Access Path Selection in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

1. Introduction

System R is an experimental database management system based on the relational model of data which has been under development at the IBM San Jose Research Laboratory since 1975 <1>. The software was developed as a research vehicle in relational database, and is not generally available outside the IBM Research Division.

This paper assumes familiarity with relational data model terminology as described in Codd <7> and Date <8>. The user interface in System R is the unified query, data definition, and manipulation language SQL <5>. Statements in SQL can be issued both from an on-line casual-user-oriented terminal interface and from programming languages such as PL/I and COBOL.

In System R a user need not know how the tuples are physically stored and what access paths are available (e.g. which columns have indexes). SQL statements do not require the user to specify anything about the access path to be used for tuple retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

This paper will address the issues of access path selection for queries. Retrieval for data manipulation (UPDATE, DELETE) is treated similarly. Section 2 will describe the place of the optimizer in the processing of a SQL statement, and section 3 will describe the storage component access paths that are available on a single physically stored table. In section 4 the optimizer cost formulas are introduced for single table queries, and section 5 discusses the joining of two or more tables, and their corresponding costs. Nested queries (queries in predicates) are covered in section 6.

2. Processing of an SQL statement

A SQL statement is subjected to four phases of processing. Depending on the origin and contents of the statement, these phases may be separated by arbitrary intervals of time. In System R, these arbitrary time intervals are transparent to the system components which process a SQL statement. These mechanisms and a description of the processing of SQL statements from both programs and terminals are further discussed in <2>. Only an overview of those processing steps that are relevant to access path selection will be discussed here.

The four phases of statement processing are parsing, optimization, code generation, and execution. Each SQL statement is sent to the parser. where it is checked for correct syntax. A query block is represented by a SELECT list, a FROM list, and a WHERE tree, containing, respectively the list of items to be retrieved, the table(s) referenced, and the boolean combination of simple predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one

operand which is itself a query.

If the parser returns without any errors detected, the OPTIMIZER component is called. The OPTIMIZER accumulates the names of tables and columns referenced in the query and looks them up in the System R catalogs to verify their existence and to retrieve information about them.

The catalog lookup portion of the OPTIMIZER also obtains statistics about the referenced relations, and the access paths available on each of them. These will be used later in access path selection. After catalog lookup has obtained the datatype and length of each column, the OPTIMIZER rescans the SELECT-list and WHERE-tree to check for semantic errors and type compatibility in both expressions and predicate comparisons.

Finally the OPTIMIZER performs access path selection. It first determines the evaluation order among the query blocks in the statement. Then for each query block, the relations in the FROM list are processed. If there is more than one relation in a block, permutations of the join order and of the method of joining are evaluated. The access paths that minimize total cost for the block are chosen from a tree of alternate path choices. This minimum cost solution is represented by a structural modification of the parse tree. The result is an execution plan in the Access Specification Language (ASL) <10>.

After a plan is chosen for each query block and represented in the parse tree, the CODE GENERATOR is called. The CODE GENERATOR is a table-driven program which translates ASL trees into machine language code to execute the plan chosen by the OPTIMIZER. In doing this it uses a relatively small number of code templates, one for each type of join method (including no join). Query blocks for nested queries are treated as "subroutines" which return values to the predicates in which they occur. The CODE GENERATOR is further described in <9>.

During code generation, the parse tree is replaced by executable machine code and its associated data structures. Either control is immediately transfered to this code or the code is stored away in the database for later execution, depending on the origin of the statement (program or terminal). In either case, when the code is ultimately executed, it calls upon the System R internal storage system (RSS) via the storage system interface (RSI) to scan each of the physically stored relations in the query. These scans are along the access paths chosen by the OPTIMIZER. The RSI commands that may be used by generated code are described in the next section.

3. The Research Storage System

The Research Storage System (RSS) is the storage subsystem of System R. It is responsible for maintaining physical storage of relations, access paths on these relations, locking (in a multi-user environment), and logging and recovery facilities. The RSS presents a tuple-oriented interface (RSI) to its users. Although the RSS may be used independently of System R, we are concerned here with its use for executing the code generated by the processing of SQL statements in System R, as described in the previous section. For a complete description of the RSS, see <1>.

Relations are stored in the RSS as a collection of tuples whose columns are physically contiguous. These tuples are stored on 4K byte pages; no tuple spans a page. Pages are organized into logical units called segments. Segments may contain one or more relations, but no relation may span a segment. Tuples from two or more relations may occur on the same page. Each tuple is tagged with the identification of the relation to which it belongs.

The primary way of accessing tuples in a relation is via an RSS scan. A scan returns a tuple at a time along a given access path. OPEN, NEXT, and CLOSE are the principal commands on a scan.

Two types of scans are currently available for SQL statements. The first type is a segment scan to find all the tuples of a given relation. A series of NEXTs on a segment scan simply examines all pages of the segment which contain tuples from any relation, and returns those tuples belonging to the given relation.

The second type of scan is an index scan. An index may be created by a System R user on one or more columns of a relation, and a relation may have any number (including zero) of indexes on it. These indexes are stored on separate pages from those containing the relation tuples. Indexes are implemented as B-trees <3>, whose leaves are pages containing sets of (key, identifiers of tuples which contain that key). Therefore a series of NEXTs on an index scan does a sequential read along the leaf pages of the index, obtaining the tuple identifiers matching a key, and using them to find and return the data tuples to the user in key value order. Index leaf pages are chained together so that NEXTs need not reference any upper level pages of the index.

