Measuring the Complexity of Join Enumeration in Query Optimization

Kiyoshi Ono[1], Guy M. Lohman

IBM Almaden Research Center
K55/801, 650 Harry Road
San Jose, CA 95120

Abstract

Since relational database management systems typically support only *diadic* join operators as primitive operations, a query optimizer must choose the "best" sequence of two-way joins to achieve the N-way join of tables requested by a query. The computational complexity of this optimization process is dominated by the number of such possible sequences that must be evaluated by the optimizer. This paper describes and measures the performance of the Starburst join enumerator, which can parameterically adjust for each query the space of join sequences that are evaluated by the optimizer to allow or disallow (1) composite tables (i.e., tables that are themselves the result of a join) as the inner operand of a join and (2) joins between two tables having no join predicate linking them (i.e., Cartesian products). To limit the size of their optimizer's search space, most earlier systems excluded both of these types of plans, which can execute significantly faster for some queries. By experimentally varying the parameters of the Starburst join enumerator, we have validated analytic formulas for the number of join sequences under a variety of conditions, and have proven their dependence upon the "shape" of the query. Specifically, "linear" queries, in which tables are connected by binary predicates in a straight line, can be optimized in polynomial time. Hence the dynamic programming techniques of System R and R* can still be used to optimize linear queries of as many as 100 tables in a reasonable amount of time!

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

Introduction

A query optimizer in a relational DBMS translates non-procedural queries into a procedural plan for execution, typically by generating many alternative plans, estimating the execution cost of each, and choosing the plan having the lowest estimated cost. Increasing this set of ***feasible plans*** that it evaluates improves the chances — but does not guarantee! — that it will find a better plan, while increasing the (compile-time) cost for it to optimize the query. A major challenge in the design of a query optimizer is to ensure that the set of feasible plans contains efficient plans without making the set too big to be generated practically.

Problem: Which join sequences to evaluate?

One of the major decisions an optimizer must make is the order in which to join the tables referenced in the query. Since the join operation is implemented in most systems as a diadic (2-way) operator, the optimizer must generate plans that achieve N-way joins as a sequence of 2-way joins. When joining more than a few tables, the number of such possible sequences is the dominant factor in the number of alternative plans: $N!$ different sequences are possible for joining N tables. Even when dynamic programming is used, as System R [SEL 79] and most current products do, theoreticians have used the exponential worst case complexity to argue that heuristic search methods should be used. However, these methods cannot guarantee optimality of their solution, as can dynamic programming.

For this reason, many existing optimizers use heuristics within dynamic programming to limit the join sequences evaluated. One heuristic employed by System R [SEL79] and R* [LOH85] constructs only joins in which a single table is joined at each step with the results of previous joins, in a pipelined fashion. This generates plans such as (((T1 ⋈ T2) ⋈ T3) ⋈ T4), and avoids so-called ***composite inners*** (often referred to as "bushy trees" [GRA87]), in which the ***inner table***

[1] Current address: Tokyo Research Laboratory IBM Japan, Ltd. 5-19, Sanbancho, Chiyoda-ku Tokyo 102, JAPAN

(the second operand of the join) is the result of a join that must be materialized in memory or — if it is too big — on disk. The heuristic saves this materialization, but may exclude better plans for certain queries. As an example, suppose a query with four large tables T1, T2, T3, and T4 has two predicates T1.C1 = T2.C2 and T3.C3 = T4.C4 that are extremely selective (restrictive), and one T2.C6 = T4.C8 that is not. Then the plan ((T1 ⋈ T2) ⋈ (T3 ⋈ T4)) with a composite inner (T3 ⋈ T4) is likely to be better than any plan avoiding the composite inner, such as ((T1 ⋈ T2) ⋈ T4) ⋈ T3, because the intermediate results of the first plan would all be significantly smaller than any in the second plan.

Another major heuristic employed by both System R [SEL79] and INGRES [WON76] always defers Cartesian products as late in the join sequence as possible, assuming that they result in large intermediate tables because there is no join predicate that restricts the result (and hence every row in one table is joined with every row in the other table). Again, this heuristic may exclude the optimal plan for certain queries that can benefit from Cartesian products. For instance, if the tables to be joined are small, and especially where they contain only one tuple each, a Cartesian product is quite inexpensive. And its result may even have columns forming a composite key for another, much larger table to be accessed later, thus making the Cartesian product more advantageous. As an example, consider a query with three tables T1, T2, and T3, and two predicates T1.C1 = T2.C2 and T2.C3 = T3.C4. The plan ((T1 ⋈ T3) ⋈ T2) is potentially the best plan if T1 and T3 are very small (or made so by additional single-table predicates) and there is a multi-column index for T2 on columns C2 and C3, even though it requires the Cartesian product of T1 and T3. This will be illustrated concretely in Section 3.1.

To make matters worse, many existing query optimizers severely restrict the class of predicates that qualify as these critical join predicates to be simple equijoins of the form *"Column1 = Column2"*, excluding any predicates that reference more than two tables or involve arithmetic on the column values. Consequently, what may appear to the user as a perfectly good join predicate is not treated as such by the system [LOH86], and a suboptimal join sequence results. For example, for a query with three tables T1, T2, and T3, and two predicates T1.C1 = T2.C2 and T2.C4 + 3 = T3.C5, the optimizer treats the latter predicate in a plan such as (T1 ⋈ T2) ⋈ T3) as a restriction of the Cartesian product between (T1 ⋈ T2) and T3, rather than as a join predicate. Thus, in conjunction with the heuristic that defers Cartesian products, any plan that might join T2 and T3 first is not considered, even if it dominates all plans joining T1 and T2 first.

