# Optimization of Nested Queries in a Distributed Relational Database

Guy M. Lohman†, Dean Daniels‡, Laura M. Haas†, Ruth Kistler†, Patricia G. Selinger†

†IBM Research Laboratory, San Jose, CA 95193
‡Carnegie-Mellon University, Pittsburg, PA 15213

ABSTRACT

This paper describes how nested queries in the SQL language are processed by R*, an experimental adaptation to the distributed environment of the well-known centralized relational DBMS, System R. Nested queries are queries in which a predicate references the result of another query block (SELECT...FROM...WHERE...), called a subquery block (subQB). SubQBs may themselves contain one or more subQBs. Depending upon whether a subQB references values in other query blocks, it is processed differently, as either an Evaluate-at-Open or Evaluate-at-Application subQB type. Three tasks comprise execution of each query block: initiation, evaluation, and application. When the query's tables are distributed among multiple sites, optimization of nested queries requires determining for each subQB: the site to perform each task, the protocols controlling interactions between those tasks, and the costs of each option, so that a minimal-cost plan can be chosen. R* optimizes each query block independently, "bottom up", using only the cost, cardinality, and result site of the subQB in the optimization of its containing query block.

## 1. INTRODUCTION

One of the principal advantages of relational query languages is that they are "closed" in the mathematical sense, i.e. that the result of a query against one or more relations (tables) is itself a relation and referenceable in another query. Some relational query languages, such as SQL [SQL] permit this nesting of one query within another as a single, unified query. This nesting of queries permits the specification of very complex queries in a structured way that aids understanding by the user and optimization of the nested queries as a unified whole. For example, the following query retrieves the names of all employees assigned to the (presumably unique) shipping department located in Denver:

```
SELECT E.NAME                                QB 1
FROM   EMPLOYEES E
WHERE  E.DEPT =                                      (A)
  (SELECT D.DEPT#                            QB 2
   FROM   DEPARTMENTS D
   WHERE  D.LOCATION = 'DENVER'
     AND  D.NAME = 'SHIPPING'  ); [1]
```

Each SELECT...FROM...WHERE... sequence is called a **query block** (hereafter abbreviated QB). Query blocks are numbered in the order in which they are specified, and are referred to by number throughout this paper. The QB within the parentheses in (A) above (QB 2) returns a single DEPT# of DEPARTMENTS that has LOCATION = 'DENVER' and NAME = 'SHIPPING', and this (single) value is then substituted as though it were a literal for the right-hand side of the E.DEPT predicate that will be used to test each instance of the EMPLOYEES table. While this same query may be expressed without nested queries as:

```
SELECT E.NAME                                QB 1
FROM   EMPLOYEES E, DEPARTMENTS D                    (B)
WHERE  E.DEPT = D.DEPT#
  AND  D.LOCATION = 'DENVER'
  AND  D.NAME = 'SHIPPING';
```

not all nested queries may be so transformed without changing the semantics that are implicit in the nested query structure, and it may not be optimal to do so. (e.g., evaluate-at-open QBs need be evaluated less often, as discussed in Section "2.2. Types of Nested Queries" below). In addition, the nested form is often easier for the user to formulate and to understand. Finally, we wished to maintain compatibility in R* with as much of System R's SQL as possible. It was for these reasons that we implemented nested queries in R*.

There has been very little discussion of nested queries in the literature on query optimization, and virtually no discussion on nested query optimization for distributed databases (e.g., [EPST 78], [YAO 79], [EPST 80], [NGUY 80], [BERN 81], [CHU 82], [KAMB 82], [KERS 82], [APER 83], [CHAN 83], [YU 83], etc.). Selinger [SELI 79] discusses the internal processing of nested queries in the (centralized) DBMS System R, particularly the order of execution of QBs within a nested query, and how values are passed between QBs. Only Kim [KIM 82] addresses the optimization of nested queries. His goal is to transform nested queries into a single QB involving joins, as was done in Examples (A) and (B) above, in order to utilize existing join optimization techniques for a single-site database. However, he



1 Table qualifiers are added to column names in the examples for clarity only: they are required in SQL only to resolve ambiguity. Similarly, indentation of QBs is not required but done here only for readability.

**Proceedings of the Tenth International Conference on Very Large Data Bases.**

**Singapore, August, 1984**

does not consider the consequences of these transformations when tables that are involved in the query are stored in different databases.

This paper discusses the optimization of *distributed* nested queries in the distributed database management system R*, a prototype distributed DBMS that is being developed and implemented at IBM San Jose Research Laboratory [HAAS 82, WILL 82, LOHM 84]. The terminology, types, and execution tasks of nested queries are introduced in Section 2. In Section 3, the problems of optimizing nested queries against a distributed database are contrasted with those of centralized databases. Section 4 provides a detailed description of how sites are selected for the three major tasks in nested query processing. The lower-level protocols between processes performing these tasks, and the cost estimates of each, are presented in Section 5. Section 6 suggests some further refinements to minimize the traffic between sites, for certain types of distributed nested queries. Section 7 discusses some difficulties encountered while implementing nested queries in R*. Section 8 concludes with a summary of the current status of, and future plans for, the R* optimizer.

To avoid lengthy reviews, we assume that the reader is familiar with the relational database model, the relational query language SQL [SQL], and System R [ASTR 76, BLAS 81, CHAM 81b]. As in System R, query compilation -- of which optimization is a part -- is essential to efficient R* execution of SQL queries that are embedded in an application program [CHAM 81a]. The R* optimizer has been adapted from the System R optimizer [SELI 79] to permit tables referenced in a single query to reside at distinct sites. And as in System R, for each QB in the query, the optimizer enumerates all "reasonable" combinations of alternative access paths to an individual table (e.g., via an index scan or relation scan) and join methods (i.e., nested-loop and merge join), incrementally adding tables and permuting the order in which tables are joined while avoiding joins that require a Cartesian product. Intermediate alternatives that produce identical results (i.e., the same set of tuples at the same site in the same order) are compared, and only the one with minimal cost is retained [LOHM 84]. In R*, costs have been extended to include a linear combination of the number of inter-site messages and the length of inter-site messages, as well as the System R cost components of CPU (number of instructions executed) and I/O (pages accessed) [DANI 82, LOHM 84]. Including site processing costs as well as inter-site communication costs has been justified based upon an analysis of the relative costs of actual communications, disk, and CPU resources consumed for typical queries [SELI 80], and distinguishes R* optimization from most other distributed database optimizers [LOHM 84].

Examples used to illustrate this paper are drawn from the following simplified database of three tables:

```
EMPLOYEES( EMP#,NAME,SAL,MGR,DEPT,HIREDATE,JOB# )
DEPARTMENTS( DEPT#,LOCATION,NAME,MANAGER )
JOBS( JOB#,TITLE )
```

Each employee has a (unique) identifying employee number, a name, salary, the employee number of his manager, the (unique) identifying number of his department, his date of hire, and the (unique) number of his job. Each department has a (unique) number, location, name, and the employee number of its manager. Each job classification has a (unique) number and name associated with it.

# 2. NESTED QUERIES

## 2.1. Nomenclature

The fundamental syntactic structure of the SQL query language is the query block (hereafter abbreviated as QB):

```
SELECT select-list
FROM   from-list
<WHERE  predicate>
```

The SELECT-list is a list of expressions involving columns of tables that are contained in the FROM-list, and the optional predicate contains one or more simple Boolean predicates. A **nested predicate** is a simple predicate which takes any one of the following three forms:

```
col-expr OP query-block
EXISTS     query-block
NOT EXISTS query-block
```

where "col-expr" is an expression involving zero or more columns, and OP may be a scalar comparison operator (=, ¬=, <, <=, >, >=), a set membership operator (IN, NOT IN), or a set comparison operator (OP ALL or OP ANY, where OP is a scalar comparison operator). Scalar comparison operators require that the query-block evaluates to a single value, whereas set operators allow the query-block to return one or more values [SQL]. A query containing one or more nested predicates is referred to as a **nested query**, and the QB in the predicate is called a **subquery block (subQB)** or **nested query block** of the containing or **parentQB** whose predicate formed the nesting.[2]

A subQB may itself have nested predicates, so that QBs may be nested to a (theoretically) arbitrary level. Also, a QB may have several subQBs, but only one parentQB. Hence the parentQB/subQB relationship forms a tree, the root of which is the first QB. The first QB and all of its descendant subQBs will be referred to collectively as one **query**.