In a segment scan, all the non-empty pages of a segment will be touched, regardless of whether there are any tuples from the desired relation on them. However, each page is touched only once. When an entire relation is examined via an index scan, each page of the index is touched

only once, but a data page may be examined more than once if it has two tuples on it which are not "close" in the index ordering. If the tuples are inserted into segment pages in the index ordering, and if this physical proximity corresponding to index key value is maintained, we say that the index is clustered. A clustered index has the property that not only each index page, but also each data page containing a tuple from that relation will be touched only once in a scan on that index.

An index scan need not scan the entire relation. Starting and stopping key values may be specified in order to scan only those tuples which have a key in a range of index values. Both index and segment scans may optionally take a set of predicates, called search arguments (or SARGS), which are applied to a tuple before it is returned to the RSI caller. If the tuple satisfies the predicates, it is returned; otherwise the scan continues until it either finds a tuple which satisfies the SARGS or exhausts the segment or the specified index value range. This reduces cost by eliminating the overhead of making RSI calls for tuples which can be efficiently rejected within the RSS. Not all predicates are of the form that can become SARGS. A sargable predicate is one of the form (or which can be put into the form) "column comparison-operator value". SARGS are expressed as a boolean expression of such predicates in disjunctive normal form.

4. Costs for single relation access paths

In the next several sections we will describe the process of choosing a plan for evaluating a query. We will first describe the simplest case, accessing a single relation, and show how it extends and generalizes to 2-way joins of relations, n-way joins, and finally multiple query blocks (nested queries).

The OPTIMIZER examines both the predicates in the query and the access paths available on the relations referenced by the query, and formulates a cost prediction for each access plan, using the following cost formula:

COST = PAGE FETCHES + W * (RSI CALLS).

This cost is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). W is an adjustable weighting factor between I/O and CPU. RSI CALLS is the predicted number of tuples returned from the RSS. Since most of System R's CPU time is spent in the RSS, the number of RSI calls is a good approximation for CPU utilization. Thus the choice of a minimum cost path to process a query attempts to minimize total resources required.

During execution of the type-compatibility and semantic checking portion of the OPTIMIZER, each query block's WHERE tree of predicates is examined. The WHERE tree is considered to be in conjunctive normal form, and every conjunct is called a boolean factor. Boolean factors are notable because every tuple returned to the user must satisfy every boolean factor. An index is said to match a boolean factor if the boolean factor is a sargable predicate whose referenced column is the index key; e.g., an index on SALARY matches the predicate 'SALARY = 20000'. More precisely, we say that a predicate or set of predicates matches an index access path when the predicates are sargable and the columns mentioned in the predicate(s) are an initial substring of the set of columns of the index key. For example, a NAME, LOCATION index matches NAME = 'SMITH' AND LOCATION = 'SAN JOSE'. If an index matches a boolean factor, an access using that index is an efficient way to satisfy the boolean factor. Sargable boolean factors can also be efficiently satisfied if they are expressed as search arguments. Note that a boolean factor may be an entire tree of predicates headed by an OR.

During catalog lookup, the OPTIMIZER retrieves statistics on the relations in the query and on the access paths available on each relation. The statistics kept are the following:

For each relation T,
- NCARD(T), the cardinality of relation T.
- TCARD(T), the number of pages in the segment that hold tuples of relation T.
- P(T), the fraction of data pages in the segment that hold tuples of relation T.
  P(T) = TCARD(T) / (no. of non-empty pages in the segment).

For each index I on relation T,
- ICARD(I), number of distinct keys in index I.
- NINDX(I), the number of pages in index I.

These statistics are maintained in the System R catalogs, and come from several sources. Initial relation loading and index creation initialize these statistics. They are then updated periodically by an UPDATE STATISTICS command, which can be run by any user. System R does not update these statistics at every INSERT, DELETE, or UPDATE because of the extra database operations and the locking bottleneck this would create at the system catalogs. Dynamic updating of statistics would tend to serialize accesses that modify the relation contents.

Using these statistics, the OPTIMIZER assigns a selectivity factor 'F' for each boolean factor in the predicate list. This selectivity factor very roughly corresponds to the expected fraction of tuples which will satisfy the predicate. TABLE 1 gives the selectivity factors for different kinds of predicates. We assume that a lack of statistics implies that the relation is small, so an arbitrary factor is chosen.

TABLE 1 SELECTIVITY FACTORS

column = value

F = 1 / ICARD(column index) if there is an index on column
This assumes an even distribution of tuples among the index key values.
F = 1/10 otherwise

column1 = column2

F = 1/MAX(ICARD(column1 index), ICARD(column2 index))
if there are indexes on both column1 and column2
This assumes that each key value in the index with the smaller cardinality has a matching value in the other index.
F = 1/ICARD(column-i index) if there is only an index on column-i
F = 1/10 otherwise

column > value (or any other open-ended comparison)

F = (high key value - value) / (high key value - low key value)
Linear interpolation of the value within the range of key values yields F if the column is an arithmetic type and value is known at access path selection time.
F = 1/3 otherwise (i.e. column not arithmetic)
There is no significance to this number, other than the fact that it is less selective than the guesses for equal predicates for which there are no indexes, and that it is less than 1/2. We hypothesize that few queries use predicates that are satisfied by more than half the tuples.

column BETWEEN value1 AND value2

F = (value2 - value1) / (high key value - low key value)

A ratio of the BETWEEN value range to the entire key value range is used as the selectivity factor if column is arithmetic and both value1 and value2 are known at access path selection.
F = 1/4 otherwise
Again there is no significance to this choice except that it is between the default selectivity factors for an equal predicate and a range predicate.

column IN (list of values)