Another limitation in handling predicates is that many systems do not derive ***implied predicates***, which are not specified but are implied by the predicates given by the user. For instance, two predicates T1.C1 = T2.C2 and T2.C2 = T3.C3 imply a new predicate, T1.C1 = T3.C3. Without this implied predicate, System R would avoid joining T1 with T3 until either had been joined with T2. A few systems have relaxed these limitations somewhat. For example, commercial INGRES can generate the join between T1 and T3 by using an *attribute equivalence class* [KOO80], although it does not apply the equivalence classes to more general forms of predicates — such as T2.C2 = T3.C3 + 1 — to derive an implied predicate T1.C1 = T3.C3 + 1, because join predicates are still restricted to be equality predicates between two columns.

Solution: An adaptable search space

Although evaluating more plans may find a more efficient plan for executing some queries, it also increases the cost of optimizing a given query. Hence it is important to be able to adjust the number of alternative plans considered by the optimizer for specific applications and queries. For example, traditional batch-oriented applications can tolerate longer optimization times than can interactive applications, for which the optimization time is as important as the execution time. This ***adaptability*** (or customizability) of a query optimizer is even more critical for non-traditional applications, such as decision support systems and expert systems, and for queries generated automatically by a user front end. These applications tend to pose very complex queries referring to more tables than traditional applications [KRI84, SWA88]; without some of the above heuristics, it might be infeasible to evaluate all the possible join sequences[2].

Even within a single type of application, certain queries might benefit more from considering more alternatives than others. A query whose best plan is expected to

2 Whether the optimizer can accurately estimate the number of rows resulting from such complex queries is an orthogonal issue, and will in any case not prevent applications from posing these queries!

run for 15 minutes is more likely to profit substantially from an additional minute of optimization than is a query estimated to execute in under a minute! The space of alternative join sequences must therefore be adjustable for each query.

Previous work

Several authors have considered the benefits of increasing or decreasing the space of possible join sequences through which the optimizer searches. In the original (University) INGRES, intermediate results were always materialized so that the optimizer could assess the next join to perform based upon the size of the composite [WON76]. The commercial version of INGRES allows plans to contain composite inners, but does not require them to do so [KOO80]. Rosenthal's optimizer [ROS82] also included plans with composite inners. However, so far as is known, none of these optimizers permitted adjusting the search space used by the optimizer. The rule-based optimizer generator built by Graefe for EXODUS was easily adapted to consider composite inners by changing just a rule or two [GRA87]. However, the optimizer so generated was then fixed and could not be adjusted for different queries without generating a new optimizer. Graefe showed that the number of plans considered by the EXODUS optimizer[3] went up dramatically between 4 and 7 joins per query, but then (surprisingly!) leveled off from 8 to 14. Graefe offered no explanation or analysis for this apparent anomaly, nor any figures for queries exceeding 14 joins. Using the EXODUS cost equations, Graefe also showed that plans containing composite inners had significantly improved (estimated) cost only when the number of joins per query exceeded 10. Since his composites were growing monotonically, he reasoned that "bushy trees" balanced the workload better as the number of joins increased, allowing more joins on moderately-sized intermediate results. Swami [SWA89] measured the actual execution of a large number of queries, allowing and disallowing both Cartesian products and composite inners. His results showed that a small percentage of the plans in the increased search space are optimal.

Starburst's Advances

The Starburst join enumerator improves upon these earlier efforts by parameterizing the space of alternative join sequences that the optimizer will consider for any given query, to include or exclude consideration of composite inners and Cartesian products. Thus, the level of optimization effort can be tailored individually to the query, either manually by the user or automatically by the system. In addition, the modular design of the Starburst join enumerator, described below, facilitates the addition or replacement of feasibility criteria for particular applications, exploits as join predicates any implied predicates and predicates involving more than two tables or arithmetic, and places no algorithmic limit on the number of tables in a query. And unlike many other optimizers that are limited to simple SELECT-PROJECT-JOIN queries, the Starburst optimizer correctly processes queries involving nested subqueries and correlation predicates [LOH84].

These enhancements allowed us to measure the impact of varying the optimizer's search space, both upon the optimization complexity and upon the resulting execution time. Our experimental results demonstrate that the complexity of optimizing a query is largely dependent upon the ***shape of the query graph***, where the shape of a query indicates how tables are connected with predicates, as well as the number of tables it references. Somewhat surprisingly, the number of feasible joins does not increase exponentially with the number of tables for certain commonly posed queries. For example, for ***linear queries***, in which tables are connected by binary predicates in a straight line, the number of feasible joins is a polynomial of the number of the tables, even when composite inners are allowed. With the same feasibility criteria, the number increases to an exponential for ***star queries***, in which a table at the center is connected by binary predicates to each of the other surrounding tables, the same as for completely connected query graphs.

This paper is organized as follows. First, a brief overview of the design of Starburst plan optimization is given, including a discussion of how its join enumerator can be adapted and extended. The remainder of the paper presents analysis and experimental results on the number of feasible joins for queries containing up to 110 tables, as the search space is varied using Starburst's parameterized join enumerator. We show how the shape affects the number of feasible joins, and how this adaptability can help balance the number of feasible joins and the practicality of optimizing queries that must join that many tables.

---

[3] Actually, the average size at the end of optimization of the data structure, called MESH, which contained plans.

