Predicate Migration: Optimizing Queries with Expensive Predicates

Joseph M. Hellerstein*
Computer Science Division, EECS Department
University of California, Berkeley, CA 94720
joey@postgres.berkeley.edu

December 3, 1992

Abstract

The traditional focus of relational query optimization schemes has been on the choice of join methods and join orders. Restrictions have typically been handled in query optimizers by "predicate pushdown" rules, which apply restrictions in some random order before as many joins as possible. These rules work under the assumption that restriction is essentially a zero-time operation. However, today's extensible and object-oriented database systems allow users to define time-consuming functions, which may be used in a query's restriction and join predicates. Furthermore, SQL has long supported subquery predicates, which may be arbitrarily time-consuming to check. Thus restrictions should not be considered zero-time operations, and the model of query optimization must be enhanced.

In this paper we develop a theory for moving expensive predicates in a query plan so that the total cost of the plan — including the costs of both joins and restrictions — is minimal. We present an algorithm to implement the theory, as well as results of our implementation in POSTGRES. Our experience with the newly enhanced POSTGRES query optimizer demonstrates that correctly optimizing queries with expensive predicates often produces plans that are orders of magnitude faster than plans generated by a traditional query optimizer. The additional complexity of considering expensive predicates during optimization is found to be manageably small.

1 Introduction

Traditional relational database (RDBMS) literature on query optimization stresses the significance of choosing an efficient order of joins in a query plan. The placement of the other standard relational operators (restriction and projection) in the plan has typically been handled by "pushdown" rules (see *e.g.*, [Ull89]), which state that restrictions and projections should be pushed down the query plan tree as far as possible. These rules place no importance on the ordering of projections and restrictions once they have been pushed below joins.

The rationale behind these pushdown rules is that the relational restriction and projection operators take essentially no time to carry out, and reduce subsequent join costs. In today's systems, however, restriction can no longer be considered to be a zero-time operation. Extensible database systems such as POSTGRES [SR86] and Starburst [HCL+90], as well as various Object-Oriented DBMSs (*e.g.*, [MS87], [WLH90], [D+90], [ONT92], etc.) allow users to implement predicate functions in a general-purpose programming language such as C or C++. These functions can be arbitrarily complex, potentially requiring access to large amounts of data, and extremely complex processing. Thus it is unwise to choose a random order of application for restrictions on such predicates, and it may not even be optimal to push them down a query plan tree. Therefore the traditional model of query optimization

*This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

does not produce optimal plans for today's queries, and as we shall see, the plans that traditional optimizers generate can be many orders of magnitude slower than a truly optimal plan.

To illustrate the significance of ordering restriction predicates, consider the following example:

Example 1.

```
/* Find all maps from week 17 showing more than 1% snow cover.
   Channel 4 contains images from the frequency range that interests us.  */
retrieve (maps.name)
   where  maps.week = 17 and maps.channel = 4
     and  coverage(maps.picture) > 1
```

In this example, the function `coverage` is a complex image analysis function that may take many thousands of instructions to compute. It should be quite clear that the query will run faster if the restrictions `maps.week = 17` and `maps.channel = 4` are applied before the restriction `coverage(maps.picture) > 1`, since doing so minimizes the number of calls to `coverage`.

While restriction ordering such as this is important, correctly ordering restrictions within a table-access is not sufficient to solve the general problem of where to place predicates in a query execution plan. Consider the following example:

Example 2.

```
/* Find all channel 4 maps from weeks starting in June that show more than 1% snow
   cover.  Info about each week is kept in the weeks table, requiring a join. */
retrieve (maps.name)
   where  maps.week = weeks.number
     and  weeks.month = "June" and maps.channel = 4
     and  coverage(maps.picture) > 1
```

Traditionally, a DBMS would execute this query by applying all the single-table restrictions in the *where* clause before performing the join of `maps` and `weeks`, since early restriction can lower the complexity of join processing. However in this example the cost of evaluating the expensive restriction predicate may outweigh the benefit gained by doing restriction before join. In other words, this may be a case where "predicate pushdown" is precisely the wrong technique. What is needed here is "predicate pullup", namely postponing the restriction `coverage(maps.picture) > 1` until after computing the join of `maps` and `weeks`.

In general it is not clear how joins and restrictions should be interleaved in an optimal execution plan, nor is it clear whether the migration of restrictions should have an effect on the join orders and methods used in the plan. This paper describes and proves the correctness of the *Predicate Migration Algorithm*, which produces an optimal query plan for queries with expensive predicates. Predicate Migration modestly increases query optimization time: the additional cost factor is polynomial in the number of operators in a query plan. This compares favorably to the exponential join enumeration schemes used by most query optimizers, and is easily circumvented when optimizing queries without expensive predicates — if no expensive predicates are found while parsing the query, the techniques of this paper need not be invoked. For queries with expensive predicates, the gains in execution speed should offset the extra optimization time. We have implemented Predicate Migration in POSTGRES, and have found that with modest overhead in optimization time, the execution time of many practical queries can be reduced by orders of magnitude. This will be illustrated below.

1.1 Application to Existing Systems: SQL and Subqueries

It is important to note that expensive predicate functions do not exist only in next-generation research prototypes. Current relational languages, such as the industry standard, SQL [ISO91], have long supported expensive predicate functions in the guise of *subquery predicates*. A subquery predicate is one of the form "expression operator query".

Evaluating such a predicate requires executing an arbitrary query and scanning its result for matches — an operation that is arbitrarily expensive, depending on the complexity and size of the subquery. While some subquery predicates can be converted into joins (thereby becoming subject to traditional join-based optimization strategies) even sophisticated SQL rewrite systems, such as that of Starburst [PHH92], cannot convert all subqueries to joins. When one is forced to compute a subquery in order to evaluate a predicate, then the predicate should be treated as an expensive function. Thus the work presented in this paper is applicable to the majority of today's production RDBMSs, which support SQL.

1.2 Related Work

Stonebraker first raised the issue of expensive predicate optimization in the context of the POSTGRES multi-level store [Sto91]. The questions posed by Stonebraker are directly addressed in this paper, although we vary slightly in the definition of cost metrics for expensive functions.

One of the main applications of the system described in [Sto91] is Project Sequoia 2000 [SD92], a University of California project that will manage terabytes of Geographic Information System (GIS) data, to support global change researchers. It is expected that these researchers will be writing queries with expensive functions to analyze this data. A benchmark of such queries is presented in [SFG92].

Ibaraki and Kameda [IK84], Krishnamurthy, Boral and Zaniolo [KBZ86], and Swami and Iyer [SI92] have developed and refined a query optimization scheme that is built on the notion of *rank* that we will use below. However, their scheme uses *rank* to reorder joins rather than restrictions. Their techniques do not consider the possibility of expensive restriction predicates, and only reorder nodes of a single path in a left-deep query plan tree, while the technique presented below optimizes all paths in an arbitrary tree. Furthermore, their schemes are a proposal for a completely new method for query optimization, while ours is an extension that can be applied to the plans of any query optimizer. It is possible to fuse the technique we develop in this paper with those of [IK84, KBZ86, SI92], but we do not focus on that issue here since their schemes are not widely in use.

The notion of expensive restrictions was considered in the context of the $\mathcal{LDL}$ logic programming system [CGK89]. Their solution was to model a restriction on relation R as a join between R and a virtual relation of infinite cardinality containing the entire logical predicate of the restriction. By modeling restrictions as joins, they were able to use a join-based query optimizer to order all predicates appropriately. Unfortunately, most traditional DBMS query optimizers have complexity that is exponential in the number of joins. Thus modelling restrictions as joins can make query optimization prohibitively expensive for a large set of queries, including queries on a single relation. The scheme presented here does not cause traditional optimizers to exhibit this exponential growth in optimization time.

Caching the return values of function calls will prove to be vital to the techniques presented in this paper. Jhingran [Jhi88] has explored a number of the issues involved in caching procedures for query optimization. Our model is slightly different, since our caching scheme is value-based, simply storing the results of a function on a set of argument values. Jhingran's focus is on caching complex object attributes, and is therefore instance-based.

1.3 Structure of the Paper

The following section develops a model for measuring the cost and selectivity of a predicate, and describes the advantages of caching for expensive functions. Section 3 presents the Predicate Migration Algorithm, a scheme for optimally locating predicates in a given join plan. Section 4 describes methods to efficiently implement the Predicate Migration Algorithm in the context of traditional query optimizer. Section 4 also presents the results of our implementation experience in POSTGRES. Section 5 summarizes and provides directions for future research.