## 2.2. Types of Nested Queries

If a subQB contains a reference in its predicate to a column from some table in the FROM-list of one of its ancestor QBs, then it is said to be **correlated** to that ancestor by that column. The subQB must be re-evaluated for each candidate value of the correlated column in the ancestor QB. A subQB that is correlated to its parentQB is referred to as an **evaluate-at-application (EAA)** subQB,[3] because the subQB must be newly evaluated for each application of the predicate in the parentQB that contains that subQB. Re-evaluation is necessary because the correlation changes the results of the subQB. This re-evaluation must be done before the correlated subQB's parentQB predicate can be tested for acceptance or rejection of the candidate tuple [SELI 79]. For example, consider the following query to retrieve all employees who work in Denver and are managers of their department:

2 Note that this use of the term "subquery" is different from that of [WONG 76] and [ONUE 83], in which a "subquery" may be any isolatable step in the execution of the query, i.e., that has at most one column in common with the rest of the query.

3 [SELI 79] and [SQL] refer to this as a "correlated subquery", but as we shall see in query (E) below, a QB must be correlated to its *parentQB* to be of this type.

```
SELECT E.NAME                          QB 1
FROM   EMPLOYEES E
WHERE  E.DEPT IN                                (C)
   (SELECT D,DEPT#                     QB 2
    FROM   DEPARTMENTS D
    WHERE  D.LOCATION = 'DENVER'
    AND    D.MANAGER = E.EMP#   );
```

For each candidate row of the EMPLOYEES table, the EMP# value is used for evaluating QB 2. The result is a set of DEPARTMENTS that is then used to test the "E.DEPT IN" predicate.

If, on the other hand, a subQB and all of its descendant QBs are not correlated to any tables in the FROM-list of its parentQB, then it is called an **evaluate-at-OPEN (EAO)** subQB, because absence of references to values from its parentQB's tables permit it to be evaluated at the time of OPEN (when it begins processing) of its parentQB or one of its ancestor QBs. For example, suppose the correlating predicate (D.MANAGER = E.EMP#) in query (C) above were removed:

```
SELECT E.NAME                          QB 1
FROM   EMPLOYEES E
WHERE  E.DEPT IN                                (D)
  (SELECT D.DEPT#                      QB 2
   FROM   DEPARTMENTS D
   WHERE  D.LOCATION = 'DENVER');
```

QB 2 (on DEPARTMENTS) can be evaluated *once* upon OPEN of QB 1 (on EMPLOYEES), and the resulting set of department numbers (DEPT#) can be stored temporarily for use in evaluating the nested predicate of QB 1 for each EMPLOYEE.

If a QB, J, is correlated to an ancestor QB other than its immediate parentQB, i.e. is separated from the QB to which it is correlated by one or more intermediate QBs, then QB J may be evaluated-at-OPEN of the "highest" (in terms of ancestry) of the intermediate QBs. For example:

```
SELECT E.NAME                          QB 1
FROM   EMPLOYEES E
WHERE  E.SAL >
  (SELECT AVG(F.SAL)                   QB 2
   FROM   EMPLOYEES F                                (E)
   WHERE  F.DEPT IN
     (SELECT D.DEPT#                   QB 3
      FROM   DEPARTMENTS D
      WHERE  D.LOCATION = 'DENVER'
      AND    D.MANAGER = E.EMP#   ) );
```

Here evaluation of the last QB (QB 3) requires a specific value for E.EMP# (from the EMPLOYEES table of QB 1). Since QB 3 contains no references to any tables in QB 2, it is evaluated once for each new QB 1 candidate tuple from EMPLOYEES, but not for every QB 2 candidate tuple from EMPLOYEES. So QB 3 is correlated to QB 1 by column E.EMP#, but is an EAO QB because it is evaluated at OPEN time of processing QB 2.

A QB, J, acquires the correlations of all of its subQBs, even if they are evaluated at QB J's OPEN, because those correlations change the value of some of QB J's predicates (those containing the correlated subQBs). If any of QB J's subQBs are correlated to QB J's parentQB, then QB J must be EAA. For example, in query (E) above, QB 2 is correlated to QB 1 because its child (QB 3), which is EAO of QB 2, must be re-evaluated for each new value of E.EMP#, and that may alter the right-hand-side of the "F.DEPT IN..." predicate. Hence QB 2 must be EAA. If we were to remove the correlating predicate, "D.MANAGER = E.EMP#", in QB 3, then both QB 2 and QB 3 would be EAO.

For more examples and a detailed discussion of how subQBs are processed in a centralized database, see [SELI 79].

## 2.3. Tasks in SubQB Execution

Processing any subQB requires three successive tasks:

1. **Initiation:** detects that a predicate is eligible to be evaluated; provides from ancestor QBs any values for correlated columns that are needed for that evaluation; and, for EAO QBs only, causes a temporary relation to be created on disk in which to store the set of results returned by the evaluation step.

2. **Evaluation:** uses the values provided by initiation to evaluate the subQB, and returns the results to the application step. Note that this may involve processing one or more subQB sequences.

3. **Application (Parent):** uses the results to apply the predicate to candidate tuples in the parentQB.

For EAO subQBs, initiation and evaluation are performed once (at OPEN of the QB at whose OPEN it can be evaluated, i.e., the parentQB or an ancestor QB), and application is repeated once per candidate tuple of the parentQB by simply reading the results from the temporary relation. For EAA QBs, initiation and application are always performed within the parentQB, and all three steps are repeated for each candidate tuple of the parentQB.

# 3. OPTIMIZATION OF NESTED QUERY BLOCKS

## 3.1. Dependencies Between Query Blocks

When all tables in the query are at a single site, optimization of a nested query is fairly simple, because optimization of a subQB does not depend upon that of its parentQB, and optimization of the parentQB depends upon that of its subQBs in a very limited way. Three attributes of a subQB might affect the cost of evaluating its parentQB: the subQB's estimated cost, its estimated cardinality, and the order of its result. The first, the subQB's cost, affects the parentQB's cost because the cost of a subQB's evaluation contributes to the cost of evaluating the parentQB. The second attribute, the subQB's cardinality, affects the parentQB's cost because each subQB predicate restricts which tuples satisfy the parentQB, so the estimated cardinality of the subQB is used to estimate the selectivity of the predicate containing that subQB. Thirdly, the order of a subQB is important because, if the column on the left-hand-side of the nested predicate is in a particular order, and the subQB is EAA, we could save re-evaluating the subQB each time the same column value occurred. However, System R (and the distributed database management system, R*) ignores the limited possible savings of enforcing such an ordering[4], so that the parentQB can be optimized once an estimate has been made for the cost and for the number of values returned by each of its subQBs. Thus, QBs in a query may be optimized independently, starting with the most nested QB and

4 See Section "6.1. Exploit ParentQB's Order" for more discussion of this decision.

working "bottom up" through the family tree of QBs, with each subQB passing to its parentQB an estimate of its cost and the number of values that it returns.