Starburst's Adaptable Join Enumeration

Overview of Starburst's Plan Optimization

This paper deals only with the plan optimization phase of query processing in Starburst, an extensible relational database management system being prototyped at the IBM Almaden Research Center [HAA90]. An overview of Starburst query processing can be found in [HAA88]. For a given query, ***plan optimization*** evaluates alternative ***query execution plans (QEPs)***, and outputs for execution the QEP with the least estimated execution cost. The input to plan optimization is a parsed query that is stored in an internal database called the ***Query Graph Model (QGM)***. QGM is an internal representation of the semantics of an SQL query, including its various entities, such as tables, quantifiers, and predicates, as well as the relationships among them. A ***quantifier*** corresponds to a *join variable* in SQL and a *tuple variable* in QUEL, and represents a tuple drawn from a table. The ***range_over*** relationship connects the quantifier to the table from which the tuple is drawn. Since a single table may be referenced in several different contexts within a single query, several quantifiers may range over the same table.

The Starburst optimizer generates plans that construct progressively larger sets of quantifiers (***quantifier sets***) by joining pairs of two smaller quantifier sets, starting initially from plans for single quantifiers. This is the same inductive (bottom-up) algorithm used in System R and R*, and enables us at each step to incorporate into the join plan the optimal plan for each of its component quantifier sets (i.e. to use dynamic programming to prune dominated plan fragments), so that we never have to reconstruct a smaller plan even if its use in a bigger plan is different.

The plan optimizer in Starburst differs from System R and R*, however, in that it consists of two separate and highly extensible sub-components: the ***join enumerator***, which enumerates ***join orders*** specifying the order in which the query's tables can be joined, and a rule-based ***plan generator*** [LOH88, LEE88], which generates alternative QEPs and evaluates their estimated cost. This separation is similar to that described in [ROS82]. Within the overall bottom-up algorithm described above, the plan generator is first invoked individually for each quantifier, in order to generate alternative QEPs that specify execution details such as the access method, the columns to retrieve, and any predicates that can be applied. Next, pairs of quantifier sets to be joined are enumerated by the join enumerator. *For each such pair of quantifier sets*, the join enumerator invokes the plan generator to generate and evaluate alternative QEPs for that join, passing to it the quantifier sets, any join predicates linking those operands, and any limitations on the order of join (discussed below). As with the plans for single tables, each join QEP returned by the plan generator specifies execution details for one alternative, such as an order and method of joining the two quantifier sets, the columns that can be projected out, the plan chosen for producing each of the two inputs to the join, and any intermediate operations (such as sorts) that must be inserted to make the chosen join method work with the chosen input plans [LOH88].

The remainder of this paper will concentrate on the top part of plan optimization, the join enumerator, whose implementation is summarized next.

Join Enumeration

Starburst's adaptable join enumerator utilizes a ***generate and filter strategy***, in which a superset of feasible joins are generated by a ***join generator***, and infeasible ones are removed by a sequence of independent ***filters*** enforcing feasibility criteria, which can be of two kinds: (1) mandatory, universally valid criteria, and (2) optional, parameterized heuristics for reducing the search space. The optional criteria, such as the heuristics deferring Cartesian products and avoiding composite inners, are parameterized so that either the user or the system can control the number of feasible joins generated during optimization. We chose this form of extensibility for the join enumerator over the rule-based approach of the plan generator because it was sufficiently flexible to enumerate a wide range of potential join sequences — including those containing composite inners, Cartesian products, implied predicates, and predicates on more than two tables — while avoiding the performance overhead of rule interpretation. Furthermore, which join sequences are feasible is determined by fundamental relational principles, not the methods by which joins are implemented, and so are unlikely to be affected by other extensions to the system (e.g., new access paths or join algorithms).

The join generator generates progressively larger quantifier sets, starting from sets containing only one quantifier, and stores information about quantifier sets resulting from feasible joins thus far in the ***quantifier set table***. This process is analogous to mathematical induction. For the N quantifiers in a query, the quantifier set table is initialized with N sets containing only one

quantifier. Now, at any point in the algorithm, suppose we have constructed the optimal plan for all feasible quantifier sets containing up to $k-1$ quantifiers ($k \geq 2$) in the quantifier set table. We can obtain all feasible joins producing k quantifiers by considering every pair of quantifier sets having i quantifiers and $k-i$ quantifiers ($1 \leq i \leq$ floor $(k/2)$).

A detailed description of the filters of the default feasibility criteria is given in [ONO88]. The mandatory filters are the same as for most systems:

- **Disjointness:** two quantifier sets to be joined must be disjoint[4].
- **Dependency:** no quantifiers in the *outer* operand of a join should ***"depend_on"*** any quantifiers in the *inner* operand. The "depend_on" relationship summarizes any restrictions on the relative order in which quantifiers must be instantiated, due to semantic constructs such as correlation predicates, particularly in nested subqueries [LOH84]. Depend_on relationships are derived from QGM, and their transitive closure is calculated before optimization begins. If neither of two given quantifier sets can be the outer, or if a seemingly-feasible join produces a composite that will never be joinable with any other quantifier set[5], the two sets cannot form a feasible join (see [ONO88] for additional examples and details).

Parameterized Control of the Number of Feasible Joins

The number of feasible joins can be controlled by parameters that control the join generation and the filters. There are currently two such parameters:

- **Cartesian products:** This parameter enables or disables a filter requiring at least one eligible join predicate that references quantifiers in the two quantifier sets to be joined. A ***join predicate*** is defined in Starburst in more general terms than those in System R [SEL79] to be any multi-table predicate, and includes ***implied predicates***, i.e. those predicates derivable from predicates given in the query. Starburst derives and exploits implied predicates in a slightly more general approach than does INGRES [KOO80]; for details see [ONO88]. Including implied predicates may produce better execution plans by allowing more feasible joins.
- **Composite inners:** Enumeration of composite inners can be controlled by a parameter *maximum_size_of_smaller_set* to the join generator. This parameter specifies the maximum size of the smaller quantifier set of each join: if it is set to 1, enumeration of composite inners is disabled[6]; by setting it to some integer j ($1 < j \leq$ floor$(N/2)$), composite inners whose size are less than or equal to j are enumerated. The larger this parameter, the more "bushy" any plan can be. For example, when *maximum_size_of_smaller_set* = 2, composite inners can be constructed from individual quantifiers only; when *maximum_size_of_smaller_set* = floor(N/2), any composite inner may itself be constructed of composite inners.

