. Both pairs of plans (a) and (b) correspond to merge-sort joins, and both plan (b)'s scan DEPT using a primary index defined on ''floor.'' In plans (c) and (d), POSTGRES uses nested iteration, scanning EMP first because the highly restrictive clause,

EMP.name = ''Diamond,''

applies in these cases. INGRES, however, uses a slight variation of nested iteration in both (c) and (d). EMP is scanned and sorted into a temporary before it is nest iterated with DEPT. Sorting only the inner relation on the inner join attribute alleviates having to scan the entire inner join relation on every iteration because the scan halts after a value greater than the current outer join attribute has been reached. In certain cases, this can be helpful; however, in these cases, there is really no advantage because only a single tuple results after scanning EMP. In fact, these INGRES and POSTGRES plans are equivalent in cost because the cost of the extra sort step in the INGRES trees is zero, and in both cases, EMP and DEPT only need to be scanned once.

POSTGRES's plan (e) is similar to INGRES's except INGRES takes advantage of an ISAM's partial sort and thus uses a slight variation of a merge-sort join with EMP and DEPTINDEX. Again, only one tuple qualifies after scanning EMP, resulting in a sort cost of zero. Ignoring the sort step, the two plans are actually equivalent.

Plan (f) is an example where INGRES's treatment of an access method's partial ordering is helpful. In this particular case, INGRES takes advantage of the feature to join EMPINDEX with DEPT. The result of this join is then sorted on the tidp field so EMP can be scanned in page order. According to POSTGRES cost estimates, the INGRES plan is better by about a factor of four to five. It is significantly better than the POSTGRES plan partly because of INGRES's treatment of secondary indices, but more so due to INGRES accounting for an ISAM's partial sort. Therefore, the fact that INGRES was significantly better in this case is misleading for the reasons discussed in the previous subsection.

The last plan pair (g) illustrates the one case where different indices are selected by POSTGRES and INGRES. INGRES chooses to use the index defined on ‘‘floor,’’ while POSTGRES chooses the index defined on ‘‘dept.’’ POSTGRES chose the plan it did because according to its cost estimates the selected plan was a mere 0.27% better than a plan equivalent to that chosen by INGRES. Thus, although the two optimizers chose different plans, the difference is very minor.

The set of tests just described indicate that differences between selected INGRES and POSTGRES plans are rather subtle, resulting in minor variations, except in one case where the difference is minimized by other circumstances. Additional strategies employed by INGRES resulted in greater differences when the following three-way join:

> **retrieve** (EMP.name, DEPT.floor, WATER.cid) **where**
> EMP.dept = DEPT.dname **and**
> DEPT.floor = WATER.floor

was run on several different relation configurations. The results are shown in table 5.2.

Plans (1), (2), and (4) in table 5.2 illustrate another strategy employed by INGRES that POSTGRES does not consider. In these plans, INGRES first sorts DEPT on ‘‘dname’’ and then nest iterates the result with WATER. Since a join preserves the sort order of its outer join relation, the join result is still sorted in ‘‘dname’’ order and thus can be used in a merge with EMP. This is advantageous because it may be cheaper to sort DEPT prior to its join with WATER, since more tuples may have to be sorted after the join. Other differences were already discussed in reference to the previous set of tests and will not be repeated.

To assess the magnitude of differences in POSTGRES and INGRES plans for the four cases shown, the costs of all INGRES plans were calculated using POSTGRES cost estimation formulas. These computed costs were compared with the costs of corresponding POSTGRES plans, and the percentage differences are shown in table 5.2. The results show that INGRES was better than POSTGRES in all four cases. However, INGRES was only significantly better in one case, and this was a result of INGRES accounting for a partial ISAM ordering. In all other cases, the additional strategies considered by

storage structure	POSTGRES	INGRES	% diff.
(1) no indices	MS - dept sort dname sort dept MS - floor scan EMP sort floor sort floor scan WATER scan DEPT	Join - dept Join - floor sort dept sort dname sort floor scan EMP scan DEPT scan WATER	13.8
(2) primary index on DEPT floor	MS - floor sort dname sort dept NL scan DEPT scan WATER indexscan DEPT (floor)	Join - dept Join - floor sort dept sort dname sort floor scan EMP indexscan DEPT (floor) scan WATER	12.5