For a nested query in a *distributed* database, the major difficulty posed in optimizing it is that we can no longer optimize its QBs independently in a simple bottom-up fashion, passing up only each subQB's cost and cardinality. This is because the choice of sites at which the QB result is materialized introduces another dependency between QBs, *in addition to* the ones relevant in a centralized database. Specifically, the site to which a subQB delivers its result -- its **delivery site** -- must necessarily be the site to which the predicate containing the subQB is applied -- its **application site**. Since the predicate involves tables belonging to the parentQB, selection of the application site must be decided during optimization of the *parentQB*, and thus is not yet known when the *subQB* is being optimized. Ideally we would optimize QB 1 to deliver its result to the site to which the query was submitted (the **master site**), and its optimization would dictate the site at which each of its subQBs would be applied, and so on. However, this "top-down" processing conflicts with the "bottom-up" dependence of QB optimization upon the cardinality and cost estimates of its subQBs, as described in the previous paragraph.

R* resolves this "top down" versus "bottom up" dependency conflict by postponing selection of the delivery site of a subQB until its parentQB has been optimized. The optimizer essentially keeps its delivery site options open as it optimizes each QB, permitting it once again to optimize QBs bottom up. The result of optimizing each QB, S, is a number of candidate plans, each the best plan to deliver S's result to its "natural" result sites, plus a "to-be-determined-later" site (these are defined below). Then, during optimization of S's parentQB, P, each candidate plan for P determines an application site to which S's result should be delivered and uses the best plan for S to deliver its result to that application site. We say that the parentQB **resolves** the postponed decision on the delivery site of its subQB, once it chooses an application site in this way. The process of generating all possible site options, and selecting the best one for each QB, will be discussed in more detail in Section "4. Site Selections for Nested QBs".

## 3.2. Master vs. Apprentice Planning

When a query references objects such as tables or views at multiple sites, compilation (and optimization) involves multiple sites. In R*, the site to which the query is submitted and any site having a table or view involved in a query must participate in the planning process for that query as the master or an apprentice.

The site to which the query was originally presented, the master site, is responsible for developing a global plan containing all *inter-site* decisions, including the site at which all operations (especially, joins) are to be performed, the method of join, the order in which the tables/composites are joined, and the required order of the result [DANI 82, LOHM 84]. For nested queries, the master site also decides the site at which subQBs are to be applied (which dictates the site to which its results should be delivered) and initiated, as will be described in detail in Section "4. Site Selections for Nested QBs" below.

All other sites participating in the compilation of a query compile it as an **apprentice site**. Compilation by each apprentice site is initiated by receiving from the master site a copy of the global plan and the original query, which it re-parses. Each apprentice must (1) find the portions of the global plan that are relevant to it, and (2) develop a local plan for itself within the constraints imposed upon it by the global plan. It does this by traversing the global plan and re-creating its own version of the global plan, using the global plan for direction on inter-site matters and for making some shortcuts [DANI 82, LOHM 84]. For example, the apprentice uses the global plan to decide at which sites to apply subQBs, and thus the site at which each subQB is to deliver its result. So, unlike the master planner, the apprentice can choose the optimal plan for a QB to result at the dictated site only, and need not "keep its options open". This significantly streamlines apprentice planning.

An apprentice site is granted latitude in planning decisions affecting only its tables (e.g., alternative access paths, join methods, and permutations of *its* tables to join). The only decision involving nested queries over which the apprentice retains jurisdiction is deciding the order in which to apply multiple subQBs that the master dictates should be applied at that site, subject to the requirement that all correlated columns have values when their predicates are applied. Therefore, because the interesting decisions for QBs is at which site they are applied and initiated, we will be concerned only with master planning for the remainder of this paper.

# 4. SITE SELECTIONS FOR NESTED QBS

This section describes the master's procedure for determining at which site(s) to perform the three tasks -- initiation, evaluation, and application -- involved in executing each subQB in a query statement, in order to minimize the total cost of executing the entire query statement. In a distributed query, each of these three tasks may be executed at different sites, depending upon where the tables referenced in that query are stored, and where the master's optimizer chooses to join tables, apply predicates, etc. (see [DANI 82]). Each task may be executed by one or more processes, as follows:

1. **Initiation** usually requires only a single process, which we will call the **initiator**. In cases where the initiator's site is different from the applier's site (discussed below), initiation also requires some work to be done at the applier's site. The initiator spawns an agent process, called INIT', to perform this work (see Section "5. Evaluation Protocols and Costs" for more detailed protocols between sites).

2. **Evaluation** may require any number of processes. However, all of these processes are subordinate to a single, first process that OPENs processing of the QB and that receives the final results of the QB, i.e., the **result site** of that QB. This "chief process of the QB" is the only process with which the initiator and applier process interact, and will be referred to as the **evaluator** process.

3. **Application** never requires more than one process, called the **applier**.

We now discuss in more detail how the procedure chooses the sites for the evaluator, applier, and initiator processes, in that order.

## 4.1. Evaluator's Site Selection

The evaluator's (delivery) site is chosen from the "natural" result sites of that QB, plus a "to-be-determined-later" site, as described below.

### 4.1.1. Result Sites

To join two tables at different sites, R* generates candidate plans to perform the join at three different sites [DANI 82, LOHM 84]:

```
(J1) the inner table's site,
(J2) the outer table's site, and
(J3) an unknown site (or, more accurately,
     a "to-be-determined-later" site,
     designated here as "??".
```

To illustrate, consider the following distributed version of the query of example (B):

```
SELECT E.NAME
FROM   EMPLOYEES@SF E, DEPARTMENTS@NY D
WHERE  E.DEPT = D.DEPT#                    (F)
  AND  D.LOCATION = 'DENVER'
  AND  D.NAME = 'SHIPPING';
```

where table EMPLOYEES@SF is stored at site SF and table DEPARTMENTS@NY is stored at site NY.[5] When DEPARTMENTS@NY is the outer table and EMPLOYEES@SF is the inner table of the join in a candidate plan, then the three different site options for performing this join would be:

```
(J1) Ship DEPARTMENTS@NY to SF
     and join at SF (result site = SF).
(J2) Ship EMPLOYEES@SF to NY
     and join at NY (result site = NY).
(J3) Ship EMPLOYEES@SF and DEPARTMENTS@NY to
     some other site, ??, and join them there.
```

If this query originated in SF, option (J1) is likely to be optimal, whereas option (J2) would be favored by a query originating in NY. The third option is generated in case the query originated at some third site, say LA, or in case the result of the join is to be joined later with a third table at another site, perhaps KC. At the time EMPLOYEES@SF and DEPARTMENTS@NY are joined, however, the possible third sites are not yet known. So the third join site is left "to be determined later".

By generating all combinations of inner and outer tables, R* will generate a distinct optimal plan for a QB to result at each site having one or more tables in the FROM-list of that QB, or in the FROM-lists of its descendant QBs, plus site ??. We call sites other than ?? the **natural result sites** for a QB.

### 4.1.2. Resolving the Evaluation Site of a QB

Once the delivery site X of a subQB is chosen by the parentQB (as described earlier), we say that its evaluation site is **resolved** (to be X). From the set of all plans for the subQB, each of which result at one of the subQB's result sites as described above, R* picks the cheapest of the following three candidates for the best plan to deliver the subQB's result to X:

```
(P1) The best plan to result at site X.
     (This option is viable only if site
     X is a natural result site of the subQB.)
(P2) The best plan that results at ??,
     where ?? is replaced by X.
(P3) The best overall plan for the subQB,
     independent of result site, shipped
     from its natural result site, Y,
     to the dictated delivery site, X.
```

As discussed above, the delivery site for QB 1 is the site at which the query was presented (the master site), and the delivery site for any other QB is the application site chosen by its parentQB. We next discuss how this choice is made.