Replacing Parts of Join Enumeration

The modular construction of the Starburst join enumerator facilitates the wholesale replacement of many of its pieces:

- **Replacing the join generator:** The join generator described previously is a general-purpose generator, and can be replaced with a special-purpose generator for particular feasible join criteria and queries. In fact, we have implemented an alternative generation method, called the ***Graph Traversal (GT)*** generator, that generates only pairs of quantifier sets satisfying the heuristic requiring a join between the quantifier sets. It does this by maintaining, for each quantifier set, a pointer to each quantifier set with which it shares at least one common join predicate. Although deriving these pointers requires more work initially,

---

4 Note that this condition does not exclude joins of a table with itself, because two accesses to the table are represented as two different quantifiers. Recursive joins are specifically detected and excluded from this criterion.

5 This conflict occurs when there are some quantifiers upon which the inner depends and which themselves depend upon some quantifiers in the outer.

6 Actually, the join enumerator only enumerates pairs of quantifier sets, one of which always has a single quantifier if maximum_size_of_smaller_set is set to 1. In conjunction with the *depend_on* relationship, the plan generator then decides which quantifier set is the inner operand of the join in any particular alternative plan.

it reduces the time to enumerate feasible joins for certain queries, as shown in Section 3.4.

- **Replacing the join feasibility filters:** Because each join feasibility criterion is an independent Boolean function, it is easy to change the function to another Boolean function that decides whether it is advantageous to join two quantifier sets. For instance, the filter requiring join predicates could be replaced by a new function that allows joins between any two sets whose estimated number of tuples are sufficiently small, even though there might be no join predicates between the sets.

- **Replacing the entire enumeration algorithm:** The entire enumeration method can be replaced as a whole without affecting the rest of the plan optimizer. This extension might be beneficial or necessary for handling very large queries with heuristic methods, such as Iterative Improvement [SWA88] or the Greedy Algorithm of the OS/2 Extended Edition Database Management System [LOH89].

Experimental Results

Using the Starburst adaptable join enumerator described above to vary the search space, we measured the complexity of enumerating the feasible joins for several sample queries. Since we wanted to concentrate on the primary factor in optimization cost, namely the complexity of enumerating join sequences, we tried to minimize the impact of what join and access methods were currently implemented in our testbed by first just counting the number of feasible joins. The total optimization time is approximately the product of the number of feasible joins and the average time to generate plans for a single feasible join. The latter is the time for the rule-based plan generator to construct plans for each feasible join, which is an orthogonal issue unrelated to the question at hand, dependent upon how many alternative access and join methods are available to choose from, is not significantly affected by the characteristics of queries, and therefore is essentially constant for all joins. We also measured the extent to which the Graph Traversal (GT) join generator reduced the time to enumerate joins.

The number of joins evaluated for a query depends on two classes of factors: 1) characteristics of the query, such as the number of quantifiers, the number of predicates, and the shape of the query, indicating how the quantifiers are connected by the predicates, and 2) join feasibility criteria, such as whether composite inners are allowed or not. In the following, unless otherwise noted, the ***default join feasibility criterion*** is used, i.e., composite inners are allowed and the existence of at least one join predicate is required for each join.

Verifying the Gain of Larger Search Spaces

Although often postulated as "well known" in the literature, examples from real applications that benefit from allowing Cartesian products or composite inners are rarely documented. Hence we first wanted to verify empirically that increasing the search space of the join enumerator to include Cartesian products and composite inners produced significantly better plans for some queries.

Cartesian products

Database designers often encode wide columns in a large table (e.g., encode "California" as "CA" or as an integer), and put the encodings for each column in a separate table. This was done in the following example database and query, containing a large (10,000-row) table drawn from the actual online telephone directory of IBM employees in the San Jose area. The query uses a DEPTS table of only 51 rows to encode the department name to a department number, and a NODES table of only 24 rows to encode the node name to a node identifier[7]:

```
Database Schema:
 SJDIR: LAST, FIRST, MIDDLE, PHONE,
        DEPT, OFFICE, NODEID, USERID
 DEPTS: DEPTID, DEPTLOC, DEPTNAME
 NODES: NODEID, NODENAME

Index on SJDIR: NODEID, DEPT

Query:
 SELECT last, first, dept, c.nodeid
 FROM   nodes a, depts b, sjdir c
 WHERE  a.nodename = 'STL VM #14'
 AND    b.deptname = 'DB2 Optimiz.'
 AND    a.nodeid = c.nodeid
 AND    b.deptid = c.dept;
```

This (star-shaped) query was run on Starburst, allowing

7 Of course, these cardinalities do not reflect the actual number of these entities in the San Jose area!

and disallowing Cartesian products, using 16 buffer pages that were flushed before each execution. Only one user was active. The best plan without Cartesian products did approximately 8 times the work of the best plan with Cartesian products, which formed the Cartesian product of DEPTS and NODES, then accessed SJDIR using the index:

BEST PLAN	DISK I/Os	PAGE REFNS.
WITH Cart. prods.	22	46
WITHOUT Cart. prods.	179	364

Effect of Query Characteristics

Effect of the number of quantifiers:

For a given set of join feasibility criteria, the number of feasible joins is determined primarily by two factors: the number of tables to join and the shape of the query graph. Figure 1 shows the shapes of three representative queries with 13 quantifiers and 12 binary predicates. In the figure, a dot represents a quantifier, and a rectangle represents a binary predicate. In the ***linear*** query, all 13 quantifiers are connected consecutively with 12 predicates. In the ***star*** query, the quantifier at the center is connected to 12 surrounding quantifiers.