2 Background: Expenses and Caching

To develop our optimizations, we must enhance the traditional model for analyzing query plan cost. This will involve some modifications of the usual metrics for the expense of relational operators, and will also require the introduction of *function caching* techniques. This preliminary discussion of our model will prove critical to the analysis below.

flag name	*description*
percall_cpu	execution time per invocation, regardless of the size of the arguments
perbyte_cpu	execution time per byte of arguments
byte_pct	percentage of argument bytes that the function will need to access

Table 1: Function Expense Parameters in POSTGRES

A relational query in a language such as SQL or Postquel [RS87] may have a *where* clause, which contains an arbitrary Boolean expression over constants and the range variables of the query. We break such clauses into a maximal set of conjuncts, or "Boolean factors" [SAC+79], and refer to each Boolean factor as a distinct "predicate" to be satisfied by each result tuple of the query. When we use the term "predicate" below, we refer to a Boolean factor of the query's *where* clause. A *join predicate* is one that refers to multiple tables, while a *restriction predicate* refers only to a single table.

Traditional query optimizers compute *selectivities* for both joins and restrictions. That is, for any predicate p (join or restriction) they estimate the value

$$\textit{selectivity}(p) = \frac{\textit{card}(\textit{output}(p))}{\textit{card}(\textit{input}(p))}$$

and make the assumption that selectivities of different predicates are independent. Typically these estimations are based on default values and system statistics [SAC+79], although recent work suggests that accurate and inexpensive sampling techniques can be used [LNSS93, HOT88].

2.1 Cost of User-Defined Functions in POSTGRES

In an extensible system such as POSTGRES, arbitrary user-defined functions may be introduced into both restriction and join predicates. These functions may be written in a general programming language such as C, or in the database query language, *e.g.* SQL or Postquel. In this section we discuss programming language functions; we handle query language functions below.

Given that user-defined functions may be written in a general purpose language such as C, there is little hope for the database to correctly estimate the cost and selectivity of predicates containing these functions, at least not initially.[1] In this section we extend the POSTGRES function definition syntax to capture a function's expense. Selectivity modeling for user-defined operators in POSTGRES has been described in [Mos90].

To introduce a function to POSTGRES, a user first writes the function in C and compiles it, and then issues Postquel's `define function` command. To capture expense information, the `define function` command accepts a number of special flags, which are summarized in Table 1.

The cost of a predicate in POSTGRES is computed by adding up the costs for each expensive function in the expression. Given a POSTGRES predicate $p(a_1, \ldots, a_n)$, the expense per tuple is recursively defined as:

$$e_p = \begin{cases} \sum_{i=1}^{n} e_{a_i} + \textit{percall\_cpu}(p) \\ \qquad + \textit{perbyte\_cpu}(p) * (\textit{byte\_pct}(p)/100) * \sum_{i=1}^{n} \textit{bytes}(a_i) + \textit{access\_cost} \\ \qquad \text{if } p \text{ is a function} \\ 0 \qquad \text{if } p \text{ is a constant or tuple variable} \end{cases}$$

where e_{a_i} is the recursively computed expense of argument a_i, *bytes* is the expected (return) size of the argument in bytes, and *access_cost* is the cost of retrieving any data necessary to compute the function. This data may be stored anywhere in the various levels of the POSTGRES multi-level store, but unlike [Sto91] we do not require the user to

[1] After repeated applications of a function, one could collect performance statistics and use curve-fitting techniques to make estimates about the function's behavior. Such techniques are beyond the scope of this paper.

define constants specific to the different levels of the multi-level store. Instead, this can be computed by POSTGRES itself via system statistics, thus providing more accurate information about the distribution and caching of data across the storage levels.

2.2 Cost of SQL Subqueries and Other Query Language Functions

SQL allows a variety of subquery predicates of the form "expression operator query". Such predicates require computation of an arbitrary SQL query for evaluation. Simple *uncorrelated* subqueries have no references to query blocks at higher nesting levels, while *correlated* subqueries refer to tuple variables in higher nesting levels.

In principle, the cost to check an uncorrelated subquery restriction is the cost e_m of materializing the subquery once, and the cost e_s of scanning the subquery once per tuple. However, we will need these cost estimates only to help us reorder operators in a query plan. Since the cost of initially materializing an uncorrelated subquery must be paid regardless of the subquery's location in the plan, we ignore the overhead of the materialization cost, and consider an uncorrelated subquery's cost per tuple to be e_s.

Correlated subqueries must be materialized for each value that is checked against the subquery predicate, and hence the per-tuple expense for correlated subqueries is e_m. We ignore e_s here since scanning can be done during each materialization, and does not represent a separate cost. Postquel functions in POSTGRES have costs that are equivalent to those of correlated subqueries in SQL: an arbitrary access plan is executed once per tuple of the relation being restricted by the Postquel function.

The cost estimates presented here for query language functions form a simple model and raise some issues in setting costs for subqueries. The cost of a subquery predicate may be lowered by transforming it to another subquery predicate [LDH+87], and by "early stop" techniques, which stop materializing or scanning a subquery as soon as the predicate can be resolved [Day87]. Incorporating such schemes is beyond the scope of this paper, but including them into the framework of the later sections merely requires more careful estimates of the subquery costs.

2.3 Join Expenses

In our subsequent analysis, we will be treating joins and restrictions uniformly in order to optimally balance their costs and benefits. In order to do this, we will need to measure the expense of a join per tuple of the join's input, *i.e.* per tuple of the cartesian product of the relations being joined. This can be done for any join method whose costs are linear in the cardinalities of the input relations, including the most common algorithms: nested-loop join, hash join, and merge join.[2]

Note that a query may contain many join predicates over the same set of relations. In an execution plan for a query, some of these predicates are used in processing a join, and we call these *primary join predicates*. If a join has expensive primary join predicates, then the cost per tuple of a join should reflect the expensive function costs. That is, we add the expensive functions' costs, as described in Section 2.1, to the join costs per tuple.

Join predicates that are not applicable while processing the join are merely used to restrict its output, and we refer to these as *secondary join predicates*. Secondary join predicates are essentially no different from restriction predicates, and we treat them as such. These predicates may then be reordered and even pulled up above higher join nodes, just like restriction predicates. Note, however, that a secondary join predicate must remain above its corresponding primary join. Otherwise the secondary join predicate would be impossible to evaluate.

2.4 Function Caching

The existence of expensive predicates not only motivates richer optimization schemes, it also suggests the need for DBMSs to cache the results of expensive predicate functions. Some functions, such as subquery functions, may be cached only for the duration of a query; other functions, such as functions that refer to a transaction identifier, may be cached for the duration of a transaction; most straightforward data analysis or manipulation functions can be cached

[2] Sort-merge join is not linear in the cardinalities of the input relations. However, most systems, including POSTGRES, do not use sort-merge join, since in situations where merge join requires sorting of an input, either hash join or nested-loop join is almost always preferable to sort-merge.

indefinitely. Occasionally a user will define a restriction function that cannot be cached at all, such as a function that checks the time of day, or that generates a random number. A query containing such a function is non-deterministic, since the function is not guaranteed to return the same value every time it is applied to the same arguments. Since the use of such functions results in ill-defined queries, and since they are relatively unusual, we do not consider them here.

Instead, we assume that all functions can be cached, and that the system caches the results of evaluating expensive functions at least for the duration of a query. This lowers the cost of a function, since with some probability the function can be evaluated simply by checking the cache. In this section we develop an estimate for this probability, which should be factored into the per-tuple predicate costs described above.

In addition to lowering function cost, caching will also allow us to pull expensive restrictions above joins without modifying the total cost of the restriction nodes in the plan. In general, a join may produce as many tuples as the product of the cardinalities of the inner and outer relations. However, it will produce no new *values* for attributes of the tuples; it will only recombine these attributes. If we move a restriction in a query plan from below a join to above it, we may dramatically increase the number of times we evaluate that restriction. However by caching expensive functions we will not increase the number of expensive function calls, only the number of cache lookups, which are quick to evaluate. This results from the fact that after pulling up the restriction, the same set of function calls on distinct arguments will be made. In many cases the primary join predicates will in fact *decrease* the number of distinct values passed into the function. Thus we see that with function caching, pulling restrictions above joins does not increase the number of function calls, and often will decrease that number.

The probability of a function cache miss depends on the state of the function's cache before the query begins execution, and also on the expected number of duplicate arguments passed to the function. In order to estimate the number of cache misses in a given query, we must be able to describe the distribution of values in the cache as well as the distribution of the arguments to the function. To do this, every time we invoke an n-ary function f, we cache the arguments to f and its return value in a database relation *f_cache*, which has tuples of the form

$$(arg_1, \ldots, arg_n, \mathit{return\text{-}value}).$$

We index this relation on the composite key $(arg_1, \ldots, arg_n)$, so that before computing f on a set of arguments we can quickly check whether its return value has already been computed. Since *f_cache* is a relation like any other, the system can provide distribution information for each of its attributes. As noted above, this information can be estimated with a variety of methods, including the use of system statistics or sampling. In the absence of distribution information, some default assumptions must be made as to the distribution. The issue of how to derive an accurate distribution is orthogonal to the work here, and we merely assume that it is done to a reasonable degree of accuracy.