## 4.2. Application Site Selection

As each table in the parentQB is retrieved and joined to the outer table (or composite of outer tables), columns are added to the parentQB composite tuple that may be needed to apply the subQB's predicate. A predicate therefore becomes **eligible** for application only when the last table containing columns referenced by the predicate is joined to the composite as the inner table. So, for a given choice of a (possibly composite) outer table and an inner table, one or more subQB predicates may become eligible. The problem is at which site(s) should they be applied? Deciding this determines (resolves) the desired delivery site for that subQB.

### 4.2.1. Application Site Options

A subQB may ultimately be applied, i.e., used to evaluate the predicate of which it is a part, at sites other than its natural result sites. The application site is decided by the subQB's parentQB, which has to consider the interactions between sites at which its tables are located and possibly the natural result sites of its other subQBs. In this section, we discuss which sites are reasonable sites for the parentQB to consider applying nested predicates.

Consider the following variation of example query (F) having a subQB (QB 2), in which the two tables of QB 1 and the one table of QB 2 are each at different sites:

```
SELECT E.NAME                                QB 1
FROM   EMPLOYEES@SF E, DEPARTMENTS@NY D
WHERE  E.DEPT = D.DEPT#
  AND  D.LOCATION = 'DENVER'
  AND  D.NAME = 'SHIPPING'
  AND  E.JOB# =                                (G)
    (SELECT J.JOB#                           QB 2
     FROM   JOBS@LA
     WHERE  J.TITLE = 'CLERK');
```

This query requests the names of all employees that are clerks and work in the shipping department in Denver. Note that the predicate containing QB 2 becomes eligible with the addition of table EMPLOYEES@SF, because its column E.JOB# is referenced in that predicate.

For any given set of tables already joined as composite C and any given table T in the parentQB, J, the procedure considers three (possibly redundant) classes of application sites for a subQB K whose predicate becomes eligible for application only when table T is joined to the set:

(A1) Site of table T

(A2) Site of any partial or final computation for QB J (e.g., a join between outer composite C and inner table T, or delivery of QB J's final result, or application of another subQB to QB J).

(A3) SubQB K's possible result sites (natural + ??)

---

5 The site name embedded in the table name does *not* change, even if the table is moved to another site. The current store site of each table and its name agree here *only for expository reasons*. See [LIND 81] for a complete description of R* table-naming conventions.

with the following qualifications:

- Option (A1) is viable only if the nested predicate containing subQB K references only columns of table T (or none at all) and columns of tables in QBs that are ancestors of QB J. Table T could be any table (inner or outer) in a join. For example, the predicate containing QB 2 in example query (G) references only EMPLOYEES@SF, so option (A1) is viable when EMPLOYEES@SF is table T.

- Options (A2) and (A3) apply the predicate to the result of joining T to C (if C is not empty) before any final computations for ordering or grouping QB J's result.

- Option (A3), when applied to a plan for QB J with result site ??, will cause the ?? to be replaced by subQB K's result site (which might also be ??).

- Redundant application site options are not re-evaluated. For example, if the join site is the inner site (i.e., for option (J1)) and if the predicate references only the inner table, then options (A1) and (A2) are equivalent: only one of the two options need be evaluated. Similarly, subQB result sites that have already been covered by options (A1) or (A2) are redundant.

For our example query (G) above, when (again) table DEPARTMENTS@NY is the outer table and EMPLOYEES@SF is the inner table and our table T of interest, options (J1) through (J3) have the following sub-options for applying the predicate that contains QB 2:

(J1) Join site = SF (i.e., ship DEPARTMENTS@NY to SF)
- (A1) Apply subQB predicate to EMPLOYEES@SF before join
- (A2) Apply subQB predicate at SF (same as (A1))
- (A3) Apply subQB predicate at subQB's result sites =
  - (R1) LA (i.e., ship composite after join to LA, then apply)
  - (R2) ?? (i.e., ship composite after join to ??, then apply)

(J2) Join site = NY (i.e., ship EMPLOYEES@SF to NY to join)
- (A1) Apply subQB predicate to EMPLOYEES@SF before shipping it to NY for join
- (A2) Apply subQB predicate at join site = NY
- (A3) Apply subQB predicate at subQB's result sites =
  - (R1) LA (i.e., ship composite after join to LA, then apply)
  - (R2) ?? (i.e., ship composite after join to ??, then apply)

(J3) Join site = ?? (ship both DEPARTMENTS@NY and EMPLOYEES@SF to ??)
- (A1) Apply subQB predicate to EMPLOYEES@SF before shipping it to ?? to join
- (A2) Apply subQB predicate at join site = ?? (i.e., do every- thing at site ??)
- (A3) Apply subQB predicate at subQB's result sites =
  - (R1) LA (fill in ?? to be LA, i.e., do everything at LA)
  - (R2) ?? (redundant of option (J3)-(A2) above)

Option (J1)-(A2) is equivalent to (J1)-(A1) in this example because the predicate containing QB 2 references only the inner table (EMPLOYEES@SF), and we always apply predicates as soon as they become eligible (i.e., there is nothing to gain by waiting to apply the predicate to the composite after the join). Note also that option (J3)-(A3)-(R1) could be derived by setting ?? = LA in option (J3)-(A2).

The above options and suboptions are exhaustive and may all be reasonable possibilities, depending upon the sizes of the tables and the ratio of intra-site costs (CPU and I/O) to communications costs.

For this example, the natural result sites for each QB are as follows:

```
QB 2: LA
QB 1: NY, SF, LA
```

Note that optimization of QB 1 is likely to choose a delivery site for QB 2 that is not one of its natural result sites, in which case the ?? plan for QB 2 will be used.

### 4.2.2. Applying SubQB Predicates Simultaneously

Two subQBs that have the same parentQB *and* that reference one or more columns of the same table(s) in the ancestor QBs become eligible to be applied simultaneously. In such cases, order of application of these subQBs is unimportant, unless one subQB is sufficiently more restrictive than the other *and* we choose an option that ships the composite to both of the subQBs' (different) result sites. In that case, the more restrictive predicate is applied first to reduce the volume of data shipped between the two subQB result sites.

## 4.3. Initiation Site Selection

For EAA subQBs, selection of the applier's site necessarily selects the initiator's site, since by definition they must be the same. Thus, selection of the applier's site chooses a single plan for that subQB.

For EAO subQBs, however, the initiator and the applier may have different sites and even different QBs. The site at which an EAO subQB K is initiated is the delivery site of the QB at whose OPEN QB K is initiated. So, for any given plan for the initiating QB, its delivery site dictates the initiator's site for each EAO subQB, thus choosing a single, optimal plan for it.

## 4.4. Summary of Global Optimization

An overview of the algorithm for global optimization of distributed nested queries is given in Figure 1. The portions newly introduced for nested queries are shown in bold italics.

# 5. EVALUATION PROTOCOLS AND COSTS

The protocols between sites and the cost for evaluating a QB depends upon: (a) whether an individual subQB is EAA or EAO, and (b) in the latter case, whether the initiator's site and the applier's site are chosen to be the same. A discussion of the protocols between sites and the evaluation costs for each of the three possible cases follows.