The following two theorems give the relationship between query shape and number of quantifiers for the extreme cases of linear and star-shaped queries, when constructed using dynamic programming. Note that these calculations compute the number of feasible joins (in which the outer and inner quantifier sets are not distinguished), i.e., the number of times in Starburst that the plan generator is called; to obtain the number of join sequences, multiply by 2.

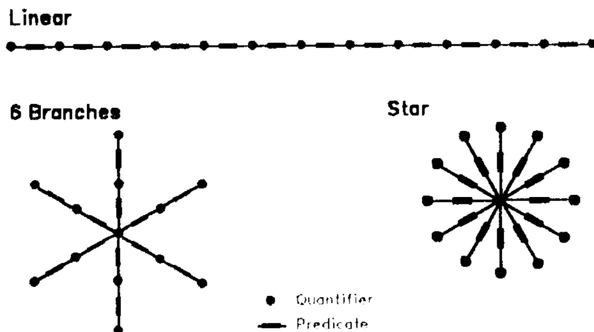

Figure 1: Examples of Query Shapes

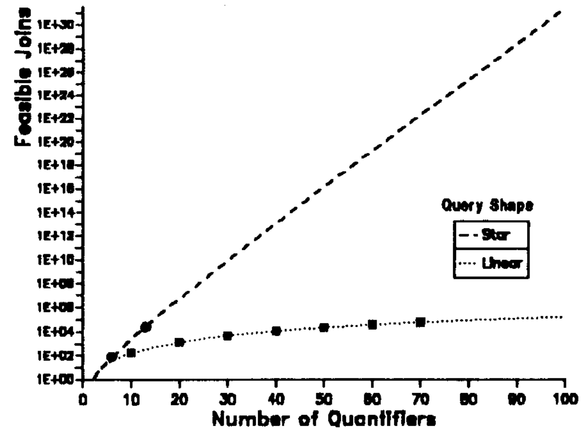

Figure 2: Effect of Number of Quantifiers on Number of Feasible Joins, for Linear and Star Queries

THEOREM 1 (Complexity of Linear Queries with Composite Inners): ***Using dynamic programming to optimize a <u>linear</u> query with N quantifiers, and allowing composite inners (bushy trees), requires evaluating $(N^3 - N)/6$ feasible joins.***

PROOF: For a linear query with N quantifiers, each of the K steps in the dynamic programming algorithm $(2 \le K \le N)$ inductively constructs joins of K consecutive quantifiers from the best plan fragments containing $1, 2, \ldots, K-1$ quantifiers constructed by the previous iterations. There are $(N - K + 1)$ such quantifier sets having K consecutive quantifiers, and each of these can be constructed from $(K - 1)$ different feasible joins, because there are exactly $(K - 1)$ places to break the K-quantifier subgraph into 2 smaller pieces to be joined. Therefore, the total number of feasible joins is:

$$\sum_{K=2}^{N}(K-1)(N-K+1) = (N^3 - N)/6$$

COROLLARY 1A (Complexity of Linear Queries without Composite Inners): ***Using dynamic programming to optimize a <u>linear</u> query with N quantifiers, and disallowing composite inners (bushy trees), requires evaluating $(N-1)^2$ feasible joins.***

PROOF: The proof follows from the above, noticing that there are only 2 (instead of K-1) ways to break the K-quantifier subgraph into fragments of size 1 and K-1 for $3 \le K \le N$, and only 1 way when K = 2.

THEOREM 2 (Complexity of Star Queries): *Using dynamic programming to optimize a* <u>*star*</u> *query with N quantifiers requires evaluating* $(N-1)2^{N-2}$ *feasible joins.*

PROOF: For a star query with N quantifiers, a feasible quantifier set with K quantifiers is obtained by choosing $(K-1)$ quantifiers from the $(N-1)$ quantifiers surrounding the center (or "***hub***") quantifier, which must be included in any join (hence, allowing composite inners or not makes no difference in the complexity!). Hence, there are $\binom{N-1}{K-1}$ feasible quantifier sets, each of which can be constructed from $(K-1)$ different joins. Therefore, the total number of feasible joins is:

$$\sum_{K=2}^{N}(K-1)\binom{N-1}{K-1} = (N-1)2^{N-2}$$

We are interested in star queries because, *for a given number of quantifiers and predicates*, star queries have worst case complexity. One can see this intuitively by moving any edge between any node j and the hub of the star so that it connects to any non-hub node k instead, producing a star having one fewer edge incident upon its hub, plus edge (k,j). Now there is only one way (via k) to join j to the rest of the alterred graph, yielding fewer choices than when j was linked directly to the hub.

The analytic formulas of the above theorems were then verified by using Starburst to optimize queries that varied each factor individually. Since the number of feasible joins for linear and star queries are the minimum and maximum numbers, respectively, of feasible joins for a query with a given number of quantifiers, we show in Figure 2 the number of joins, on a logarithmic scale, as a function of the number of quantifiers, for both linear and star queries. The lines denote the number of joins predicted by Theorems 1 and 2, and the dots and squares indicate experimental confirmation of this analysis using Starburst's adaptable join enumerator. Exactly as predicted, the number of joins for star queries increases exponentially, while the number for linear queries increases polynomially, as the number of quantifiers increases. From this, we can conclude that our join enumeration algorithm, as well as any other enumeration algorithm based upon dynamic programming, can remain practical for linear queries much larger than currently allowed by most relational DBMSs, but becomes impractical for large star queries.