F = (number of items in list) * (selectivity factor for column = value)
This is allowed to be no more than 1/2.

columnA IN subquery

F = (expected cardinality of the subquery result) /
(product of the cardinalities of all the relations in the subquery's FROM-list).
The computation of query cardinality will be discussed below.
This formula is derived by the following argument:
Consider the simplest case, where subquery is of the form "SELECT columnB FROM relationC ...". Assume that the set of all columnB values in relationC contains the set of all columnA values. If all the tuples of relationC are selected by the subquery, then the predicate is always TRUE and F = 1. If the tuples of the subquery are restricted by a selectivity factor F', then assume that the set of unique values in the subquery result that match columnA values is proportionately restricted, i.e. the selectivity factor for the predicate should be F'. F' is the product of all the subquery's selectivity factors, namely (subquery cardinality) / (cardinality of all possible subquery answers). With a little optimism, we can extend this reasoning to include subqueries which are joins and subqueries in which columnB is replaced by an arithmetic expression involving column names. This leads to the formula given above.

(pred expression1) OR (pred expression2)

F = F(pred1) + F(pred2) - F(pred1) * F(pred2)

(pred1) AND (pred2)

F = F(pred1) * F(pred2)

Note that this assumes that column values are independent.

NOT pred

F = 1 - F(pred)

Query cardinality (QCARD) is the product of the cardinalities of every relation in the query block's FROM list times the product of all the selectivity factors of that query block's boolean factors. The number of expected RSI calls (RSICARD) is the product of the relation cardinalities times the selectivity factors of the sargable boolean factors, since the sargable boolean factors will be put into search arguments which will filter out tuples without returning across the RSS interface.

Choosing an optimal access path for a single relation consists of using these selectivity factors in formulas together with the statistics on available access paths. Before this process is described, a definition is needed. Using an index access path or sorting tuples produces tuples in the index value or sort key order. We say that a tuple order is an interesting order if that order is one specified by the query block's GROUP BY or ORDER BY clauses.

For single relations, the cheapest access path is obtained by evaluating the cost for each available access path (each index on the relation, plus a segment scan). The costs will be described below. For each such access path, a predicted cost is computed along with the ordering of the tuples it will produce. Scanning along the SALARY index in ascending order, for example, will produce some cost C and a tuple order of SALARY (ascending). To find the cheapest access plan for a single relation query, we need only to examine the cheapest access path which produces tuples in each "interesting" order and the cheapest "unordered" access path. Note that an "unordered" access path may in fact produce tuples in some order, but the order is not "interesting". If there are no GROUP BY or ORDER BY clauses on the query, then there will be no interesting orderings, and the cheapest access path is the one chosen. If there are GROUP BY or ORDER BY clauses, then the cost for producing that interesting ordering must be compared to the cost of the cheapest unordered path plus the cost of sorting QCARD tuples into the proper order. The cheapest of these alternatives is chosen as the plan for the query block.

The cost formulas for single relation access paths are given in TABLE 2. These formulas give index pages fetched plus data pages fetched plus the weighting factor times RSI tuple retrieval calls. W is the weighting factor between page fetches and RSI calls. Some situations give several alternative formulas depending on whether the set of tuples retrieved will fit entirely in the RSS buffer pool (or effective buffer pool per user). We assume for clustered indexes that a page remains in the buffer long enough for every tuple to be retrieved from it. For non-clustered indexes, it is assumed that for those relations not fitting in the buffer, the relation is sufficiently large with respect to the buffer size that a page fetch is required for every tuple retrieval.

TABLE 2

COST FORMULAS

SITUATION	COST (in pages)
Unique index matching an equal predicate	1 + 1 + W
Clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + NCARD) + W * RSICARD or F(preds) * (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Clustered index I not matching any boolean factors	(NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I not matching any boolean factors	(NINDX(I) + NCARD) + W * RSICARD or (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Segment scan	TCARD/P + W * RSICARD

5. Access path selection for joins

In 1976, Blasgen and Eswaran <4> examined a number of methods for performing 2-way joins. The performance of each of these methods was analyzed under a variety of relation cardinalities. Their evidence indicates that for other than very small relations, one of two join methods were always optimal or near optimal. The System R optimizer chooses between these two methods. We first describe these methods, and then discuss how they are extended for n-way joins. Finally we specify how the join order (the order in which the relations are joined) is chosen. For joins involving two relations, the two relations are called the outer relation, from which a tuple will be retrieved first, and the inner relation, from which tuples will be retrieved, possibly depending on the values obtained in the outer relation tuple. A predicate which relates columns of two tables to be joined is called a join predicate. The columns referenced in a join predicate are called join columns.

The first join method, called the nested loops method, uses scans, in any order, on the outer and inner relations. The scan on the outer relation is opened and the first tuple is retrieved. For each outer relation tuple obtained, a scan is opened on the inner relation to retrieve, one at a time, all the tuples of the inner relation which satisfy the join predicate. The composite tuples formed by the outer-relation-tuple / inner-relation-tuple pairs comprise the result of this join.

The second join method, called merging scans, requires the outer and inner relations to be scanned in join column order. This implies that, along with the columns mentioned in ORDER BY and GROUP BY, columns of equi-join predicates (those of the form Table1.column1 = Table2.column2) also define "interesting" orders. If there is more than one join predicate, one of them is used as the join predicate and the others are treated as ordinary predicates. The merging scans method is only applied to equi-joins, although in principle it could be applied to other types of joins. If one or both of the relations to be joined has no indexes on the join column, it must be sorted into a temporary list which is ordered by the join column.