**Figure 1: Algorithm for Optimizing Nested Queries in R***

```
FOR EACH query block, H (bottom up):

  Generate tree of possible subsets of tables of H
  FOR EACH subset of tables:

    Generate tree of possible plans
    FOR EACH possible plan:

      FOR set of ALL subQBs of H eligible simultaneously:

        Generate tree of subQB application sites
        FOR EACH set of subQB application sites:
          FOR EACH subQB J in the set:

            FOR EACH subQB K of QB J
              IF subQB K is Evaluated-at-OPEN of QB J
              THEN
                Resolve initiation site of subQB K
                Add cost of initiating and evaluating subQB K to cost of QB J
            Find best plan for subQB J to be delivered to dictated application
              site for subQB J (i.e., resolve application site for subQB J)
            Record best plan in plan for QB H
            Add costs of applying subQB J to costs of QB H
            IF subQB J is Evaluate-at-Application
            THEN add costs of initiating and evaluating subQB J to costs
                of QB H
        Find plan for QB H with a matching site & result order
        IF new plan cheaper
        THEN retain new plan

  Add sorts for GROUP BY, ORDER BY, etc., if needed
  Pick best plan resulting at each result site of QB H (natural and ??).

FOR TOP query block (QB 1):

  FOR EACH result site of QB 1 (i.e., for each plan for QB 1):

    FOR EACH subQB L in plan:
      IF subQB L is Evaluated-at-OPEN of QB 1
      THEN
        Resolve initiation site of subQB L to be master's site
        Add cost of initiating and evaluating subQB L to cost of QB 1

  Find best plan for delivery to master's site (resolve result site of QB 1)
  Distribute master plan to apprentices
  Generate access structures to perform plan (see [LOHM 84]).
```

## 5.1. Evaluate-at-Application SubQB

This is the simplest case, because initiation and application always take place at the same point in the parentQB, and therefore at the same site, with only evaluation of the subQB intervening (refer to Figure 2 and Figure 3). So both the initiator and applier processes

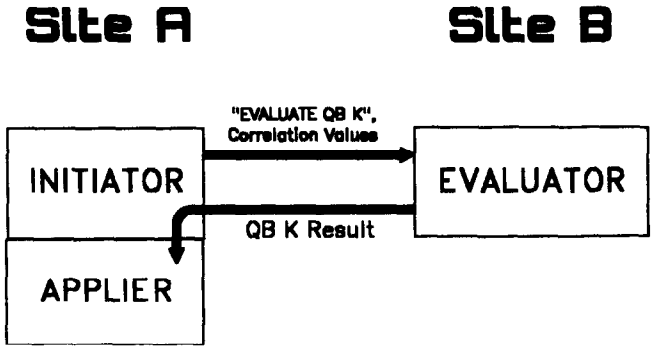


**Figure 2: Evaluate-at-Application (EAA) subQB.**

**Figure 3: Procedure for Evaluate-at-Application (EAA) subQB.**

```
FOR EACH tuple of QB K's parentQB, QB J:

  (1) Initiation: Initiator initiates Evaluator of QB K at Site B:

    (1.1) Initiator detects eligibility of QB K's predicate
    (1.2) Initiator sends values of correlated variables
          to Evaluator

  (2) Evaluation: Evaluator evaluates QB K and
                  returns results to Initiator at Site A,
                  who passes them to Applier.

  (3) Application: Applier applies results immediately
                   in predicate in QB J that contains QB K

Note: Site A and Site B may be the same.
```

belong to the parentQB, QB J. For each candidate tuple in the parentQB J, site A sends to the evaluator at site B the values of all correlated columns for that tuple, and prepares to receive back the result. The evaluator receives the correlation values, evaluates the subQB K, and returns a single-valued result to site A. EAA QBs that might return multiple values (those QBs having a set membership or set comparison OP in the predicate containing that QB) can be transformed during optimization into an equivalent QB that returns only a single value -- EXISTS or NOT EXISTS (see Section 6.5 below). The initiator receives the result and passes it to the applier, which applies the results in the predicate and either accepts or rejects the candidate tuple.

The estimated cardinality and cost of evaluating QB K and delivering it to the applier at site A for each tuple of the parentQB (J) is obtained by dictating that QB K deliver its result to site A. This resolves the evaluator's site for QB K, yielding an optimal plan for QB K. Its cost, multiplied by the estimated cardinality of QB J, gives the total cost to apply this subQB in QB J. This total cost is added into the cost associated with all candidate plans for QB J that dictate delivering QB K to site A.

## 5.2. Evaluate-at-Open SubQB, Applier's Site = Initiator's Site

EAO subQBs are initiated once, evaluated once at that time, and then sometime later applied repeatedly, once for each candidate tuple to which the predicate applies (refer to Figure 4 and Figure 5). In fact, as we saw in example query (E) above, initiation and application may (but need not) take place in different QBs. So the initiator process and applier process may belong to different QBs. For this case, however, we assume that the initiator and applier are at the same *site* (site A), even if they belong to different QBs. Recall that the initiator's site is defined to be the delivery site of the QB at whose OPEN the subQB is initiated.

At OPEN of the initiating query block, QB I, the initiator starts evaluation of QB K at site B by the evaluator, and receives back the results, which the initiator stores in a temporary relation at site A. Later, when the predicate containing QB K is encountered and needs to be applied, the applier process retrieves the results from the temporary relation to use in applying the predicate, once for each candidate tuple.

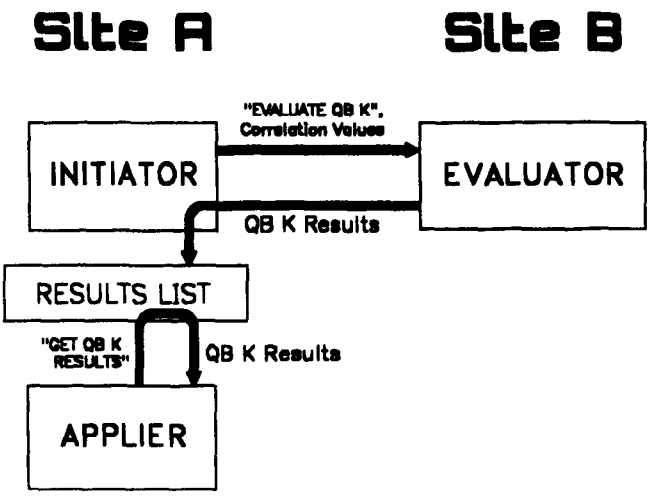


**Figure 4: Evaluate-at-OPEN (EAO) subQB, Initiator's site = Applier's site.**

The cost of evaluating QB K and of shipping correlation values from A to B and subQB K's results from B to A is incurred only once: a significant savings. However, there are additional costs at site A for storing the result in the temporary relation and for reading from that relation each time. And because a temporary relation is not indexable in R*, retrieval from it cannot be in order or subsetted by a predicate without scanning the relation until the desired value is found. For purposes of optimization, the applier (QB J) is assessed only the cost of reading one half of the temporary table (on the average) N times, where N is the cardinality of QB J. The cost of initiating and evaluating QB K, and writing its results in the temporary relation at site A, is assessed to the initiator (QB I), not the applier. The reason that this cost assessment is different from that for EAA subQBs will become apparent when we contrast this case with case 3 below.

## 5.3. Evaluate-at-Open SubQB, Applier's Site ≠ Initiator's Site

It may happen that the initiator's site of an EAO query is not the same as the applier's site, in which case two sites are involved in initiation, as shown in Figure 6. Here site C is responsible for initiating the evaluation of QB K, but will not receive the results. So steps (1.2) and (1.3) of case 2 must be augmented into steps (1.2) through (1.4), as shown in Figure 7. The initiator INIT spawns an agent process at site A, called INIT′ in Figure 6 and Figure 7, which performs the usual functions of the initiator described in the second case. In addition, INIT′ must "close the loop" of communications that INIT started, by acknowledging the successful completion of evaluation (step (2.4)) to INIT at Site C.