Effect of query shape: To quantify the relationship between complexity and query shape, we held the number of quantifiers and predicates constant at 13 and 12, respectively, and just varied the shape of the query. Figure 3 shows how fast the number of feasible joins increased as the shape of the queries varied from linear to star, even though all queries have 13 quantifiers and 12 binary predicates. The abscissa measures the number of branches in the query graph, varying from 1 branch (a linear query) to 12 of them (a star query).

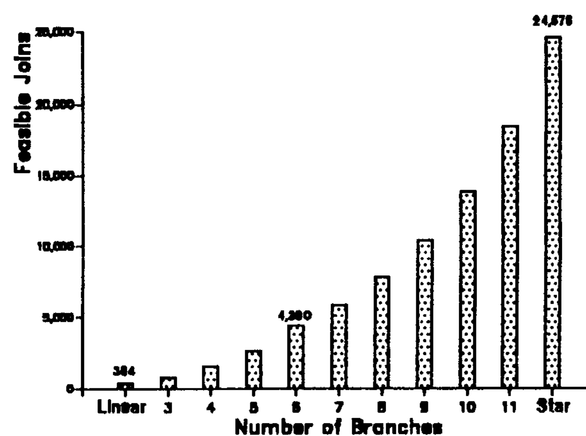

Figure 3: Effect of Query Shape on Number of Joins

Effect of Feasiblity Criteria

For a given query, varying the join enumerator's parameters can drastically alter the number of joins considered, as demonstrated by the following experiments.

Effect of composite inners for linear queries: Figure 4 shows how the number of feasible joins for linear queries increases as the maximum size of the smaller quantifier set. In the figure, the total number of quantifiers in a query is parameterized by N, and the abscissa represents Starburst's composite inner parameter, *maximum_size_of_smaller_set*, the maximum number of quantifiers allowed in the smaller quantifier set. This parameter may vary from one, where no composite inners are allowed, to the floor of $N / 2$, where composite inners of any size are allowed. Again, the lines in the figure are obtained by our combinatorial analysis, and the squares indicate experimental confirmation of this analysis by executing the query on Starburst. As predicted, the observed number of joins without composite inners is significantly smaller than that with composite inners. Note that the measured number of feasible joins for 110 quantifiers *without* composite inners was less than the number of joins for 50 quantifiers *with* composite inners. Therefore, we can optimize a larger

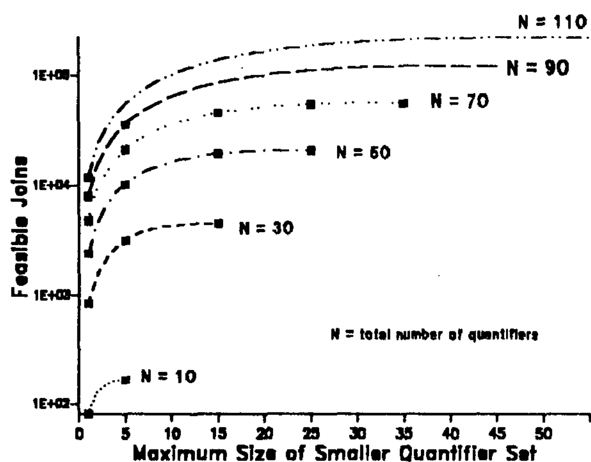

Figure 4: Effect of Composite Inner on Feasible Joins, for Linear Query

linear query in the same amount of time as that for smaller linear queries, if we disable composite inners for the larger query. This is precisely what System R and R* did for *all* queries, but, in Starburst, our parameterized adaptability gives the user control over how many joins are enumerated. With Starburst's flexible enumeration method, we can set the maximum number of quantifiers in the smaller set to any integer between one and $N/2$ to control the number of feasible joins.

Effect of composite inners for star queries: One of the operands of a feasible join for a star query must be a single quantifier, because two sets of more than two quantifiers will share the quantifier at the center, thus violating the disjointness feasibility criterion for those two quantifier sets. Therefore, the number of feasible joins for a star query is the same regardless of whether enumeration of composite inners are enabled or disabled.

Effect of Cartesian products: When Cartesian products between two quantifier sets are allowed, regardless of the existence of a join predicate between the sets, the number of feasible joins for a query are the same, independent of the shape of the query. The number of feasible joins for this case is interesting because it gives the maximum number of feasible joins for the query. For a query with N quantifiers, the number is $(3^N - 2^{N+1} + 1)/2$ *with* composite inners, and $N2^{N-1} - N(N+1)/2$ *without* composite inners. It is also interesting to note that the number of feasible joins for a query with N quantifiers *allowing* Cartesian products but *not* composite inners is the same as the number of feasible joins for a star query with $N+1$ quantifiers, less $N(N+1)/2$. This can be explained intuitively as follows: for a star query with $N+1$ quantifiers, any surrounding quantifier is connected to any other quantifier through the quantifier at the center. Therefore, the N surrounding quantifiers become fully connected after quantifier sets containing two quantifiers are formed. The number of ways to make the initial join adds the minor term $N(N+1)/2$, which is $\binom{N}{1} + \binom{N}{2}$. This result further confirms our use of star queries as a worst case when join predicates are required.