Given a model of the distribution of a function's cache, and the distribution of the inputs to a function, one can trivially derive a ratio of cache misses to cache lookups for the function. This ratio serves as the probability of a cache miss for a given tuple.

To capture caching information in POSTGRES, we introduce one additional flag to the `define function` command. This *cache_life* flag lets the system know long it may cache the results of executing the function: setting *cache_life* = *infinite* implies that the function may be cached indefinitely, while *cache_life* = *xact* and *cache_life* = *query* denote that the cache must be emptied at end of transaction or query respectively.

2.4.1 Subquery Caching in SQL Systems

Current SQL systems do not support arbitrary caching of the results of evaluating subquery predicates. To benefit from the techniques described in this paper, an SQL system must be enhanced to do this caching, at least for the duration of a query. It is interesting to note that in the original paper on optimizing SQL queries in System R [SAC+79], there is a description of a limited form of caching for correlated subqueries. System R saved the materialization of a correlated subquery after each evaluation, and if the subsequent tuple had the same values for the columns referenced in the subquery, then the predicate could be evaluated by scanning the saved materialization of the subquery. Thus System R would cache a single materialization of a subquery, but did not cache the result of the subquery predicate. That is, for a subquery of the form "expression operator query", System R cached the result of "query", but not "expression operator query".

Table	Tuple Size	#Tuples
maps	1 040 424	932
weeks	24	19
emp	32	10 000
dept	44	20

Table 2: Benchmark Database

To apply the techniques presented here, we require caching of all values of the predicate for the duration of a query. It is sufficient for our purposes to cache only the values of the entire predicate, and not the values of each subquery. The two techniques are, however, orthogonal optimizations that can coexist. The System R approach (*i.e.* caching "query") saves materialization costs for adjacent tuples with duplicate values in the fields referenced by the subquery. Our approach (*i.e.* caching "expression operator query") saves materialization and scan costs for those tuples that have duplicate values *both* in the fields referenced by the subquery *and* in the fields on the left side of the subquery operator. In situations where either cache could be used to speed evaluation of a predicate, the latter is obviously a more efficient choice, since the former requires a scan of an arbitrarily sized set.

2.5 Environment for Performance Measurements

It is not uncommon for queries to take hours or even days to complete. The techniques of this paper can improve performance by several orders of magnitude — in many cases converting an over-night query to an interactive one. We will be demonstrating this fact during the course of the discussion by measuring the performance effect of our optimizations on various queries. In this section we present the environment used for these measurements.

We focus on a complex query workload (involving subqueries, expensive user-defined functions, etc), rather than a transaction workload, where queries are relatively simple. There is no accepted standard complex query workload, although several have been proposed ([SFG92, TOB89, O'N89], etc.) To measure the performance effect of Predicate Migration, we have constructed our own benchmark database, based on a combined GIS and business application. Each tuple in maps contains a reference to a POSTGRES large object [Ols92], which is a map picture taken by a satellite. These map pictures were taken weekly, and the maps table contains a foreign key to the weeks table, which stores information about the week in which each picture was taken. The familiar emp and dept tables store information about employees and their departments. Some physical characteristics of the database are shown in Table 2.

Our performance measurements were done in a development version of POSTGRES, similar to the publicly available version 4.0.1 (which itself contains a version of the Predicate Migration optimizations). POSTGRES was run on a DECStation 5000/200 workstation, equipped with 24Mb of main memory and two 300Mb DEC RZ55 disks, running the Ultrix 4.2a operating system. We measured the elapsed time (total time taken by system), and CPU time (the time for which CPU is busy) of optimizing and executing each example query, both with and without Predicate Migration. These numbers are presented in the examples which appear throughout the rest of the paper.

3 Optimal Plans for Queries With Expensive Predicates

At first glance, the task of correctly optimizing queries with expensive predicates appears exceedingly complex. Traditional query optimizers already search a plan space that is exponential in the number of relations being joined; multiplying this plan space by the number of permutations of the restriction predicates could make traditional plan enumeration techniques prohibitively expensive. In this section we prove the reassuring results that:

1. Given a particular query plan, its restriction predicates can be optimally interleaved based on a simple sorting algorithm.
2. As a result of the previous point, we need merely enhance the traditional join plan enumeration with techniques to interleave the predicates of each plan appropriately. This interleaving takes time that is polynomial in the

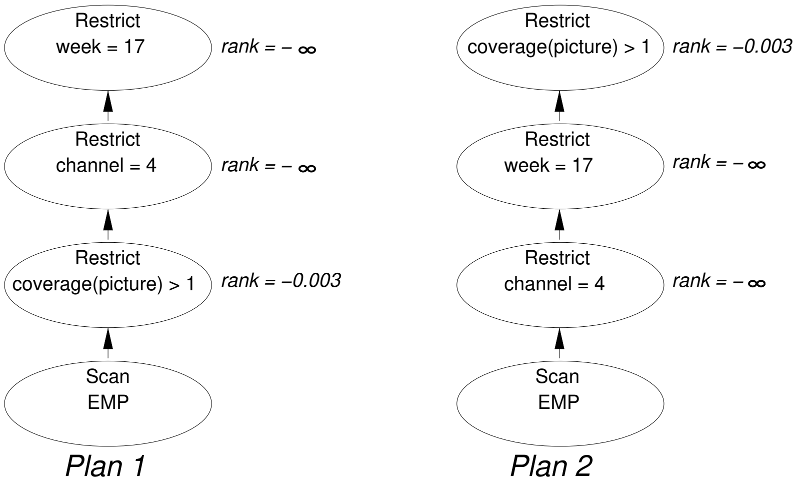

Figure 1: Two Execution Plans for Example 1

number of operators in a plan.

The proofs for the lemmas and theorems that follow are presented in Appendix A.

3.1 Optimal Predicate Ordering in Table Accesses

We begin our discussion by focusing on the simple case of queries over a single table. Such queries may have an arbitrary number of restriction predicates, each of which may be a complicated Boolean function over the table's range variables, possibly containing expensive subqueries or user-defined functions. Our task is to order these predicates in such a way as to minimize the expense of applying them to the tuples of the relation being scanned.

If the access path for the query is an index scan, then all the predicates that match the index and can be applied during the scan are applied first. This is because such predicates are essentially of zero cost: they are not actually evaluated, rather the indices are used to retrieve only those tuples which qualify.[3] We will represent each of the subsequent non-index predicates as $p_1, \ldots, p_n$, where the subscript of the predicate represents its place in the order in which the predicates are applied to each tuple of the base table. We represent the expense of a predicate p_i as e_{p_i}, and its selectivity as s_{p_i}. Assuming the independence of distinct predicates, the cost of applying all the non-index predicates to the output of a scan containing t tuples is

$$e_1 = e_{p_1}t + s_{p_1}e_{p_2}t + \cdots + s_{p_1}s_{p_2}\cdots s_{p_{n-1}}e_{p_n}t.$$

The following lemma demonstrates that this cost can be minimized by a simple sort on the predicates. It is analogous to the Least-Cost Fault Detection problem solved in [MS79].

Lemma 1 *The cost of applying expensive restriction predicates to a set of tuples is minimized by applying the predicates in ascending order of the metric*

$$\textit{rank} = \frac{\textit{selectivity} - 1}{\textit{cost-per-tuple}}$$

Thus we see that for single table queries, predicates can be optimally ordered by simply sorting them by their *rank*. Swapping the position of predicates with equal *rank* has no effect on the cost of the sequence.

To see the effects of reordering restrictions, we return to Example 1 from the introduction. We ran the query in POSTGRES without the *rank*-sort optimization, generating Plan 1 of Figure 1, and with the *rank*-sort optimization,

[3] It is possible to index tables on function values as well as on table attributes [MS86, LS88]. If a scan is done on such a "function" index, then predicates over the function may be applied during the scan, and are considered to have zero cost, regardless of the function's expense.

Execution Plan	Optimization Time		Execution time	
	CPU	Elapsed	CPU	Elapsed
Plan 1	0.12 sec	0.24 sec	20 min 34.36 sec	20 min 37.69 sec
Plan 2 (ordered by *rank*)	0.12 sec	0.24 sec	0 min 2.66 sec	0 min 3.26 sec

Table 3: Performance of Example 1

generating Plan 2 of Figure 1. As we expect from Lemma 1, the first plan has higher cost than the second plan, since the second is correctly ordered by *rank*. The optimization and execution times were measured for both runs, as illustrated in Table 3. We see that correctly ordering the restrictions can improve query execution time by orders of magnitude.