The more complex logic of the merging scan join method takes advantage of the ordering on join columns to avoid rescanning the entire inner relation (looking for a match) for each tuple of the outer relation. It does this by synchronizing the inner and outer scans by reference to matching join column values and by "remembering" where matching join groups are located. Further savings occur if the inner relation is clustered on the join column (as would be true if it is the output of a sort on the join column). "Clustering" on a column means that tuples which have the same value in that column are physically stored close to each other so that one page access will retrieve several tuples.

N-way joins can be visualized as a sequence of 2-way joins. In this visualization, two relations are joined together, the resulting composite relation is joined with the third relation, etc. At each step of the n-way join it is possible to identify the outer relation (which in general is composite) and the inner relation (the relation being added to the join). Thus the methods described above for two way joins are easily generalized to n-way joins. However, it should be emphasized that the first 2-way join does not have to be completed before the second 2-way join is started. As soon as we get a composite tuple for the first 2-way join, it can be joined with tuples of the third relation to form result tuples for the 3-way join, etc. Nested loop joins and merge scan joins may be mixed in the same query, e.g. the first two relations of a three-way join may be joined using merge scans and the composite result may be joined with the third relation using a nested loop join. The intermediate composite relations are physically stored only if a sort is required for the next join step. When a sort of the composite relation is not specified, the composite relation will be materialized one tuple at a time to participate in the next join.

We now consider the order in which the relations are chosen to be joined. It should be noted that although the cardinality of the join of n relations is the same regardless of join order, the cost of joining in different orders can be substantially different. If a query block has n relations in its FROM list, then there are n factorial permutations of relation join orders. The search space can be reduced by observing that that once the first k relations are joined, the method to join the composite to the k+1-st relation is independent of the order of joining the first k; i.e. the applicable predicates are the same, the set of interesting orderings is the same, the possible join methods are the same, etc. Using this property, an efficient way to organize the search is to find the best join order for successively larger subsets of tables.

A heuristic is used to reduce the join order permutations which are considered. When possible, the search is reduced by consideration only of join orders which have join predicates relating the inner relation to the other relations already participating in the join. This means that in joining relations t1,t2,...,tn only those orderings ti1,ti2,...,tin are examined in which for all j (j=2,...,n) either
(1) tij has at least one join predicate

with some relation tik, where k < j, or (2) for all k > j, tik has no join predicate with ti1,ti2,...,or ti(j-1).
This means that all joins requiring Cartesian products are performed as late in the join sequence as possible. For example, if T1,T2,T3 are the three relations in a query block's FROM list, and there are join predicates between T1 and T2 and between T2 and T3 on different columns than the T1-T2 join, then the following permutations are not considered:

T1-T3-T2
T3-T1-T2

To find the optimal plan for joining n relations, a tree of possible solutions is constructed. As discussed above, the search is performed by finding the best way to join subsets of the relations. For each set of relations joined, the cardinality of the composite relation is estimated and saved. In addition, for the unordered join, and for each interesting order obtained by the join thus far, the cheapest solution for achieving that order and the cost of that solution are saved. A solution consists of an ordered list of the relations to be joined, the join method used for each join, and a plan indicating how each relation is to be accessed. If either the outer composite relation or the inner relation needs to be sorted before the join, then that is also included in the plan. As in the single relation case, "interesting" orders are those listed in the query block's GROUP BY or ORDER BY clause, if any. Also every join column defines an "interesting" order. To minimize the number of different interesting orders and hence the number of solutions in the tree, equivalence classes for interesting orders are computed and only the best solution for each equivalence class is saved. For example, if there is a join predicate E.DNO = D.DNO and another join predicate D.DNO = F.DNO, then all three of these columns belong to the same order equivalence class.

The search tree is constructed by iteration on the number of relations joined so far. First, the best way is found to access each single relation for each interesting tuple ordering and for the unordered case. Next, the best way of joining any relation to these is found, subject to the heuristics for join order. This produces solutions for joining pairs of relations. Then the best way to join sets of three relations is found by consideration of all sets of two relations and joining in each third relation permitted by the join order heuristic. For each plan to join a set of relations, the order of the composite result is kept in the tree. This allows consideration of a merge scan join which would not require sorting the composite. After the complete solutions (all of the relations joined together) have been found, the optimizer chooses the cheapest solution which gives the required order, if any was specified. Note that if a solution exists with the correct order, no sort is performed for ORDER BY or GROUP BY, unless the ordered solution is more expensive than the cheapest unordered solution plus the cost of sorting into the required order.

The number of solutions which must be stored is at most 2**n (the number of subsets of n tables) times the number of interesting result orders. The computation time to generate the tree is approximately proportional to the same number. This number is frequently reduced substantially by the join order heuristic. Our experience is that typical cases require only a few thousand bytes of storage and a few tenths of a second of 370/158 CPU time. Joins of 8 tables have been optimized in a few seconds.

Computation of costs

The costs for joins are computed from the costs of the scans on each of the relations and the cardinalities. The costs of the scans on each of the relations are computed using the cost formulas for single relation access paths presented in section 4.

Let C-outer(path1) be the cost of scanning the outer relation via path1, and N be the cardinality of the outer relation tuples which satisfy the applicable predicates. N is computed by:

N = (product of the cardinalities of all relations T of the join so far) * (product of the selectivity factors of all applicable predicates).