CPU Time for Enumeration

The time to enumerate joins can be reduced substantially by reducing the number of potential joins coming from the join generator. Figure 5 and Figure 6 show that exploiting the algorithm's adaptability by changing the join generation part of the algorithm reduced the time to enumerate feasible joins. In both cases, the number of potential joins coming from the join generator was reduced without affecting the set of feasible joins. In the first case, where linear queries were optimized, the alternative (GT) generator was used. In the second case, where star queries were optimized, composite inners were disabled. The CPU time was measured on an IBM RT/PC model 6150 with 4MB of memory, running the AIX version 2.1.2 Operating System. Note once again that the time shown is only the time to enumerate feasible joins, and intentionally excludes the time to construct and evaluate actual plans for the feasible joins.

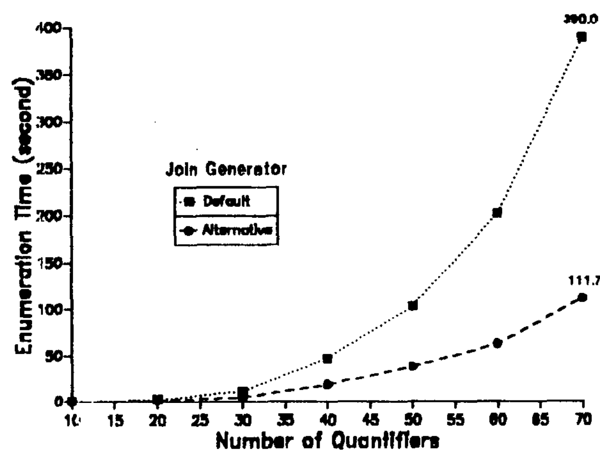

Figure 5: Reduction of Enumeration Time by the Graph Traversal Generation Method, for Linear Queries

Reduction of enumeration time for linear queries: Figure 5 shows that the Graph Traversal (GT) join generator reduced the time to enumerate the same set of feasible joins for linear queries by a factor of 2.7 on average, and by a factor of 3.5 (from 390.0 seconds to 111.7 seconds) for a query with 70 quantifiers, in particular. In addition, the GT join generator handled larger queries as efficiently as smaller queries, because its average time to enumerate each feasible join was constant at about 1.8 milliseconds for all queries, whereas the average time for the default method increased from 1.8 milliseconds (for $N = 10$) to 6.8 milliseconds (for $N = 70$). The reason for these improvements is that the GT join generator generates fewer pairs of potential quantifier sets than the default method, so fewer must be filtered. For instance, for the query with 70 quantifiers, the GT generator generates only 139,265 pairs, of which almost half (57,155) were feasible joins, compared with 2,542,470 pairs by the default method.

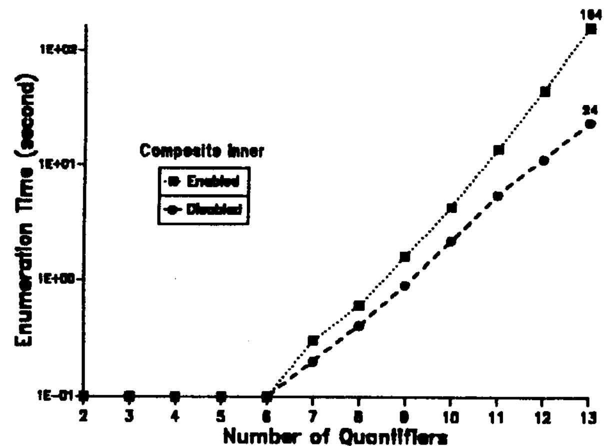

Figure 6: Reduction of Enumeration Time (on a Logarithmic Scale) by Disabling Composite inners for Star Queries

However, the GT join generator worked poorly for star queries because the feasibility criterion requiring a join predicate — which the GT generator exploits — is vacuously satisfied in star queries for all pairs containing two or more quantifiers. In fact, for star queries, the number of potential joins coming from the GT join generator actually *increases*, because every quantifier set is put into entries for every predicate in the modified quantifier set table. For instance, for a star query with 13 quantifiers, the GT generator increased the number of potential joins from 3,566,678 to 14,717,640, increasing the total enumeration time from 164.0 seconds to 680.0 seconds. For this reason, the GT join generator is invoked only for linear queries in Starburst. This illustrates how adaptability of our join enumeration algorithm for a particular query reduces the time to enumerate feasible joins.

Reduction of enumeration time for star queries: The time to enumerate feasible joins for star queries was reduced by disabling the enumeration of composite inners, as shown in Figure 6. Note that a logarithmic scale is used for the ordinate to show the exponential increase of the time as the number of quantifiers in a star query increases, and that all times less than 0.1 seconds are shown to be 0.1 seconds in the figure because the resolution of the CPU clock was 0.1 seconds. Recall that disabling the enumeration of composite inners for star queries does not reduce the set of feasible joins, since one of the operands must be a single quantifier anyway. For a star query with 13 quantifiers, the enumeration time was reduced by a factor of 6.8 (from 164.0 seconds to 24.0 seconds), and this reduction factor increased as the number of quantifiers increased.

Conclusions

Enumeration of the join sequences for a query is the dominant factor in both the optimization time for the query and the quality of resulting execution plans. To generate better execution plans, the join enumeration algorithm in this paper enlarges the set of feasible plans. The resulting plans can contain composite inners, and Cartesian products can occur at any place in join sequences, not necessarily at the end. Furthermore, by generalizing join predicates to include non-equality predicates, implied predicates, and predicates on more than two tables, the Starburst optimizer can generate more efficient plans that exploit these predicates as join predicates, rather than having to resort to Cartesian products.

Although enlarging the set of feasible joins generally makes the optimization time larger, we can balance the number of feasible plans with the optimization time by varying the number of feasible joins using our parameterized feasible join criteria. This kind of adaptability is important so that some queries may be optimized extensively into an extremely efficient plan, and complex queries can be optimized at all.