3.2 Predicate Migration: Moving Restrictions Among Joins

In the previous section, we established an optimal ordering for restrictions. In this section, we explore the issue of ordering restrictions among joins. Since we will eventually be applying our optimization to each plan produced by a typical join-enumerating query optimizer, our model here is that we are given a fixed join plan, and want to minimize the plan's cost under the constraint that we may not change the order of the joins. This section develops a poly-time algorithm to optimally place restrictions and secondary join predicates in a join plan. In Section 4 we show how to efficiently integrate this algorithm into a traditional optimizer.

3.2.1 Definitions

The thrust of this section is to handle join predicates in our ordering scheme in the same way that we handle restriction predicates: by having them participate in an ordering based on *rank*. However, since joins are binary operators, we must generalize our model for single-table queries to handle both restrictions and joins. We will refer to our generalized model as a *global* model, since it will encompass the costs of all inputs to a query, not just the cost of a single input to a single node.

Definition 1 *A* plan tree *is a tree whose leaves are* scan *nodes, and whose internal nodes are either* joins *or* restrictions*. Tuples are produced by scan nodes and flow upwards along the edges of the plan tree.*[4]

Some optimization schemes constrain plan trees to be within a particular class, such as the *left-deep* trees, which have scans as the right child of every join. Our methods will not require this limitation.

Definition 2 *A* stream *in a plan tree is a path from a leaf node to the root.*

Figure 2 below illustrates a plan tree, with one of its two plan streams outlined. Within the framework of a single stream, a join node is simply another predicate; although it has a different number of inputs than a restriction, it can be treated in an identical fashion. We do this by considering each predicate in the tree — restriction or join — as an operator on the *entire* input stream to the query. That is, we consider the input to the query to be the cartesian product of the relations referenced in the query, and we model each node as an operator on that cartesian product. By modeling each predicate in this global fashion, we can naturally compare restrictions and joins in different streams. However, to do this correctly, we must modify our notion of the per-tuple cost of a predicate:

Definition 3 *Given a query over relations $a_1, \ldots, a_n$, the* global cost *of a predicate p over relations $a_1, \ldots, a_k$ is defined as:*

$$\textit{global-cost}(p) = \frac{\textit{cost-per-tuple}(p)}{\textit{card}(a_{k+1}) \cdots \textit{card}(a_n)}$$

where cost-per-tuple *is the cost attribute of the predicate, as described in Section 2.*

[4] We do not consider common subexpressions or recursive queries in this paper, and hence disallow plans that are dags or general graphs.

That is, to define the cost of a predicate over the entire input to the query, we must divide out the cardinalities of those tables that do not affect the predicate. As an illustration, consider the case where p is a single-table restriction over relation a_1. If we push p down to directly follow the table-access of a_1, the cost of applying p to that table is $\textit{cost-per-tuple}(p)\textit{card}(a_1)$. But in our new global model, we consider the input to each node to be the cartesian product of $a_1, \ldots, a_n$. However, note that the cost of applying p in both the global and single-table models is the same, *i.e.*,

$$\textit{global-cost}(p)\textit{card}(a_1 \times \ldots \times a_n) = \textit{cost-per-tuple}(p)\textit{card}(a_1).$$

Recall that because of function caching, even if we pull p up to the top of the tree, its cost should not reflect the cardinalities of relations $a_2, \ldots, a_n$. Thus our global model does not change the cost analysis of a plan. It merely provides a framework in which we can treat all predicates uniformly.

The selectivity of a predicate is independent of the predicate's location in the plan tree. This follows from the fact that $\textit{card}(a_1 \times a_2) = \textit{card}(a_1)\textit{card}(a_2)$. Thus the global rank of a predicate is easily derived:

Definition 4 *The* global rank *of a predicate p is defined as*

$$\textit{rank} = \frac{\textit{selectivity}(p) - 1}{\textit{global-cost}(p)}$$

Note that the global cost of a predicate in a single-table query is the same as its user-defined *cost-per-tuple*, and hence the global rank of a node in a single-table query is the same as its *rank* as defined previously. Thus we see that the global model is a generalization of the one presented for single-table queries. In the subsequent discussion, when we refer to the *rank* of a predicate, we mean its global rank.

In later analysis it will prove useful to assume that all nodes have distinct *rank*s. To make this assumption, we must prove that swapping nodes of equal *rank* has no effect on the cost of a plan.

Lemma 2 *Swapping the positions of two equi-*rank *nodes has no effect on the cost of a plan tree.*

Knowing this, we could achieve a unique ordering on *rank* by assigning unique ID numbers to each node in the tree and ordering nodes on the pair (*rank*, ID). Rather than introduce the ID numbers, however, we will make the simplifying assumption that *rank*s are unique.

In moving restrictions around a plan tree, it is possible to push a restriction down to a location in which the restriction cannot be evaluated. This notion is captured in the following definition:

Definition 5 *A plan stream is* semantically incorrect *if some predicate in the stream refers to attributes that do not appear in the predicate's input.*

Streams can be rendered semantically incorrect by pushing a secondary join predicate below its corresponding primary join, or by pulling a restriction from one input stream above a join, and then pushing it down below the join into the other input stream. We will need to be careful later on to rule out these possibilities.

In our subsequent analysis, we will need to identify plan trees that are equivalent except for the location of their restrictions and secondary join predicates. We formalize this as follows:

Definition 6 *Two plan trees T and T' are* join-order equivalent *if they contain the same set of nodes, and there is a one-to-one mapping g from the streams of T to the streams of T' such that for any stream s of T, s and $g(s)$ contain the same join nodes in the same order.*

3.2.2 The Predicate Migration Algorithm: Optimizing a Plan Tree By Optimizing its Streams

Our approach in optimizing a plan tree will be to treat each of its streams individually, and sort the nodes in the streams based on their *rank*. Unfortunately, sorting a stream in a general plan tree is not as simple as sorting the restrictions in a table access, since the order of nodes in a stream is constrained in two ways. First, we are not allowed to reorder join nodes, since join-order enumeration is handled separately from Predicate Migration. Second, we must ensure that each stream remains semantically correct. In some situations, these constraints may preclude the option of simply ordering a stream by ascending *rank*, since a predicate p_1 may be constrained to precede a predicate p_2,

Execution Plan	Optimization Time		Execution time	
	CPU	Elapsed	CPU	Elapsed
Without Predicate Migration	0.29 sec	0.30 sec	20 min 29.79 sec	21 min 12.98 sec
With Predicate Migration	0.36 sec	0.57 sec	0 min 3.46 sec	0 min 6.75 sec

Table 4: Performance of Plans for Example 2

even though $rank(p_1) > rank(p_2)$. In such situations, we will need to find the optimal ordering of predicates in the stream subject to the precedence constraints.

Monma and Sidney [MS79] have shown that finding the optimal ordering under arbitrary precedence constraints can be done fairly simply. Their analysis is based on two key results:

1. A stream can be broken down into *modules*, where a module is defined as a set of nodes that have the same constraint relationship with all nodes outside the module. An optimal ordering for a module forms a subset of an optimal ordering for the entire stream.

2. For two predicates p_1, p_2 such that p_1 is constrained to precede p_2 and $rank(p_1) > rank(p_2)$, an optimal ordering will have p_1 directly preceding p_2, with no other predicates in between.

Monma and Sidney use these principles to develop the *Series-Parallel Algorithm Using Parallel Chains*, an $O(n \log n)$ algorithm for optimizing an arbitrarily constrained stream. The algorithm repeatedly isolates modules in a stream, optimizing each module individually, and using the resulting orders for modules to find a total order for the stream. We use their algorithm as a subroutine in our optimization algorithm:

Predicate Migration Algorithm: *To optimize a plan tree, we push all predicates down as far as possible,*[5] *and then repeatedly apply the Series-Parallel Algorithm Using Parallel Chains [MS79] to each stream in the tree, until no more progress can be made.*

Upon termination, the Predicate Migration Algorithm produces a tree in which each stream is *well-ordered* (*i.e.* optimally ordered subject to the precedence constraints). We proceed to prove that the Predicate Migration Algorithm is guaranteed to terminate in polynomial time, and we also prove that the resulting tree of well-ordered streams represents the optimal choice of predicate locations for the given plan tree.

Theorem 1 *Given any plan tree as input, the Predicate Migration Algorithm is guaranteed to terminate in polynomial time, producing a join-order equivalent tree in which each stream is semantically correct and well-ordered.*

Theorem 2 *For every plan tree T_1 there is a unique join-order equivalent plan tree T_2 with only well-ordered streams, and T_2 is a minimal cost tree that is join-order equivalent to T_1.*

Theorems 1 and 2 demonstrate that the Predicate Migration Algorithm produces our desired minimal-cost interleaving of predicates. As a simple illustration of the efficacy of Predicate Migration, we go back to Example 2 from the introduction. Figure 2 illustrates plans generated for this query by POSTGRES running both with and without Predicate Migration. The performance measurements for the two plans appear in Table 4.