Let C-inner(path2) be the cost of scanning the inner relation, applying all applicable predicates. Note that in the merge scan join this means scanning the contiguous group of the inner relation which corresponds to one join column value in the outer relation. Then the cost of a nested loop join is

C-nested-loop-join(path1,path2)=
C-outer(path1) + N * C-inner(path2)

The cost of a merge scan join can be broken up into the cost of actually doing the merge plus the cost of sorting the outer or inner relations, if required. The cost of doing the merge is

C-merge(path1,path2)=
C-outer(path1) + N * C-inner(path2)

For the case where the inner relation is sorted into a temporary relation none of the single relation access path formulas in section 4 apply. In this case the inner scan is like a segment scan except that the merging scans method makes use of the fact that the inner relation is sorted so that it is not necessary to scan the entire inner relation looking for a match. For this case we use the following formula for the cost of the inner scan.

C-inner(sorted list) =
TEMPPAGES/N + W*RSICARD

where TEMPPAGES is the number of pages

required to hold the inner relation. This formula assumes that during the merge each page of the inner relation is fetched once.

It is interesting to observe that the cost formula for nested loop joins and the cost formula for merging scans are essentially the same. The reason that merging scans is sometimes better than nested loops is that the cost of the inner scan may be much less. After sorting, the inner relation is clustered on the join column which tends to minimize the number of pages fetched, and it is not necessary to scan the entire inner relation (looking for a match) for each tuple of the outer relation.

The cost of sorting a relation, C-sort(path), includes the cost of retrieving the data using the specified access path, sorting the data, which may involve several passes, and putting the results into a temporary list. Note that prior to sorting the inner table, only the local predicates can be applied. Also, if it is necessary to sort a composite result, the entire composite relation must be stored in a temporary relation before it can be sorted. The cost of inserting the composite tuples into a temporary relation before sorting is included in C-sort(path).

Example of tree

We now show how the search is done for the example join shown in Fig. 1. First we find all of the reasonable access paths for single relations with only their local predicates applied. The results for this example are shown in Fig. 2. There are three access paths for the EMP table: an index on DNO, an index on JOB, and a segment scan. The interesting orders are DNO and JOB. The index on DNO provides the tuples in DNO order and the index on JOB provides the tuples in JOB order. The segment scan access path is, for our purposes, unordered. For this example we assume that the index on JOB is the cheapest path, so the segment scan path is pruned. For the DEPT relation there are two access paths, an index on DNO and a segment scan. We assume that the index on DNO is cheaper so the segment scan path is pruned. For the JOB relation there are two access paths, an index on JOB and a segment scan. We assume that the segment scan path is cheaper, so both paths are saved. The results just described are saved in the search tree as shown in Fig. 3. In the figures, the notation C(EMP.DNO) or C(E.DNO) means the cost of scanning EMP via the DNO index, applying all predicates which are applicable given that tuples from the specified set of relations have already been fetched. The notation Ni is used to represent the cardinalities of the different partial results.

Next, solutions for pairs of relations are found by joining a second relation to

EMP

NAME	DNO	JOB	SAL
SMITH	50	12	8500
JONES	50	5	15000
DOE	51	5	9500

DEPT

DNO	DNAME	LOC
50	MFG	DENVER
51	BILLING	BOULDER
52	SHIPPING	DENVER

JOB

JOB	TITLE
5	CLERK
6	TYPIST
9	SALES
12	MECHANIC

```
SELECT  NAME, TITLE, SAL, DNAME
FROM    EMP, DEPT, JOB
WHERE   TITLE='CLERK'
AND     LOC='DENVER'
AND     EMP.DNO=DEPT.DNO
AND     EMP.JOB=JOB.JOB
```

"Retrieve the name, salary, job title, and department name of employees who are clerks and work for departments in Denver."

Figure 1. JOIN example

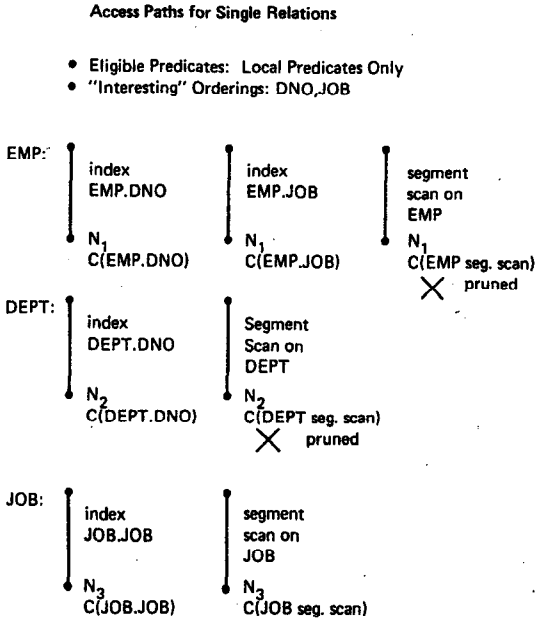

Figure 2.

the results for single relations shown in Fig. 3. For each single relation, we find access paths for joining in each second relation for which there exists a predicate connecting it to the first relation. First we consider access path selection for nested loop joins. In this example we assume that the EMP-JOB join is cheapest by accessing JOB on the JOB index. This is

likely since it can fetch directly the tuples with matching JOB (without having to scan the entire relation). In practice the cost of joining is estimated using the formulas given earlier and the cheapest path is chosen. For joining the EMP relation to the DEPT relation we assume that the DNO index is cheapest. The best access path for each second-level relation is combined with each of the plans in Fig. 3 to form the nested loop solutions shown in Fig. 4.