Table 5.2
Results of executing a three-way join on various storage structures

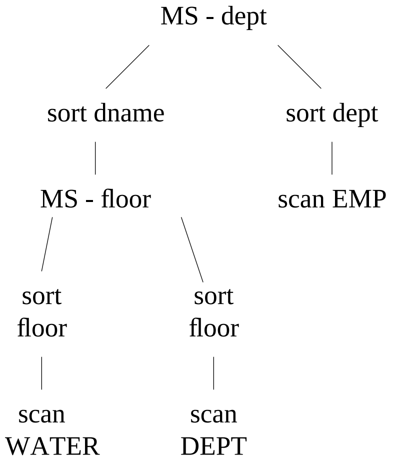
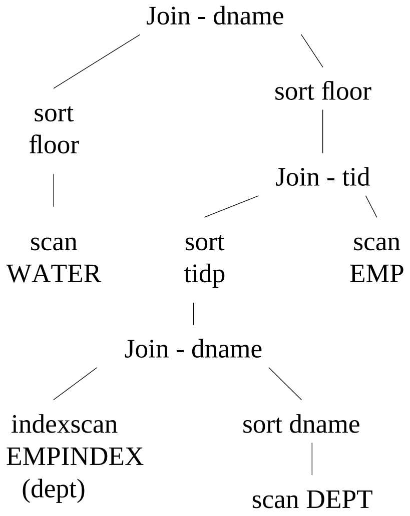
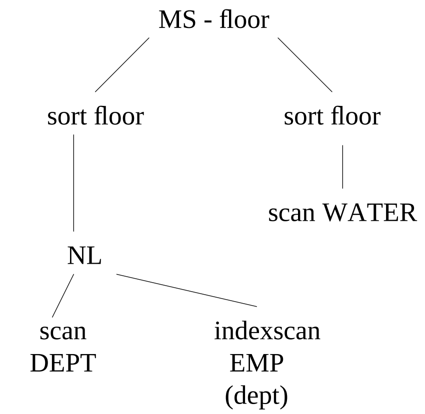
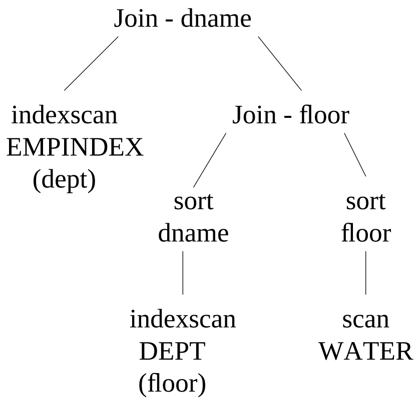
<table>
<tr><th>storage structure</th><th>POSTGRES</th><th>INGRES</th><th>% diff.</th></tr>
<tr><td>(3) secondary index on EMP dept</td><td>
</td><td>
</td><td>487</td></tr>
<tr><td>(4) primary index on DEPT floor and secondary index on EMP dept</td><td>
</td><td>
</td><td>3.4</td></tr>
</table>

Table 5.2
(continued)

INGRES resulted in no more than a 13.8% difference.

To summarize, in spite of differences between the POSTGRES and INGRES optimizers, we were able to draw two promising conclusions from our performance tests. For the most part, both systems selected similar strategies for queries involving a single join. Assuming that the INGRES optimizer is correct, these results indicate that POSTGRES selects true optimal plans. For queries for which INGRES was able to capitalize on a broader range of strategies, INGRES did not perform considerably better. Thus, not having considered all these other processing strategies, which would have added further complexity to our implementation effort, did not dramatically affect the overall performance of our optimizer.

6. CONCLUSION

It will be interesting to run further tests on the optimizer once it has been fully integrated with the rest of POSTGRES. Among the issues of interest are:

- whether writing the optimizer in LISP is a performance concern
- whether the optimizer *actually* selects true optimal plans
- whether the optimizer really works with *all* user-defined types, operators, and access methods
- whether the current algorithm for optimizing nested-attribute queries is ‘‘good enough.’’

Clearly, further validation of the POSTGRES optimizer is necessary to examine the more subtle issues raised by these questions. However, the preliminary work described in this report has at least illustrated the feasibility of many important ideas. Most significant of these are the concepts introduced to support optimization of extendibile objects in a database system.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my research advisor, Professor Michael Stonebraker, for his guidance and support throughout this project. I am also grateful to Professor Larry Rowe and members of the POSTGRES project for contributing ideas that improved the optimizer, particularly Chin Heng Hong for his careful scrutiny of optimizer-generated query plans. Finally, I would like to thank Professor Eugene Wong for reading this report.

REFERENCES

[ASTR76] Astrahan, M., et.al., ‘‘System R: A Relational Approach to Database Management,’’ *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[CHAM76] Chamberlin, D., et. al., ‘‘SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control,’’ *IBM Journal of Research and Development*, Vol. 20, No. 6, November 1976.