4 Implementation Issues

4.1 Preserving Opportunities for Pruning

In the previous section we presented the Predicate Migration Algorithm, an algorithm for optimally placing restriction and secondary join predicates within a plan tree. If applied to every possible join plan for a query, the Predicate Migration Algorithm is guaranteed to generate a minimal-cost plan for the query.

[5] Most systems perform this operation while building plan trees, since "predicate pushdown" is traditionally considered a good heuristic.

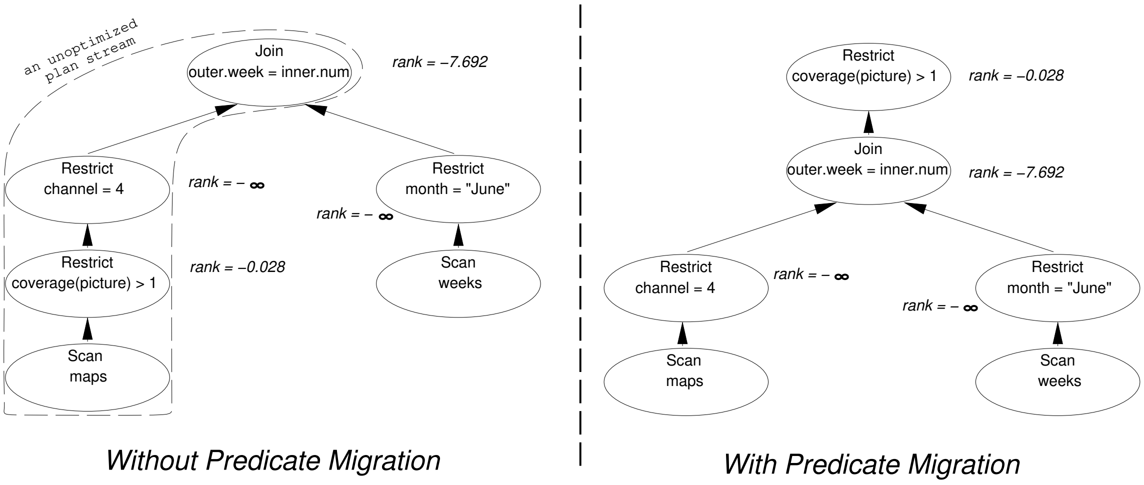

Figure 2: Plans For Example 2, With and Without Predicate Migration

A traditional query optimizer, however, does not enumerate all possible plans for a query; it does some pruning of the plan space while enumerating plans [SAC+79]. Although this pruning does not affect the basic exponential nature of join plan enumeration, it can significantly lower the amounts of space and time required to optimize queries with many joins. The pruning in a System R-style optimizer is done by a dynamic programming algorithm, which builds optimal plans in a bottom-up fashion. Unfortunately, this dynamic programming approach does not integrate well with Predicate Migration.

To illustrate the problem, we consider an example. We have a query which joins three relations, A, B, C, and performs an expensive restriction on C. A relational algebra expression for such a query, after the traditional predicate pushdown, is $A \bowtie B \bowtie \sigma_p(C)$. A traditional query optimizer would, at some step, enumerate all plans for $B \bowtie \sigma_p(C)$, and discard all but the optimal plan for this subgoal. Assume that because restriction predicate p has extremely high *rank*, it will always be pulled above all joins in any plan for this query. Then the join method which the traditional optimizer saved for $B \bowtie \sigma_p(C)$ is quite possibly sub-optimal, since in the final tree we would want the optimal subplan for $B \bowtie C$, not $B \bowtie \sigma_p(C)$. In general, the problem is that the subgoals of the dynamic programming algorithm may not actually form part of the optimal plan, since predicates may later migrate. Thus the pruning done during dynamic programming may actually discard part of an optimal plan for the entire query.

Although this looks troublesome, in many cases it is still possible to allow pruning to happen. First, note that pruning can take place for subtrees in which there are no expensive predicates; in such subtrees no restrictions will be pulled up, and therefore traditional pruning techniques work correctly. The following lemma helps to isolate more situations in which pruning may take place:

Lemma 3 *For a restriction or secondary join predicate R in a subtree, if the* rank *of R is greater than the* rank *of any join in any plan for the subtree, then in the optimal complete tree R will appear above the highest join in the subtree.*

This lemma can be used to allow some predicate pullup to happen during join enumeration. While builing a join node for some subgoal in the dynamic programming algorithm of System R, if the restrictions below the join are of higher *rank* than the join node, the lemma shows that they may be pulled above the join. If we pull up all expensive restrictions in all plans for the subgoal, then we may prune the plan space of the subgoal, just as we prune subtrees which contain no expensive restrictions initially. Extending this idea to its logical conclusion, we see that if we pull all expensive restrictions above the final, uppermost join in all plans, then Predicate Migration need not even be invoked — the optimal plan after Predicate Migration has already been generated by the modified dynamic programming algorithm.

As an additional optimization, note that the choice of an optimal join algorithm is sometimes independent of the sizes of the inputs. For example, if both of the inputs to a join are sorted on the join attributes, one may safely conclude that merge join will be a minimal-cost algorithm, regardless of the sizes of the inputs. Heuristics such as this may be used to allow pruning in cases where there is no way to verify which restrictions will be in the subtree after Predicate Migration. We believe that the observations of this section compensate for most of the pruning opportunities lost when enhancing a System R optimizer to support Predicate Migration. Particularly, note that no pruning opportunities are lost for traditional queries without expensive predicates.

4.2 Implementation in POSTGRES, and Further Measurement

The Predicate Migration Algorithm, as well as the pruning optimizations described above, were implemented in the POSTGRES next-generation DBMS, which has an optimizer based on that of System R. The addition of Predicate Migration to POSTGRES was fairly straightforward, requiring slightly more than one person-month of programming. The implementation consists of two files containing a total of about 2000 lines, or 600 statements, of C language code. It should thus be clear that enhancing an optimizer to support Predicate Migration is a fairly manageable task.

Given the ease of implementation, and the potential benefits for both standard SQL and extensible query languages, it is our belief that Predicate Migration is a worthwhile addition to any DBMS. To further motivate this, we present two more examples, which model SQL queries that would be natural to run in most commercial DBMSs. We simulate an SQL correlated subquery with a Postquel query language function, since POSTGRES does not support SQL. As noted above, SQL's correlated subqueries and Postquel's query language functions require the same processing to evaluate, namely the execution of a subplan per value. The only major distinction between our Postquel queries and an SQL system is that Postquel may return a different number of duplicate tuples than SQL, since Postquel assigns no semantics to the duplicates in a query's output. In our benchmark database the example queries return no tuples, and hence this issue does not affect the performance of our examples.

Example 3. This query finds all technical departments with either low budgets or an employee over the age of 65. In SQL, the query is:

```
SELECT name FROM dept d1
 WHERE d1.category = 'tech'
   AND (d1.budget < 1000
        OR EXISTS (SELECT 1 FROM emp
                   WHERE emp.dno = d1.dno AND emp.age > 65));
```

Since the existential subquery is nested within an OR, the subquery cannot be converted to a join [PHH92]. To simulate this query in Postquel, we define a function `seniors`, which takes one argument (`$1`) of type integer, and executes the Postquel query:

```
retrieve (x = "t")
   where  emp.dno = $1 and emp.age > 65
```

Given this function, the SQL query is simulated by the following Postquel query:

```
retrieve (dept.name)
   where  dept.category = "tech"
     and (dept.budget < 1000 or seniors(dept.dno))
```

Predicate Migration ensures that the expensive `OR` clause containing `seniors` is applied after the restriction `dept.category = "tech"`.[6] As shown in Table 5, Predicate Migration speeds up execution time by orders of magnitude, while affecting optimization time only marginally.

[6] As an additional optimization, POSTGRES orders the operands of `OR` by *rank*, and quits evaluating the `OR` expression as soon as any operand evaluates to true. This issue was left out of the discussion previously in order to simplify matters. It is a straightforward extension to the techniques presented here.

Execution Plan	Optimization Time		Execution time	
	CPU	Elapsed	CPU	Elapsed
Unoptimized Plan	0.34 sec	0.75 sec	2 min 25.61 sec	2 min 26.32 sec
Optimized Plan	0.34 sec	0.88 sec	0 min 0.06 sec	0 min 0.39 sec

Table 5: Performance of Plans for Example 3

Execution Plan	Optimization Time		Execution time	
	CPU	Elapsed	CPU	Elapsed
Unoptimized Plan	0.13 sec	0.42 sec	2 min 24.51 sec	2 min 25.69 sec
Optimized Plan	0.16 sec	0.52 sec	0 min 0.06 sec	0 min 0.39 sec

Table 6: Performance of Plans for Example 4

Example 4. Our final example uses a subquery and a join to find the managers of the departments found in the previous example. The SQL version of the query is:

```
SELECT dept.name, mgr.name FROM dept d1, emp mgr
 WHERE d1.category = 'tech'
   AND d1.dno = mgr.dno
   AND (d1.budget < 1000 OR EXISTS (SELECT 1 FROM emp e1
                                    WHERE e1.dno = d1.dno AND e1.age > 65));
```

Since this uses the same subquery as the previous example, the equivalent Postquel query can reuse the function `seniors`:

```
retrieve(dept.name, mgr.name) from mgr in emp
   where dept.category = "tech" and dept.dno = mgr.dno
     and (dept.budget < 1000 or seniors(dept.dno))
```

Predicate Migration in this query pulls the expensive `OR` clause above the join of `dept` and `emp`, resulting in the dramatic execution speedup shown in Table 6. Once again, the increase in optimization time is comfortably low.