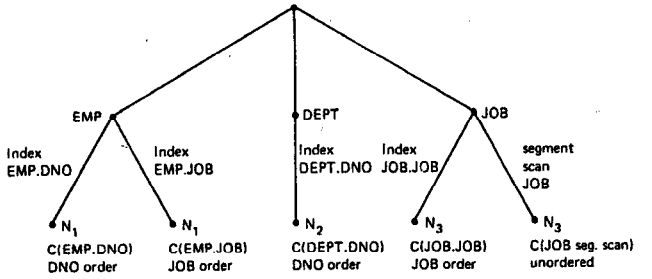

Figure 3. Search tree for single relations

Next we generate solutions using the merging scans method. As we see on the left side of Fig. 3, there is a scan on the EMP relation in DNO order, so it is possible to use this scan and the DNO scan on the DEPT relation to do a merging scans join, without any sorting. Although it is possible to do the merging join without sorting as just described, it might be cheaper to use the JOB index on EMP, sort on DNO, and then merge. Note that we never consider sorting the DEPT table because the cheapest scan on that table is already in DNO order.

For merging JOB with EMP, we only consider the JOB index on EMP since it is the cheapest access path for EMP regardless of order. Using the JOB index on JOB, we can merge without any sorting. However, it might be cheapter to sort JOB using a relation scan as input to the sort and then do the merge.

Referring to Fig. 3, we see that the access path chosen for the the DEPT relation is the DNO index. After accessing DEPT via this index, we can merge with EMP using the DNO index on EMP, again without any sorting. However, it might be cheaper to sort EMP first using the JOB index as input to the sort and then do the merge. Both of these cases are shown in Fig. 5.

As each of the costs shown in Figs. 4 and 5 are computed they are compared with the cheapest equivalent solution (same tables and same result order) found so far, and the cheapest solution is saved. After this pruning, solutions for all three relations are found. For each pair of relations, we find access paths for joining in the remaining third relation. As before we will extend the tree using nested loop joins and merging scans to join the third relation. The search tree for three relations is shown in Fig. 6. Note that in one case both the composite relation and the table being added (JOB) are sorted. Note also that for some of the cases no sorts are performed at all. In these cases, the composite result is materialized one tuple at a time and the intermediate composite relation is never stored. As before, as each of the costs are computed they are compared with the cheapest solu-

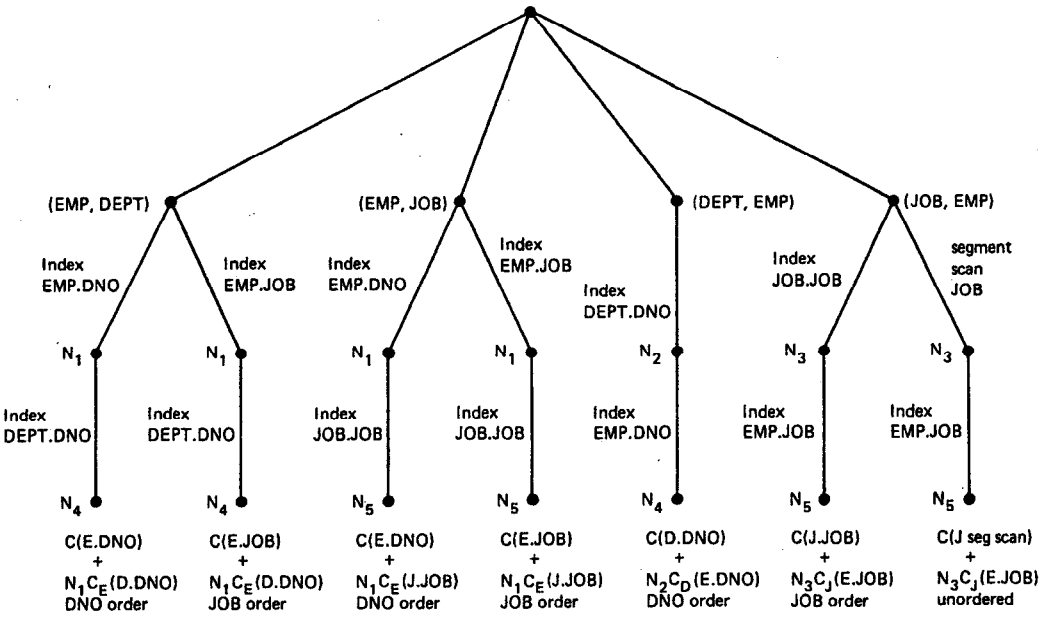

Figure 4. Extended search tree for second relation (nested loop join)