Our experimental results on the number of feasible joins show that we can find the optimal plan using dynamic programming for a complex query referring

to as many as 100 tables if the shape of the query is linear or almost linear. For linear queries, the number of feasible joins can also be controlled by a parameterized feasible join criterion on composite inners that controls the maximum size of smaller quantifier sets for a join. It can be set to any integer from one (no composite inners) to the maximum, which is half of the total number of quantifiers (full composite inners). Although intuitively the linear shape seems to be the most common shape of queries, it would be an interesting future study to examine shapes of queries in real applications, classify the shapes, and obtain empirical formulas for typical shapes, because the shape of a query largely determines the number of feasible joins for the query and, thus, the practicality of optimizing it.

We also measured the join enumeration time, and found that adaptability in changing part of the enumeration method allows us to reduce the enumeration time for queries of various shapes and sizes; there seems to be no single join enumeration method that works best for all queries. Using our Graph Traversal join generator for linear queries, and disabling composite inners for star queries, we reduced the time to enumerate joins without reducing the number of resulting feasible joins.

Currently, these adaptability mechanisms are controlled by a user at a terminal. For instance, the user must disable composite inners and select the alternative join generation method based on the shape and size of the query at hand. It should be straightforward to automate these decisions in the future, so that the enumerator itself decides the *best* options based on the submitted query. We also hope to derive a repertoire of heuristic join enumeration methods for very large queries with hundreds of tables.

Acknowledgements

The authors wish to thank Laura Haas, John McPherson, and Pat Selinger for reviewing an earlier draft of this paper, significantly improving its readability. The implementation of the join enumerator was facilitated considerably by the well-designed QGM data structures developed by Hamid Pirahesh, who generously explained and adapted them when necessary for the join enumeration algorithm. We are grateful to Laura Haas for her technical as well as managerial guidance and support. Hanh Nguyen and George Lapis provided invaluable programming support for some of the empirical results. The work reported in this paper was done while the primary author was on assignment from IBM's Tokyo Research Laboratory. He wishes to express his thanks to the members of the Starburst project for providing a stimulating and supportive working environment during that assignment at the Almaden Research Center.

Bibliography

[GRA87] G. Graefe., Rule-Based Query Optimization in Extensible Database Systems, *Computer Sciences Tech. Report #724 (PhD thesis)* (Univ. of Wisconsin, Madison, WI, Nov. 1987).

[HAA88] L.M. Haas, W.F. Cody, S. Finkelstein, J.C. Freytag, G. Lapis, B.G. Lindsay, G.M. Lohman, K. Ono, and H. Pirahesh., An Extensible Processor for an Extended Relational Query Language, *IBM Research Report RJ6182, IBM Almaden Research Center, San Jose, CA* (Apr. 1988).

[HAA90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, Starburst Mid-Flight: As the Dust Clears, *IEEE Trans. on Knowledge and Data Engineering* (March 1990). Also available as IBM Research Report RJ7278, San Jose, CA, Jan. 1990.

[KOO80] R.P. Kooi, The Optimization of Queries on Relational Databases, *Report No. CES-80-8, Case Western Reserve University, Cleveland, Ohio* (Sept. 1980).

[KRI84] R. Krishnamurthy, H. Boral, and C. Zaniolo, Optimization of Nonrecursive Queries, *Procs. of VLDB* (1984).

[LEE88] M.K. Lee, J.C. Freytag, and G.M. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Procs. of 14th VLDB* (Long Beach, August 1988) pp. 218—229.

[LOH84] G.M. Lohman, D. Daniels, L.M. Haas, R. Kistler, and P.G. Selinger, Optimization of Nested Queries in a Distributed

Relational Database, *Procs. of 10th VLDB* (August 1984) pp. 403—415.

[LOH85] G.M. Lohman, C. Mohan, L.M. Haas, B.G. Lindsay, P.G. Selinger, P.F. Wilms, and D. Daniels, Query Processing in R*, *Query Processing in Database Systems* (Kim, Batory, & Reiner (eds.), 1985) pp. 31—47. Springer-Verlag, Heidelberg. Also available as IBM Research Report RJ4272, San Jose, CA, April 1984.

[LOH86] G.M. Lohman, Do Semantically Equivalent SQL Queries Perform Differently?, *Procs. of IEEE Data Engineering* (February 1986) pp. 225—226.

[LOH88] G.M. Lohman, Grammar-like Functional Rules for Representing Query Optimization Alternatives, *Procs. of ACM-SIGMOD* (June 1988).

[LOH89] G.M. Lohman, Is Query Optimization a 'Solved' Problem? (extended abstract), *Workshop on Database Query Optimization (CSE Tech. Report 89-005)* (Oregon Graduate Center, Portland, OR, June 1989).

[ONO88] K. Ono and G.M. Lohman, Extensible Enumeration of Feasible Joins for Relational Query Optimization, *IBM Research Report RJ6625* (San Jose, CA, Dec. 1989).

[ROS82] A. Rosenthal and D. Reiner, An Architecture For Query Optimization, *Procs. of ACM-SIGMOD* (1982).

[SEL79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, Access Path Selection in a Relational Database Management System, *Procs. of ACM-SIGMOD* (1979).

[SWA88] A. Swami and A. Gupta, Optimization of Large Join Queries, *Procs. of ACM-SIGMOD* (June 1988).

[SWA89] A. Swami, Optimization of Large Join Queries: Distributions of Query Plan Costs, *Tech. Report HPL-SAL-89-24* (Hewlett-Packard Labs, Palo Alto, CA, June 1989).

[WON76] E. Wong and K. Youssefi,, Decomposition -- a Strategy for Query Processing, *ACM Transactions on Database Systems* **1**,3 (September 1976) pp. 223—241.