[DEWI84] DeWitt, D., et. al., ‘‘Implementation Techniques for Main Memory Database Systems,’’ *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984.

[FOGG82] Fogg, D., *Implementation of Domain Abstraction in the Relational Database System INGRES*, Masters Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, September 1982.

[FRAN85] Franz, Inc., *Franz LISP Reference Manual, Opus 42*, Berkeley, CA, September 1985.

[GUTT84] Guttman, A., ‘‘R-trees: A Dynamic Index Structure for Spatial Searching,’’ *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984.

[HASK82] Haskings, M. and R. Lorie, ‘‘On Extending the Functions of a Relational Database System,’’ *Proceedings of the 1982 ACM-SIGMOD Conference on Management of Data*, Orlando, FLA, June 1982.

[HELD75a] Held, G., and M. Stonebraker, ‘‘Access Methods in the Relational Data Base Management System INGRES,’’ *Proceedings of the ACM-Pacific-75*, San Francisco, CA, April 1975.

[HELD75b] Held, G., et. al., ‘‘INGRES: A Relational Data Base System,’’ *Proceedings of the 1975 AFIPS National Computer Conference*, Vol. 44, Anaheim, CA, May 1975.

[KENT79] Kent, W., ‘‘Limitations of Record-Based Information Models,’’ *ACM Transactions on Database Systems*, Vol. 4, No. 1, March 1979.

[KOOI82] Kooi, R. and D. Frankforth, ‘‘Query Optimization in INGRES,’’ *IEEE Database Engineering Newsletter: Special Issue on Query Optimization*, Vol. 5, No. 1, September 1982.

[KUNG84] Kung, R., et. al., ‘‘Heuristic Search in Data Base Systems,’’ *Proceedings of the 1st International Workshop on Expert Data Base Systems*, Kiowah Isl., SC, October 1984.

[ONG82] Ong, J., *The Design and Implementation of Data Abstraction in the Relational Database System INGRES*, Masters Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, September 1982.

[ROBI81] Robinson, J., ‘‘The K-D-B Tree: A Search Structure for Large Multidimensional Indexes,’’ *Proceedings of the 1981 ACM-SIGMOD Conference on Management of Data*, Ann Arbor, MI, May 1981.

[ROWE86] Rowe, L. A. and M. Stonebraker, ‘‘The Commercial INGRES Epilogue,’’ *The INGRES Papers: Anatomy of a Relational Database System*, (M. Stonebraker, editor), Addison Wesley, 1986.

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

[SELL85] Sellis, T. and L. Shapiro, ''Optimization of Extended Database Query Languages,'' *Proceedings of the 1985 ACM-SIGMOD Conference on Management of Data*, Austin, TX, May 1985.

[SELL86] Sellis, T., ''Global Query Optimization,'' *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washingtion D.C., May 1986.

[STEE84] Steele, G., *CommonLISP*, Digital Press, 1984.

[STON75] Stonebraker, M., ''Implementation of Integrity Constraints and Views by Query Modification,'' *Proceedings of the 1975 ACM-SIGMOD Conference on Management of Data*, San Jose, CA, May 1975.

[STON76] Stonebraker, M., et. al., ''The Design and Implementation of INGRES,'' *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[STON84] Stonebraker, M., et. al., ''QUEL as a Data Type,'' *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1984.

[STON85a] Stonebraker, M., ''Triggers and Inference in Data Base Systems,'' *Proceedings of the Islamoora Conference on Expert Data Bases*, Islamoora, FLA, February 1985.

[STON85b] Stonebraker, M., et. al., ''Extending a Data Base System With Procedures,'' *(submitted for publication)*.

[STON86a] Stonebraker, M., ''Inclusion of New Types in Relational Data Base Systems, *Proceedings of the Second International Conference on Database Engineering*, Los Angeles, CA, February 1986.

[STON86b] Stonebraker, M. and L. A. Rowe, ''The Design of POSTGRES,'' *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washington D.C., May 1986.

[STON86c] Stonebraker, M., ''Object Management in POSTGRES Using Procedures,'' *International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[STON86d] Stonebraker, M., et. al., ''The Design of the POSTGRES Rules System,'' *(in preparation)*.

[WONG76] Wong, E. and K. Youssefi, ''Decomposition: A Strategy for Query Processing,'' *ACM Transactions on Database Systems*, Vol. 1, No. 3. September 1976.

[ZANI83] Zaniolo, C., ''The Database Language GEM,'' *Proceedings of the 1983 ACM-SIGMOD Conference on Management of Data*, San Jose, CA, May 1983.