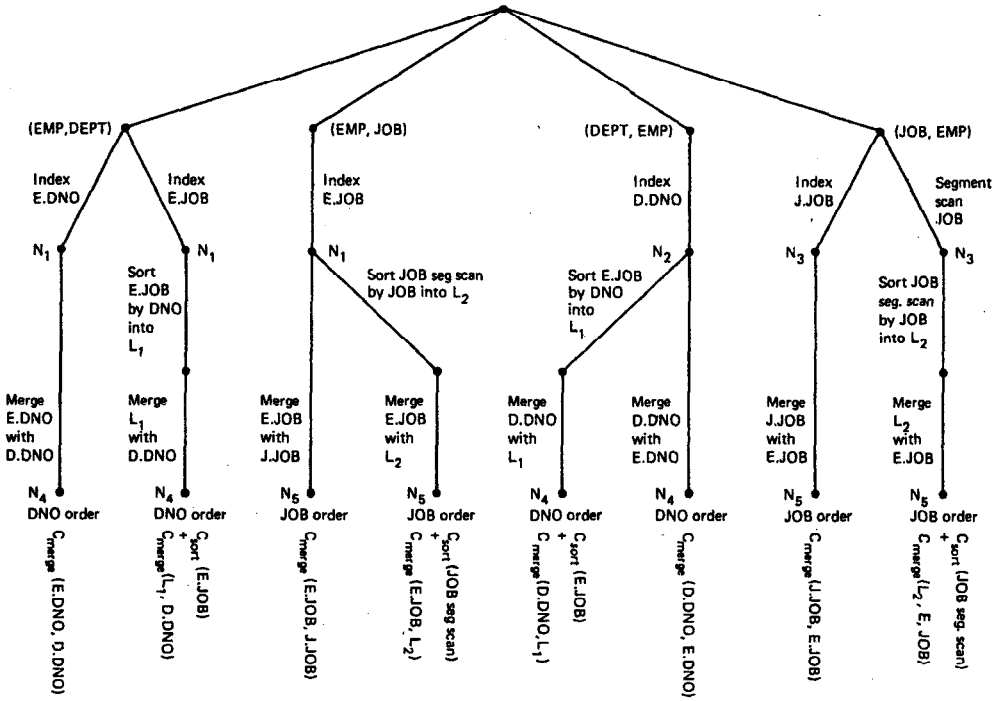

Figure 5. Extended search tree for second relation (merge join)

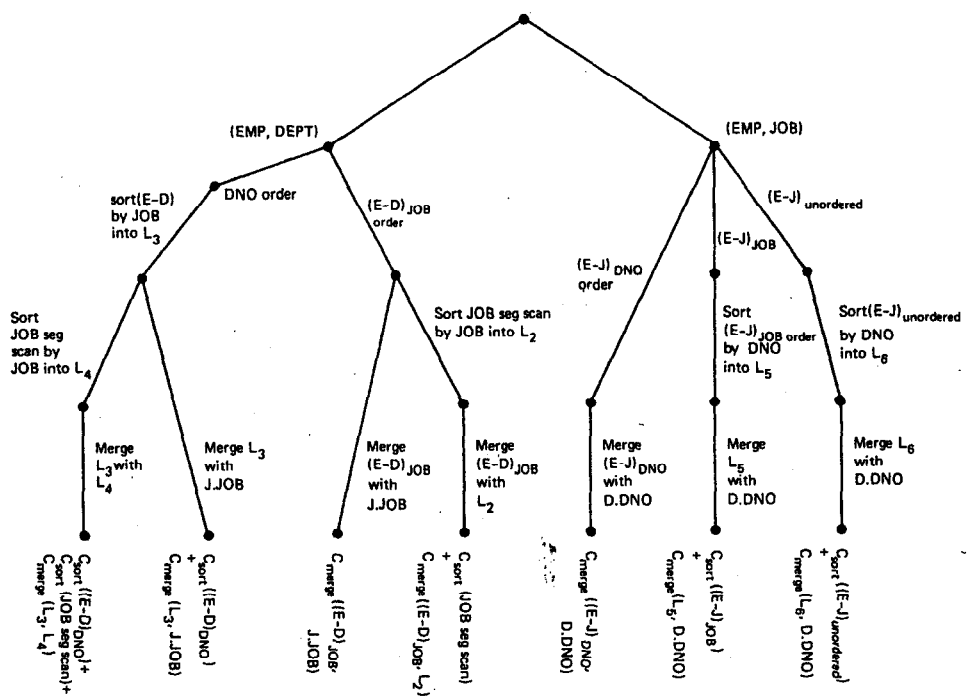

Figure 6. Extended search tree for third relation

6. Nested Queries

A query may appear as an operand of a predicate of the form "expression operator query". Such a query is called a Nested Query or a Subquery. If the operator is one of the six scalar comparisons (=, ¬=, >, >=, <, <=), then the subquery must return a single value. The following example using the "=" operator was given in section 2:

```
SELECT NAME
FROM EMPLOYEE
WHERE SALARY =
   (SELECT AVG(SALARY)
    FROM EMPLOYEE)
```

If the operator is IN or NOT IN then the subquery may return a set of values. For example:

```
SELECT NAME
FROM EMPLOYEE
WHERE DEPARTMENT_NUMBER IN
    (SELECT DEPARTMENT_NUMBER
     FROM DEPARTMENT
     WHERE LOCATION='DENVER')
```

In both examples, the subquery needs to be evaluated only once. The OPTIMIZER will arrange for the subquery to be evaluated before the top level query is evaluated. If a single value is returned, it is incorporated into the top level query as though it had been part of the original query statement; for example, if AVG(SAL) above evaluates to 15000 at execution time, then the predicate becomes "SALARY = 15000". If the subquery can return a set of values, they are returned in a temporary list, an internal form which is more efficient than a relation but which can only be accessed sequentially. In the example above, if the subquery returns the list (17,24) then the predicate is evaluated in a manner similar to the way in which it would have been evaluated if the original predicate had been DEPARTMENT_NUMBER IN (17,24).

A subquery may also contain a predicate with a subquery, down to a (theoretically) arbitrary level of nesting. When such subqueries do not reference columns from tables in higher level query blocks, they are all evaluated before the top level query is evaluated. In this case, the most deeply nested subqueries are evaluated first, since any subquery must be evaluated before its parent query can be evaluated.

A subquery may contain a reference to a value obtained from a candidate tuple of a higher level query block (see example below). Such a query is called a correlation subquery. A correlation subquery must in principle be re-evaluated for each candidate tuple from the referenced query block. This re-evaluation must be done before the correlation subquery's parent predicate in the higher level block can be tested for acceptance or rejection of the candidate tuple. As an example, consider the query:

```
SELECT NAME
FROM EMPLOYEE X
WHERE SALARY > (SELECT SALARY
                FROM EMPLOYEE
                WHERE EMPLOYEE_NUMBER=
                      X.MANAGER)
```

This selects names of EMPLOYEE's that earn more than their MANAGER. Here X identifies the query block and relation which furnishes the candidate tuple for the correlation. For each candidate tuple of the top level query block, the MANAGER value is used for evaluation of the subquery. The subquery result is then returned to the "SALARY >" predicate for testing acceptance of the candidate tuple.