**Figure 5: Procedure for Evaluate-at-OPEN (EAO) subQB, Initiator's site = Applier's site.**

(1) Initiation: Initiator initiates Evaluator at Site B:

- (1.1) Initiator's OPEN triggers initiation of QB K processing
- (1.2) Initiator creates a temporary relation to accept result(s) of QB K
- (1.3) Initiator sends values of correlated variables, if any, to Evaluator at Site B

(2) Evaluation: Evaluator at Site B evaluates QB K:

- (2.1) Evaluator evaluates QB K
- (2.2) Evaluator returns result(s) to Initiator
- (2.3) Initiator stores result(s) in temporary relation created in Step (1.2), enabling Application at a later time

(3) Application: FOR EACH tuple in QB K's parentQB, QB J, the predicate containing QB K is applied:

- (3.1) Applier detects eligibility of QB K's predicate
- (3.2) Applier reads QB K result(s) from temporary relation
- (3.3) Applier applies predicate to its candidate tuple

**Note:**

- Site A and Site B may be the same.
- Initiator must be the first computation within its QB

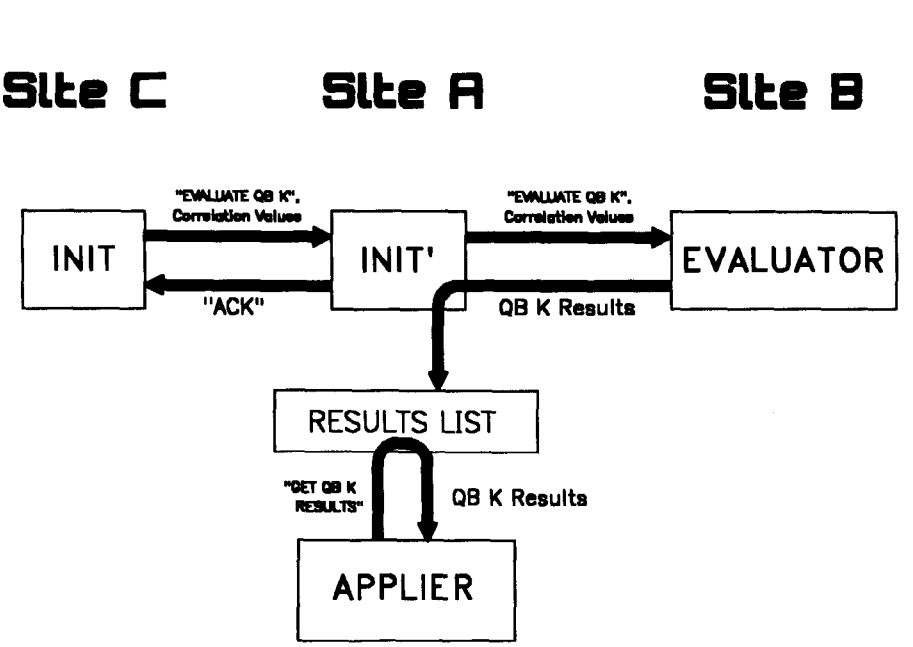


**Figure 6: Evaluate-at-OPEN (EAO) subQB, Initiator's site ≠ Applier's site.**

As in case 2, the costs of initiation and evaluation are assessed to the initiating QB, but in this case there are additional communications costs for procedure steps (1.2) and (2.4).

**Figure 7: Procedure for Evaluate-at-OPEN (EAO) subQB, Initiator's site ≠ Applier's site.**

(1) Initiation: Initiator at Site C (INIT) initiates QB K at Site B:

- (1.1) INIT's OPEN triggers initiation of QB K
- (1.2) INIT sends values of correlated variables, if any, to an agent process, INIT′, that it spawns at Site A
- (1.3) INIT′ creates temporary relation to accept result(s) of QB K
- (1.4) INIT′ forwards values of correlated variables, if any, to Evaluator at Site B

(2) Evaluation: Evaluator at Site B evaluates QB K:

- (2.1) Evaluator evaluates QB K
- (2.2) Evaluator returns result(s) to INIT′ at Site A
- (2.3) INIT′ stores result(s) in temporary relation that was created in Step (1.3)
- (2.4) INIT′ acknowledges proper completion to INIT at Site C, enabling Application at a later time

(3) Application: FOR EACH tuple in QB J, predicate containing QB K is applied:

- (3.1) Applier detects eligibility of QB K's predicate
- (3.2) Applier reads QB K result(s) from temporary relation
- (3.3) Applier applies predicate to its candidate tuple

**Note:**

- Only Steps (1.2),(1.4), and (2.4) are different from case of Figure 4.
- Site A and Site B may be the same.
- When Site C = Site B, see previous case.
- Initiator must be the first computation within its QB

One difficulty with EAO QBs is that the optimizer does not know whether case 2 or case 3 will happen until a delivery site is known for QB I, the initiating QB. Hence the plan for an EAO QB cannot be finalized until that QB's *initiating* QB -- not applying QB -- is optimized. Internally, the optimizer keeps different plans for both cases 2 and 3, with costs for each, so that when it optimizes QB I it can try both alternatives and choose the one having minimal cost.

## 6. STREAMLINING EAA SUBQUERY BLOCKS

When the initiator and evaluator for an EAA subQB are at different sites, a great deal of inefficient communication traffic is generated: one exchange of correlation values and subQB results per parentQB tuple (see Figure 2). This section presents several possible ways to reduce this traffic, using the following distributed query that contains an EAA subQB:

```
SELECT E.NAME                          QB 1
FROM   EMPLOYEES@SF E
WHERE  E.DEPT =                                  (H)
    (SELECT D.DEPT#                    QB 2
     FROM   DEPARTMENTS@NY D
     WHERE D.LOCATION = 'DENVER'
       AND D.NAME = 'SHIPPING'
       AND D.MANAGER = E.EMP#    );
```

### 6.1. Exploit ParentQB's Order

As suggested in Section "3.1. Dependencies Between Query Blocks", when the values for both the column in the nested predicate (E.DEPT in example (H)) *and* the correlation values of QB 1 (E.EMP#) are repeated in the next tuple, the evaluation of QB 2 will have the same result. Therefore, redundant evaluations as well as the communications to and from the initiator could be saved if the parentQB were ordered on those columns, so that duplicate values of those columns could be detected and redundant evaluation avoided. While R* in fact does exploit duplicate correlation values when it detects them, it does not *plan* it that way. The optimizer could trade off the estimated savings of enforcing a particular order for the parentQB

against the higher costs of either sorting the composite or using an index scan (if one existed on the predicate and correlation columns) to achieve that order. The problem is that, given the statistics currently maintained for each table, it is very difficult to estimate reliably the number of tuples having duplicate values of just the predicate and correlation columns, and thus the potential savings. For this reason, the R* optimizer does not consider this option.

## 6.2. Transform Correlated QBs to Join

Kim has noted that EAA subQBs may be transformed to joins under certain circumstances [KIM 82]. For example, query (H) can be transformed to a single QB:

```
SELECT E.NAME
FROM   EMPLOYEES@SF E, DEPARTMENTS@NY
WHERE  E.DEPT = D.DEPT#                          (H1)
  AND  D.LOCATION = 'DENVER'
  AND  D.NAME = 'SHIPPING'
  AND  D.MANAGER = E.EMP#;
```

The benefit of such a transformation is that the optimizer is given more leeway in deciding how the query is to be processed. By placing the DEPARTMENTS@NY table in a subQB in query (H), the user effectively undermines the optimizer's function by *dictating* to the optimizer that a nested loop join with EMPLOYEES@SF as the outer table will be done. As transformed in (H1), existing mechanisms within the optimizer could evaluate alternative join methods (possibly exploiting orderings of both tables to do a merge scan join), storing the inner table at the join site to save re-shipping it, and using either EMPLOYEES@SF or DEPARTMENTS@NY as the outer table.

The difficulty is that (H1) is semantically not equivalent to (H) when the values of the SELECT-column of the subQB (D.DEPT) are not unique. As stated in Section "2.1. Nomenclature", the "=" operator in a nested predicate implies that a unique value is expected from the evaluation of the subQB, but there is no such implication in a join predicate. Furthermore, not all EAA subQBs may be so transformed. For example, in example (C) QB 2 cannot be transformed to a join with its parentQB because its nested predicate contains a set operator (...E.DEPT *IN*...) that is not expressible as a join with equivalent semantics when the subQB returns duplicate values matching E.DEPT: the nested query form would result in only one qualifying tuple, but the join form would result in one tuple per matching value found.