These examples demonstrate that even for reasonable queries in standard SQL, the techniques presented in this paper can improve execution time by orders of magnitude.

5 Conclusions and Future Work

In this paper we highlight the fact that database query optimization has up until now ignored the costs associated with restriction. We present a framework for measuring these costs, and we argue the necessity of caching expensive functions in an RDBMS. We develop the Predicate Migration Algorithm, which is proven to transform query plans in a way that optimally interleaves restriction and join predicates. This was implemented in POSTGRES, and measurements show that Predicate Migration is a low-overhead optimization that can produce query plans that run orders of magnitude faster than those produced by systems without Predicate Migration. This work can be applied not only to advanced research DBMSs such as POSTGRES, but also to any DBMS that supports SQL. There are not many additions to current DBMSs that can produce dramatic performance gains with modest implementation cost. Predicate Migration is one such addition.

The optimization schemes in this paper are useful for run-time re-optimization. That is, if a query is optimized and the resulting plan is stored for a period of time, the statistics that shaped the choice of the optimal plan may have changed. Predicate Migration can be re-applied to the stored plan at runtime with little difficulty. This may not produce an optimal plan, since the join orders and methods may no longer be optimal. But it will optimize the stored plan itself, without incurring the exponential costs of completely re-optimizing the query. This could be particularly beneficial for queries with subqueries, since the costs of the subqueries are likely to change over time.

This paper represents only an initial effort at optimizing queries with expensive predicates, and there is substantial work remaining to be done in this area. The first and most important question is whether the assumptions of this paper can be relaxed without making query optimization time unreasonably slow. The two basic assumptions in the paper are (1) that function caching is implemented, and (2) that join costs are linear in the size of the inputs. Without either of these assumptions, there are no obvious directions to pursue a poly-time algorithm for Predicate Migration. If one does not have function caching, then our cost model no longer applies, since a restriction function will be called once for every *tuple* that flows through its predicate, rather than once per *value* of the attributes on which it is defined. If one does not assume linear join costs, then the algorithm of [MS79] no longer applies. It would be interesting to discover whether the problem of Predicate Migration can be solved in polynomial time in general, or whether the assumptions made here are in fact crucial to a poly-time solution.

The implementation of function caching in POSTGRES has not been completed. Once that is accomplished, we will be able to perform more complex experiments than the ones presented here, which were carefully tailored to produce no duplicate function calls after pullup. A more comprehensive performance study could develop a test suite of queries with expensive functions, and compare the performance of the Predicate Migration Algorithm against more naive predicate pullup heuristics.

It would be interesting to attempt to extend this work to handle queries with common subexpressions and recursion. The Magic Sets optimization technique for recursive and non-recursive queries [MFPR90] actually generates predicates in a query plan and pushes them down. It is not clear when this generation is cost-effective, and our model here may be useful for making that decision.

Finally, our cost analyses for user-defined functions could be dramatically improved by techniques to more correctly assess the expected running-time of a function on a given set of inputs. Particularly, the POSTGRES `define function` command includes an implicit assumption that users' functions will have complexity that is linear in the size of their data objects. This simplifying assumption was made to ease implementation, but it is certainly possible to add curve-fitting algorithms to better model a function's running time and complexity.

6 Acknowledgments

Mike Stonebraker brought the issue of expensive predicates to my attention, and regularly clarified what often seemed to be muddy and complicated problems. Wei Hong provided extensive and regular feedback on this work throughout my time at Berkeley. Jeff Naughton's encouragement, patience and support helped to bring about the completion of this work. Thanks to Mike Olson and Mark Sullivan for their comments on earlier drafts of this paper. This work could not have been completed without the assistance, suggestions, and friendly support of the entire POSTGRES research group.

References

[CGK89] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards an Open Architecture for LDL. In *Proc. 15th International Conference on Very Large Data Bases*, Amsterdam, August 1989.

[D+90] O. Deux et al. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In Proc. VLDB 87 [Pro87], pages 197–208.

[HCL+90] 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 Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990.

[HOT88] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeao K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 276–287, Austin, March 1988.

[IK84] Toshihide Ibaraki and Tiko Kameda. Optimal Nesting for Computing N-relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, October 1984.

[ISO91] ISO_ANSI. Database Language SQL ISO/IEC 9075:1992, 1991.

[Jhi88] Anant Jhingran. A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures. In Proc. VLDB 88 [Pro88].

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *Proc. 12th International Conference on Very Large Data Bases*, pages 128–137, Kyoto, August 1986.

[LDH+87] Guy M. Lohman, Dean Daniels, Laura M. Haas, Ruth Kistler, and Patricia G. Selinger. Optimization of Nested Queries in a Distributed Relational Database. In Proc. VLDB 87 [Pro87].

[LNSS93] Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. To appear in *Theoretical Computer Science*, 1993.

[LS88] C. Lynch and M. Stonebraker. Extended User-Defined Indexing with Application to Textual Databases. In Proc. VLDB 88 [Pro88].

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is Relevant. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 247–258, Atlantic City, May 1990.

[Mos90] Claire Mosher (ed.). The POSTGRES Reference Manual, Volume 2. Technical Report M90/53, Electronics Research Laboratory, University of California, Berkeley, July 1990.

[MS79] C. L. Monma and J.B. Sidney. Sequencing with Series-Parallel Precedence Constraints. *Mathematics of Operations Research*, 4:215–224, 1979.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In Klaus R. Dittrich and Umeshwar Dayal, editors, *Proc. Workshop on Object-Oriented Database Systems*, Asilomar, September 1986.

[MS87] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[Ols92] Michael A. Olson. Extending the POSTGRES Database System to Manage Tertiary Storage. Master's thesis, University of California, Berkeley, May 1992.

[O'N89] P. O'Neil. Revisiting DBMS Benchmarks. *Datamation*, pages 47–54, September 15, 1989.

[ONT92] ONTOS, Inc. *ONTOS Object SQL Guide*, February 1992. For the ONTOS DB database, Release 2.2.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule-Based Query Rewrite Optimization in Starburst. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 39–48, San Diego, June 1992.

[Pro87] *Proc. 13th International Conference on Very Large Data Bases*, Brighton, September 1987.

[Pro88] *Proc. 14th International Conference on Very Large Data Bases*, Los Angeles, August-September 1988.

[RS87] L.A. Rowe and M.R. Stonebraker. The POSTGRES Data Model. In Proc. VLDB 87 [Pro87], pages 83–96.

[SAC+79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, June 1979.

[SD92] Michael Stonebraker and Jeff Dozier. Sequoia 2000: Large Capacity Object Servers to Support Global Change Research. Technical Report Sequoia 2000 91/1, University of California, Berkeley, March 1992.

[SFG92] Michael Stonebraker, James Frew, and Kenn Gardels. The Sequoia 2000 Benchmark. Technical Report Sequoia 2000 92/12, University of California, Berkeley, June 1992.

[SI92] Arun Swami and Balakrishna R. Iyer. A Polynomial Time Algorithm for Optimizing Join Queries. Research Report RJ 8812, IBM Almaden Research Center, June 1992.

[SR86] M.R. Stonebraker and L.A. Rowe. The Design of POSTGRES. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1986.

[Sto91] Michael Stonebraker. Managing Persistent Objects in a Multi-Level Store. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 2–11, Denver, June 1991.

[TOB89] C. Turbyfill, C. Orji, and Dina Bitton. AS3AP - A Comparative Relational Database Benchmark. In *Proc. IEEE Compcon Spring '89*, February 1989.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

A Proofs

Lemma 1 *The cost of applying expensive restriction predicates to a set of tuples is minimized by applying the predicates in ascending order of the metric*

$$\mathit{rank} = \frac{\mathit{selectivity} - 1}{\mathit{cost\text{-}per\text{-}tuple}}$$

Proof. Although this result is essentially identical to the solution in [MS79] of the Least-Cost Fault Detection problem, we review it in our context for completeness.