If a correlation subquery is not directly below the query block it references but is separated from that block by one or more intermediate blocks, then the correlation subquery evaluation will be done before evaluation of the highest of the intermediate blocks. For example:

```
level 1   SELECT NAME
          FROM EMPLOYEE X
          WHERE SALARY >
level 2        (SELECT SALARY
                FROM EMPLOYEE
                WHERE EMPLOYEE_NUMBER =
level 3              (SELECT MANAGER
                      FROM EMPLOYEE
                      WHERE EMPLOYEE_NUMBER =
                            X.MANAGER))
```

This selects names of EMPLOYEE's that earn more than their MANAGER's MANAGER. As before, for each candidate tuple of the level-1 query block, the EMPLOYEE.MANAGER value is used for evaluation of the level-3 query block. In this case, because the level 3 subquery references a level 1 value but does not reference level 2 values, it is evaluated once for every new level 1 candidate tuple, but not for every level 2 candidate tuple.

If the value referenced by a correlation subquery (X.MANAGER above) is not unique in the set of candidate tuples (e.g., many employees have the same manager), the procedure given above will still cause the subquery to be re-evaluated for each occurrence of a replicated value. However, if the referenced relation is ordered on the referenced column, the re-evaluation can be made conditional, depending on a test of whether or not the current referenced value is the same as the one in the previous candidate tuple. If they are the same, the previous evaluation result can be used again. In some cases, it might even pay to sort the referenced relation on the referenced column in order to avoid re-evaluating subqueries unnecessarily. In order to determine whether or not the referenced column values are unique, the OPTIMIZER can use clues like NCARD > ICARD, where NCARD is the relation cardinality and ICARD is the index cardinality of an index on the referenced column.

7. Conclusion

The System R access path selection has been described for single table queries, joins, and nested queries. Evaluation work on comparing the choices made to the "right" choice is in progress, and will be described in a forthcoming paper. Preliminary results indicate that, although the costs predicted by the optimizer are often not accurate in absolute value, the true optimal path is selected in a large majority of cases. In many cases, the ordering among the estimated costs for all paths considered is precisely the same as that among the actual measured costs.

Furthermore, the cost of path selection is not overwhelming. For a two-way join, the cost of optimization is approximately equivalent to between 5 and 20 database retrievals. This number becomes even more insignificant when such a path selector is placed in an environment such as System R, where application programs are compiled once and run many times. The cost of optimization is amortized over many runs.

The key contributions of this path selector over other work in this area are the expanded use of statistics (index cardinality, for example), the inclusion of CPU utilization into the cost formulas, and the method of determining join order. Many queries are CPU-bound, particularly merge joins for which temporary relations are created and sorts performed. The concept of "selectivity factor" permits the optimizer to take advantage of as many of the query's restriction predicates as possible in the RSS search arguments and access paths. By remembering "interesting ordering" equivalence classes for joins and ORDER or GROUP specifications, the optimizer does more bookkeeping than most path selectors, but this additional work in many cases results in avoiding the storage and sorting of intermediate query results. Tree pruning and tree searching techniques allow this additional bookkeeping to be performed efficiently.

More work on validation of the optimizer cost formulas needs to be done, but we can conclude from this preliminary work that database management systems can support non-procedural query languages with performance comparable to those supporting the current more procedural languages.

Cited and General References

<1> Astrahan, M. M. et al. System R: Relational Approach to Database Management. ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

<2> Astrahan, M. M. et al. System R: A Relational Database Management System. To appear in Computer.

<3> Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indices. Acta Informatica, Vol. 1, 1972.

<4> Blasgen, M.W. and Eswaran, K.P. On the Evaluation of Queries in a Relational Data Base System. IBM Research Report RJ1745, April, 1976.

<5> Chamberlin, D.D., et al. SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, Vol. 20, No. 6, Nov. 1976, pp. 560-575.

<6> Chamberlin, D.D., Gray, J.N., and Traiger, I.L. Views, Authorization and Locking in a Relational Data Base System. ACM National Computer Conference Proceedings, 1975, pp. 425-430.

<7> Codd, E.F. A Relational Model of Data for Large Shared Data Banks. ACM Communications, Vol. 13, No. 6, June, 1970, pp. 377-387.

<8> Date, C.J. An Introduction to Data Base Systems, Addison-Wesley, 1975.

<9> Lorie, R.A. and Wade, B.W. The Compilation of a Very High Level Data Language. IBM Research Report RJ2008, May, 1977.

<10> Lorie, R.A. and Nilsson, J.F. An Access Specification Language for a Relational Data Base System. IBM Research Report RJ2218, April, 1978.

<11> Stonebraker, M.R., Wong, E., Kreps, P., and Held, G.D. The Design and Implementation of INGRES. ACM Trans. on Database Systems, Vol. 1, No. 3, September, 1976, pp. 189-222.

<12> Todd, S. PRTV: An Efficient Implementation for Large Relational Data Bases. Proc. International Conf. on Very Large Data Bases, Framingham, Mass., September, 1975.

<13> Wong, E., and Youssefi, K. Decomposition - A Strategy for Query Processing. ACM Transactions on Database Systems, Vol. 1, No. 3 (Sept. 1976) pp. 223-241.

<14> Zloof, M.M. Query by Example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431-437.