## 6.3. Move Correlations into Nesting Predicate

This transformation converts EAA subQBs to EAO subQBs by moving all correlation predicates into the nesting predicate, which now must permit more than one column to be specified. Query (H) thus becomes:

```
SELECT E.NAME                                QB 1
FROM   EMPLOYEES@SF E
WHERE  (E.DEPT, E.EMP#) =                          (H2)
    (SELECT D.DEPT#, D.MANAGER               QB 2
     FROM   DEPARTMENTS@NY D
     WHERE D.LOCATION = 'DENVER'
       AND D.NAME = 'SHIPPING');
```

Since QB 2 is now EAO, it need be evaluated only once, instead of once per tuple of EMPLOYEES@SF. On the other hand, the subQB's result in (H2) could be much bigger without the correlation predicate. So we might end up shipping *more* data from NY to SF *with* the transformation, but we save the multiple evaluations of the subQB and the communications overhead of a conversation between SF and NY for each new value of EMPLOYEES@SF. And as with the previous transformation, not all EAA subQBs can be transformed: for example, we cannot move the correlation predicate in query (C) because its scalar operator does not match the set operator of the nesting predicate. For this reason, and because allowing multiple columns in the nested predicate required significant changes to the semantics of SQL and the implementation of subQBs in R*, this transformation was also shelved.

## 6.4. Ship Subset of ParentQB to SubQB

Inspired by semijoins, this strategy still ships a subset of an entire table only once, but is the opposite of the previous transformation in that the parentQB -- not the subQB -- is the table to be shipped. For example, in query (H) we would ship EMPLOYEES@SF, projected on columns DEPT and EMP# (and restricted if there were other predicates on EMPLOYEES@SF), once to NY. At NY, each tuple in the subset would be re-read, used to evaluate the subQB, and either accepted or rejected by applying the nested predicate there. While this again reduces the number of conversations between SF and NY, it does not reduce the number of times the subQB is evaluated and has the added cost of storing and re-reading the subset of the parentQB if it does not fit in a buffer.

Note that the last two transformations would effectively be considered as alternative join options (shipping the inner table to the site of the outer, or *vice versa*) if Kim's transformation to a join predicate were done.

## 6.5. Return Only Single SubQB Result

The R* optimizer transforms the internal (parse-tree) representation of an EAA SubQB that may return a set of values into that of an equivalent subQB that returns exactly one value: FOUND or NOT FOUND. For example, when the nested predicate operator of example query (H) contains a set operator (IN) instead of a scalar operator (=):

```
SELECT E.NAME                                QB 1
FROM   EMPLOYEES@SF E
WHERE  E.DEPT IN                                    (I)
    (SELECT D.DEPT#                          QB 2
     FROM   DEPARTMENTS@NY D
     WHERE D.LOCATION = 'DENVER'
       AND D.NAME = 'SHIPPING'
       AND D.MANAGER = E.EMP#   );
```

it is modified to appear as:

```
SELECT E.NAME                                QB 1
FROM   EMPLOYEES@SF E
WHERE  E.DEPT IS NOT NULL                           (I1)
  AND EXISTS
    (SELECT <any columns>                    QB 2
     FROM   DEPARTMENTS@NY D
     WHERE D.LOCATION = 'DENVER'
       AND D.NAME = 'SHIPPING'
       AND D.MANAGER = E.EMP#
       AND E.DEPT = D.DEPT#   );
```

Because QB 2 returns only a single value rather than a potentially

long set of values, communications volume may be substantially reduced. Evaluation of QB 2 may be cut short as soon as it finds a qualifying tuple in (I1), whereas in the original query the subQB had no hint from its nesting predicate that it need not scan all of DEPARTMENTS@NY to ensure that the subQB had a unique value. Finally, this transformation saved us having to implement new mechanisms for looping through multiple values of the right-hand-side of a predicate, looking for a match, for this type of predicate only.

## 7. IMPLEMENTATION CONSIDERATIONS

The extensions to R* to properly plan and execute nested queries required adding code to the optimizer: (1) to retain optimal plans for each natural result site of each QB plus the "unknown" site, rather than just the best overall plan for the QB; (2) to generate the application site alternatives ; (3) to evaluate the three possible cases for subQB execution; and (4) to determine for each apprentice those portions of the global plan pertaining to it. In addition, the run-time routines to execute the directives of the chosen plan had to be coded.

The additional bookkeeping required to keep track of optimal plans for each natural result site of each QB significantly increased the storage required and the code complexity. Because option (A3) adds a subQB's natural result sites to those of its parentQB (as described in Section "4.2.1. Application Site Options" above), the number of alternative application sites are compounded as the optimizer works its way up to the top QB. So the number of candidate plans may grow exponentially with the number of tables, if each table is at a different site *and* all tables are joined via a nested predicate. This problem was graphically illustrated by the simple example in Section "4.2.1. Application Site Options" (query (G)), which generated 10 different plans for joining two tables and applying one subQB, for a single join order, join method, and access method for each table. When the R* optimizer considers all feasible combinations of merge scan joins, alternative inner table transfer strategies (store the inner table at the join site versus re-fetch as needed), and EMPLOYEES as the outer table of the join (see [DANI 82] and [LOHM 84]), the costs of 48 different plans are evaluated. This is three times the number when no subquery is to be applied, and ten times the number of plans considered when all tables are at the same site. When the partial plans for single tables are included, the distributed query of example (G) considers 64 partial or complete plans. Each plan is composed of a "miniplan" for each table, inter-site transfer, or subQB application in the plan.

The plan storage problem has been somewhat ameliorated by re-structuring the optimizer to store a plan only if it improves upon an existing plan to produce the same result (same tuples) in the same order at the same site, or if it provides a new result, order, or result site. Previously, all plans were allocated space that was not reclaimed even when the plan was later found to be dominated by another plan. Although evaluating all of the alternative plans sounds costly, in practice it is not noticeable to an interactive user, and for each alternative that is considered, one can construct cases in which it significantly out-performs all other alternatives. In addition, practical limits on the number of tables, and the inability of users to formulate queries that are nested sufficiently deep, should prevent the space and time demands from becoming too severe.

## 8. CONCLUSIONS

We have presented in this paper the problems associated with optimizing nested queries in a distributed relational database management system, and the approach used by R* to solve these problems. In particular, R* postpones determining the result site of a subQB until it optimizes its parentQB (applying QB), or, in the case of evaluate-at-open subQBs, its initiating QB. We have itemized the cases possible for both evaluate-at-application and evaluate-at-open types of QBs, and discussed algorithms for processing each case. In addition, we have presented the breakdown of work among sites cooperating in a distributed nested query, and how agreement upon plans for executing the query is coordinated by a master site while allowing the other (apprentice) sites a degree of site autonomy in matters pertaining to their own tables.

Nested queries for distributed databases have been implemented in R*, and have been tested for correct execution of complex queries involving up to five nested query blocks. To do this, we developed a test program that automatically varies the location of tables using SYNONYMs and tests the results of the queries against a table of expected results.

In the near future we intend to evaluate and validate the performance of the R* extensions to the System R optimizer, much as was done for System R by [ASTR 80], and in more detail than was done for Distributed INGRES [STON 82]. We also hope to improve the cost formulas to better conform to reality, and to isolate and improve situations for which bad plans are chosen or performance is anomalous. Sensitivity analyses will aid us in deciding where to concentrate improved detail in our modeling, and where the model can be simplified. Perhaps some heuristics or "rules of thumb" can be gleaned from this empirical analysis, which should help us to prune our enumeration tree of alternative plans further, without risking the omission of a truly optimal plan.