Assume the contrary. Then in an minimal cost ordering $p_1, \ldots, p_n$, for some predicate p_k there is a predicate p_{k+1} where $\mathit{rank}(p_k) > \mathit{rank}(p_{k+1})$. Now, the cost of applying all the predicates to t tuples is

$$e_1 = e_{p_1}t + s_{p_1}e_{p_2}t + \cdots + s_{p_1}s_{p_2}\cdots s_{p_{k-1}}e_{p_k}t + s_{p_1}s_{p_2}\cdots s_{p_k}e_{p_{k+1}}t + \cdots + s_{p_1}s_{p_2}\cdots s_{p_{n-1}}e_{p_n}t.$$

But if we swap p_k and p_{k+1}, the cost becomes

$$e_2 = e_{p_1}t + s_{p_1}e_{p_2}t + \cdots + s_{p_1}s_{p_2}\cdots s_{p_{k-1}}e_{p_{k+1}}t + s_{p_1}s_{p_2}\cdots s_{p_{k+1}}e_{p_k}t + \cdots + s_{p_1}s_{p_2}\cdots s_{p_{n-1}}e_{p_n}t.$$

By subtracting e_1 from e_2 and factoring we get

$$e_2 - e_1 = ts_{p_1}s_{p_2}\cdots s_{p_{n-1}}(e_{p_{k+1}} + s_{p_{k+1}}e_{p_k} - e_{p_k} - s_{p_k}e_{p_{k+1}})$$

Now recall that $\mathit{rank}(p_k) > \mathit{rank}(p_{k+1})$, *i.e.*

$$(s_{p_k} - 1)/e_{p_k} > (s_{p_{k+1}} - 1)/e_{p_{k+1}}$$

After some simple algebra (taking into account the fact that expenses must be non-negative), this inequality reduces to

$$e_{p_{k+1}} + s_{p_{k+1}} e_{p_k} - e_{p_k} - s_{p_k} e_{p_{k+1}} < 0$$

i.e. this shows that the parenthesized term in the equation $e_2 - e_1$ is less than zero. By definition both t and the selectivities must be non-negative, and hence $e_2 < e_1$, demonstrating that the given ordering is not minimal cost, a contradiction. ■

Lemma 2 *Swapping the positions of two equi-rank nodes has no effect on the cost of a plan tree.*

Proof. Note that swapping two nodes in a plan tree only affects the costs of those two nodes. Consider two nodes p and q of equal rank, operating on input of cardinality t. If we order p before q, their joint cost is $e_1 = te_p + ts_p e_q$. Swapping them results in the cost $e_2 = te_q + ts_q e_p$. Since their ranks are equal, it is a matter of simple algebra to demonstrate that $e_1 = e_2$, and hence the cost of a plan tree is independent of the order of equi-*rank* nodes. ■

A.1 Pseudocode, Complexity and Optimality

In this section we present the Predicate Migration Algorithm in detail, and prove that it optimizes a plan tree in polynomial time.

Pseudocode for the Predicate Migration Algorithm is given in Figure 3, and we provide a brief explanation of the algorithm here. The function `predicate_migration` first pushes all predicates down as far as possible. The rest of `predicate_migration` is made up of a nested loop. The outer `do` loop ensures that the algorithm terminates only when no more progress can be made (*i.e.* when each stream is optimally ordered). The inner loop cycles through all the streams in the plan tree, applying Monma and Sidney's Series-Parallel Algorithm using Parallel Chains.

The Series-Parallel Algorithm repeatedly finds modules of the stream to optimize, and calls the Parallel Chains algorithm to optimize each module. Once all modules are optimized, the Parallel Chains algorithm can be used to optimize the entire stream.

The Parallel Chains algorithm expects as input a set of nodes that can be partitioned into two subsets: one of nodes that are constrained to form a chain, and another of nodes that are unconstrained relative to any node in the entire set. Note that by traversing the stream from top down, `series_parallel` always provides correct input to `parallel_chains`.[7] The Parallel Chains algorithm first finds groups of nodes in the chain that are constrained to be ordered sub-optimally (*i.e.* by descending *rank*). As shown in [MS79], there is always an optimal ordering in which such nodes are adjacent, and hence such nodes may be considered as an undivided group. The `find_groups` routine identifies the maximal-sized groups of poorly-ordered nodes. After all groups are formed, the module can be sorted by the rank of each group. The resulting total order of the module is preserved as a chain by introducing extra constraints. These extra constraints are discarded after the entire stream is completely ordered.

Thus when `predicate_migration` terminates, it leaves a tree in which each stream has been ordered by the Series-Parallel Algorithm using Parallel Chains. The interested reader is referred to [MS79] for justification of why the Series-Parallel Algorithm using Parallel Chains optimally orders a stream.

Lemma A.1 *Given a join node J in a module, adding a restriction or secondary join predicate R to the stream does not raise the rank of J's group.*

Proof. Assume J is initially in a group of k members, $\overline{p_1 \ldots p_{j-1} J p_{j+1} \ldots p_k}$ (from this point on we will represent grouped nodes as an overlined string). If R is not constrained with respect to any of the members of this group, then it will not affect the *rank* of the group — it will be placed either above or below the group, as apropriate. If R is constrained with some member p_i of the group, it is constrained to be *above* p_i (by semantic correctness); no predicate is ever constrained to be below any node. Now, the Predicate Migration Algorithm will eventually call `parallel_chains` on the module of all nodes constrained to follow p_i, and R will be pulled up within that module so that it is ordered by ascending *rank* with the other groups in the module. Thus if R is part of J's group in any

[7] Note also that for each module S' that `series_parallel` constructs from a stream S, each node of S' is constrained in exactly the same way with each node of $S - S'$. Thus `parallel_chains` is always passed a valid *Job Module*, in the terminology of [MS79].

```
/* Optimally locate restrictions in a query plan tree. */
predicate_migration(tree)
{
  preprocess:
    push all predicates down as far as possible;
  do {
    for (each stream)
      series_parallel(stream);
  } until no progress can be made;
}

/* Monma & Sidney's Series-Parallel Algorithm */
series_parallel(stream);
{
      for each join node J in stream, from top to bottom
        if (there is a node constrained to follow J
            and it is not constrained to precede anything else)
          parallel_chains(all nodes constrained to follow J);
      parallel_chains(stream);
      discard any constraints introduced by parallel_chains;
}

/* Monma and Sidney's Parallel Chains Algorithm */
parallel_chains(module)
{
  chain = {the nodes in module that form a chain of constraints};
  /* all nodes in the tree are in groups by themselves by default */
  find_groups(chain);
  if (groups in module are not sorted by their group's ranks) {
    progress = TRUE;
    sort nodes in module by their group's ranks;
  }
  /* the resulting order reflects the optimized module */
  introduce constraints to preserve the resulting order;
}

/* find adjacent groups that are constrained to be ordered by decreasing rank, and merge them. */
find_groups(chain)
{
  initialize each node in chain to be in a group by itself;
  while any two adjacent groups a, b are not ordered by ascending group rank {
    form a group ab of a and b;
    group_cost(ab) = group_cost(a) + group_selectivity(a) * group_cost(b);
    group_selectivity(ab) = group_selectivity(a) * group_selectivity(b);
  }
}
```

Figure 3: Predicate Migration Algorithm

module, it is only because the nodes below R form a group of higher *rank* than R. (The other possibility, *i.e.* that the nodes above R formed a group of lower *rank*, could not occur since `parallel_chains` would have pulled R above such a group.)

Given predicates p_1, p_2 such that $\textit{rank}(p_1) > \textit{rank}(p_2)$, it is easy to show that $\textit{rank}(p_1) > \textit{rank}(\overline{p_1 p_2})$. Therefore since R can only be constrained to be *above* another node, when it is added to a subgroup it will not raise the subgroup's *rank*. Although R may not be at the top of the total group including J, it should be evident that since it lowers the *rank* of a subgroup, it will lower the *rank* of the complete group. Thus if the *rank* of J's group changes, it can only change by decreasing. ∎

Lemma A.2 *For any join J and restriction or secondary join predicate R in a plan tree, if the Predicate Migration Algorithm ever places R above J in any stream, it will never subsequently place J below R.*

Proof. Assume the contrary, and consider the first time that the Predicate Migration Algorithm pushes a restriction or secondary join predicate R back below a join J. This can happen only because the *rank* of the group that J is now in is higher than the *rank* of J's group at the time R was placed above J. By Lemma A.1, pulling up nodes can not raise the *rank* of J's group. Since this is the first time that a node is pushed down, it is not possible that the *rank* of J's group has gone up, a contradiction. ∎

As a corollary to Lemma A.2, we can modify the `parallel_chains` routine: instead of actually sorting a module, it can simply pull up each restriction or secondary join above as many groups as possible, thus potentially lowering the number of comparisons in the routine. This optimization is implemented in POSTGRES.