## 9. ACKNOWLEDGEMENTS

The authors are indebted to Morton Astrahan, Shel Finkelstein, Won Kim, Bruce Lindsay, Paul Wilms, and Bob Yost, who carefully and constructively reviewed an earlier draft of this paper. We also wish to acknowledge members of the R* project who also suggested some of the transformations of Section "6. Streamlining EAA Subquery Blocks": Elisa Bertino suggested the transformation of Section "6.3. Move Correlations into Nesting Predicate", and C. Mohan suggested that of Section "6.4. Ship Subset of ParentQB to SubQB".

## 10. REFERENCES

[APER 83] P.M.G. Apers, A.R. Hevner, and S.B. Yao, "Optimizing Algorithms for Distributed Queries", *IEEE Transactions on Software Engineering, SE-9,* 1 (January 1983), pp. 57-68.

[ASTR 76] M.M. Astrahan et al., "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems 1,* 2 (June 1976), pp. 97-137.

[ASTR 80] M.M. Astrahan, W. Kim, and M. Schkolnick, "Evaluation of the System R Access Path Selection Mechanism", *Proc. of 1980 IFIP Congress*, North-Holland Publishing, 1980. Also available as IBM Research Laboratory RJ2797, San Jose, Calif., April 1980.

[BERN 81b] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems 6*, 4 (December 1981), pp. 602-625.

[BLAS 81] M.W. Blasgen, M.M. Astrahan, D.D. Chamberlin, J.N. Gray, W.F. King, B.G. Lindsay, R.A. Lorie, J.W. Mehl, T.G. Price, G.R. Putzolu, M. Schkolnick, P.G. Selinger, D.R. Slutz, H.R. Strong, I.L. Traiger, B.W. Wade, and R.A. Yost, "System R: An Architectural Overview", *IBM Systems Journal 20*, 1 (1981), pp. 41-62.

[CHAM 81a] D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnick, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost, "Support for Repetitive Transactions and Ad Hoc Queries in System R", *ACM Transactions on Database Systems 6*, 1 (March 1981). Also available as IBM Research Laboratory RJ2551, San Jose, Calif., May 1979.

[CHAM 81b] D.D. Chamberlin, M.M. Astrahan, M.W. Blasgen, J.N. Gray, W.F. King, B.G. Lindsay, R. Lorie, J.W. Mehl, T.G. Price, F. Putzolu, P.G. Selinger, M. Schkolnick, D.R. Slutz, I.L. Traiger, B.W. Wade, R.A. Yost, "A History and Evaluation of System R", *Communications of the ACM, 24*, 10 (October 1981). Also available as IBM Research Laboratory RJ2843, San Jose, Calif., June 1980.

[CHAN 82] J-M. Chang, "A Heuristic Approach to Distributed Query Processing", *Procs. of the Eighth International Conference on Very Large Data Bases*, Mexico City, VLDB Endowment, Saratoga, CA, September 1982, pp. 54-61.

[CHAN 83] A. Chan, U. Dayal, S. Fox, N. Goodman, D. Ries, D. Skeen, "Overview of An ADA Compatible Distributed Database Manager", *Proc. SIGMOD 83*, May 1983.

[CHU 82] W.W. Chu and P. Hurley, "Optimal Query Processing for Distributed Database Systems", *IEEE Trans. on Computers, C-31*, 9 (September 1982), pp. 835-850.

[DANI 82] D. Daniels, P.G. Selinger, L.M. Haas, B.G. Lindsay, C. Mohan, A. Walker, P. Wilms. "An Introduction to Distributed Query Compilation in R*", *Proceedings Second International Conference on Distributed Databases*, Berlin, Sept. 1982. Also available as IBM Research Laboratory RJ3497, San Jose, Calif., June 1982.

[EPST 78] R. Epstein, M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Data Base System", *Proceedings of ACM-SIGMOD*, Austin, TX, May 1978, pp. 169-180.

[EPST 80] R. Epstein and M. Stonebraker, "Analysis of Distributed Data Base Processing Strategies", *Procs. of the Sixth International Conference on Very Large Data Bases*, Montreal, IEEE, October 1980, pp. 92-101.

[HAAS 82] L. M. Haas, P. Selinger, E. Bertino, D. Daniels, B. Lindsay, G. Lohman, Y. Masunaga, C.Mohan, P. Ng, P. Wilms and R. Yost, "R*: A Research Project on Distributed Relational DBMS", *Database Engineering 5*, 4 (Dec. 1982), IEEE Computer Society, pp. 28-32.

[HEVN 79] A.R. Hevner and S.B. Yao, "Query Processing in Distributed Database Systems", *IEEE Trans. Software Engr., SE-5*, (May 1979), pp. 177-187.

[KIM 82] W. Kim, "On Optimizing an SQL-like Nested Query", *ACM Transactions on Database Systems 7*, 3 (September 1982), pp. 443-469. Also available as IBM Research Laboratory RJ3063, San Jose, Calif., February 1981.

[KERS 82] L. Kerschberg, P.D. Ting, S.B. Yao, "Query Optimization in Star Computer Networks", *ACM Transactions on Database Systems 7*,4 (December 1982).

[LIND 81] B. G. Lindsay, "Object Naming and Catalog Management for a Distributed Database Manager", *Proceedings 2nd International Conference on Distributed Computing Systems*, Paris, France, April 1981. Also available as IBM Research Laboratory RJ2914, San Jose, Calif., August 1980.

[LOHM 84] G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, "Query Processing in R*", *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory (Eds.), Springer-Verlag, to appear in 1984.

[NGUY 80] G.T. Nguyen, "Decentralized Dynamic Query Decomposition for Distributed Database Systems", *Proc. ACM Pacific '80: Distributed Processing - New Directions for a New Decade*, San Francisco, November 1980.

[ONUE 83] E. Onuegbe, S. Rahimi, and A.R. Hevner, "Local Query Translation and Optimization in a Distributed System", *Proceedings NCC 1983*, July, 1983, pp. 229-239.

[SELI 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price, "Access Path Selection in a Relational Database Management System", *Proceedings of ACM-SIGMOD*, May 1979. Also available as IBM Research Laboratory RJ2429, San Jose, Calif., August 1979.

[SELI 80] P. G. Selinger and M. Adiba, "Access Path Selection in Distributed Database Management Systems", *Proceedings International Conference on Data Bases*, ed. Deen and Hammersly, University of Aberdeen,

July 1980, pp. 204-215. Also available as IBM Research Laboratory RJ2883, San Jose, Calif., August 1980.

[SQL] IBM Corporation, SQL/Data System Concepts and Facilities, IBM Form No. GH24-5013.

[STON 82] M. Stonebraker, J. Woodfill, J. Ranstrom, M. Murphy, J. Kalash, M. Carey, K. Arnold, "Performance Analysis of Distributed Data Base Systems", *Database Engineering 5*, 4 (Dec. 1982), IEEE Computer Society, pp. 58-65.

[WILL 82] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermark, P. Selinger, A. Walker, P. Wilms, R. Yost, "R*: An Overview of the Architecture", *Improving Database Usability and Responsiveness*, (Procs. International Conference on Databases, Jerusalem, Israel, June 1982), P. Scheuermann, ed., Academic Press, New York, 1982, pp. 1-27. Also available as IBM Research Laboratory RJ3325, San Jose, Calif., December 1981.

[WONG 76] E. Wong and K. Youssefi, "Decomposition -- a Strategy for Query Processing", *ACM Transactions on Database Systems 1*, 3 (September 1976), pp. 223-241.

[YAO 79] S.B. Yao, "Optimization of Query Algorithms", *ACM Transactions on Database Systems 4*, 2 (June 1979), pp. 133-155.

[YU 83] C.T. Yu, and C.C. Chang, "On the Design of a Query Processing Strategy in a Distributed Database Environment", *Proc. SIGMOD 83*, May 1983.