Theorem 1 *Given any plan tree as input, the Predicate Migration Algorithm is guaranteed to terminate in polynomial time, producing a join-order equivalent tree in which each stream is semantically correct and well-ordered.*

Proof. From Lemma A.2, we know that after the pre-processing phase, the Predicate Migration Algorithm only moves predicates upwards in a stream. In the worst-case scenario, each pass through the `do` loop of `predicate_migration` makes minimal progress, *i.e.* it pulls a single predicate above a single join in only one stream. Each predicate can only be pulled up as far as the the top of the tree, *i.e.* h times, where h is the height of the tree. Thus the Predicate Migration Algorithm visits each stream at most hk times, where k is the number of expensive restriction and secondary join predicates in the tree. The tree has r streams, where r is the number of relations referenced in the query, and each time the Predicate Migration Algorithm visits a stream of height h it performs Monma and Sidney's $O(h \log h)$ algorithm on the stream. Thus the Predicate Migration Algorithm terminates in $O(hkrh \log h)$ steps.

Now the number of restrictions, the height of the tree, and the number of relations referenced in the query are all bounded by n, the number of operators in the plan tree. Hence a trivial upper bound for the Predicate Migration Algorithm is $O(n^4 \log n)$. Note that this is a very conservative bound, which we present merely to demonstrate that the Predicate Migration Algorithm is of polynomial complexity. In general the Predicate Migration Algorithm should perform with much greater efficiency. After some number of steps in $O(n^4 \log n)$, the Predicate Migration Algorithm will have terminated, because each stream will be optimally ordered under the constraints of the given join order and semantic correctness. ∎

We have now seen that the Predicate Migration Algorithm correctly orders each stream within a polynomial number of steps. All that remains is to show that the resulting tree is in fact optimal. We do this by showing the that there is only one tree of well-ordered streams, and that such a tree is in fact of minimal cost.

Theorem 2 *For every plan tree T_1 there is a unique join-order equivalent plan tree T_2 with only well-ordered streams, and T_2 is a minimal cost tree that is join-order equivalent to T_1.*

Proof. Theorem 1 demonstrates that for each tree there exists a join-order equivalent tree of well-ordered streams (since the Predicate Migration Algorithm is guaranteed to terminate). To prove that the tree is unique, we proceed by induction on the number of join nodes in the tree. Following the argument of Lemma 2, we assume that all groups are of distinct *rank*.

Base case: The base case of zero join nodes is a simply a Scan node followed by a series of restrictions, which can be uniquely ordered as shown in Lemma 1.

Induction Hypothesis: For any tree with k join nodes or less, the lemma holds.

Induction: We consider two join-order equivalent plan trees, T and T', each having $k+1$ join nodes and well-ordered, semantically correct streams. We will show that these trees are identical, hence proving the uniqueness property of the lemma.

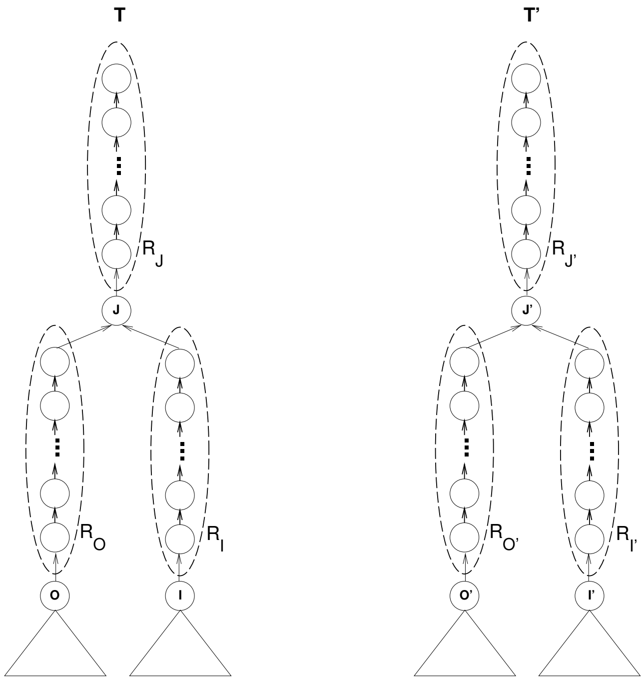

Figure 4: Two Join-Order Equivalent Plan Trees with Semantically Correct, Well-Ordered Streams

As illustrated in Figure 4, we refer to the uppermost join nodes of T and T' as J and J' respectively. We refer to the uppermost join or scan in the outer and inner input streams of J as O and I respectively (O' and I' for J'). We denote the set of restrictions and secondary join predicates above a given join node p as R_p, and hence we have, as illustrated, R_J above J, $R_{J'}$ above J', R_O between O and J, etc. We call a predicate in such a set *mobile* if there is a join below it in the tree, and the predicate refers to the attributes of only one input to that join. Mobile predicates can be moved below such joins without affecting the semantics of the plan tree. First we establish that the subtrees O and O' are identical. The corresponding proof for I and I' is analogous.

Consider a plan tree O^+ composed of subtree O with a *rank*-ordered set R_{O+} of predicates above it, where R_{O+} is made up of the union of R_O and those predicates of R_J that do not refer to attributes from I. If O and J are grouped together in T, then let the cost and selectivity of O in O^+ be modified to include the cost and selectivity of J. Consider an analogous tree $O^{+\prime}$, with $R_{O+'}$ being composed of the union of $R_{O'}$ and those predicates of $R_{J'}$ that do not refer to I'. Modify the cost and selectivity of O' in $O^{+\prime}$ as before. It should be clear that O^+ and $O^{+\prime}$ are join-order equivalent trees of less than k nodes. Since T and T' are assumed to have well-ordered streams, then clearly so do O and O'. Hence by the induction hypothesis O^+ and $O^{+\prime}$ are identical, and therefore the subtrees O and O' are identical.

Thus the only differences between T and T' must occur above O, I, O', and I'. Now since the sets of predicates in the two trees are equal, and since O and O', I and I' are identical, clearly $R_O \cup R_I \cup R_J = R_{O'} \cup R_{I'} \cup R_{J'}$. Semantically, predicates can only travel downward along a single stream, and hence we see that $R_O \cup R_J = R_{O'} \cup R_{J'}$, and $R_I \cup R_J = R_{I'} \cup R_{J'}$. Thus if we can show that $R_J = R_{J'}$, we will have shown that T and T' are identical.

Assume the contrary, *i.e.* that $R_J \neq R_{J'}$. Then without loss of generality $R_J - R_{J'} \neq \emptyset$. Recalling that both trees are well-ordered, this implies that either

- The minimal-*rank* mobile predicate of R_J has lower *rank* than the minimal-*rank* mobile predicate of $R_{J'}$, or
- $R_{J'}$ contains no mobile predicates.

In either case, we see that R_J is a superset of $R_{J'}$.

Knowing that, we proceed to show that R_J cannot contain any predicate not in $R_{J'}$, hence demonstrating that $R_J = R_{J'}$, and therefore that T is identical to T', completing the proof.

We have assumed that T and T' have only well-ordered streams. The only distinction between T and T' is that more predicates have been pulled above J than above J'. Thus it must be possible to pull the predicates in $J - J'$ above J' and make T' identical to T. Since T' is well ordered, this means pulling a predicate p above J' such that *rank*(p) is strictly less than the *rank* of the group containing J'. Recall from Lemma A.1 that after pulling p above J the *rank* of J's group cannot increase. Each subsequent "pullup" can only change the *rank* of J's group by decreasing it, and hence after we transform T' to be identical to T, we can still be assured that *rank*(p) is less than the *rank* of the group containing J'. Since p is above J after these pullups, then we know that that when T' is transformed to be identical to T, T' has a stream that is not well-ordered. This contradicts the assumption that T is well-ordered, and hence it must be that T and T' were identical to begin with; *i.e.* there is only one unique tree with well-ordered streams.

Having established the uniqueness of the well-ordered tree, it is easy to see that this tree is of minimal cost. Assume the contrary, *i.e.* that there is a tree T of minimal cost that has a stream that is not well ordered. Then in this stream there is a group $\overline{v}$ adjacent to a group $\overline{w}$ such that $\overline{v}$ and $\overline{w}$ are not well-ordered, and $\overline{v}$ and $\overline{w}$ may be swapped without violating the constraints. Since swapping the order of these two groups affects only the cost of the nodes in $\overline{v}$ and $\overline{w}$, the total cost of T can be made lower by swapping $\overline{v}$ and $\overline{w}$, contradicting our assumption that T was of minimal cost. ■

Lemma 3 *For a restriction or secondary join predicate R in a subtree, if the* rank *of R is greater than the* rank *of any join in any plan for the subtree, then in the optimal tree R will appear above the highest join in the subtree.*

Proof. Recall that *rank*$(p_1) >$ *rank*$(\overline{p_1 p_2})$. Thus the highest-*rank* group containing nodes from the subtree is of *rank* less than or equal to the *rank* of the highest-*rank* join node in the subtree. A restriction of higher *rank* than the highest-*rank* join node is therefore certain to be placed above the subtree in the optimal tree. ■