# Practical Predicate Placement

**Joseph M. Hellerstein***
University of California, Berkeley
*joey@cs.wisc.edu*

**Abstract.** Recent work in query optimization has addressed the issue of placing expensive predicates in a query plan. In this paper we explore the predicate placement options considered in the Montage DBMS, presenting a family of algorithms that form successively more complex and effective optimization solutions. Through analysis and performance measurements of Montage SQL queries, we classify queries and highlight the simplest solution that will optimize each class correctly. We demonstrate limitations of previously published algorithms, and discuss the challenges and feasibility of implementing the various algorithms in a commercial-grade system.

## 1 Introduction

Relational Database Management Systems have begun to allow user-defined data types and operators to be utilized in ad-hoc queries. Simultaneously, Object-Oriented DBMSs have begun to offer ad-hoc query facilities, allowing declarative access to objects and methods that were previously only accessible through hand-coded, imperative applications. These two supposedly distinct approaches to data management are converging on similar sets of new problems in query optimization and execution.

One of the major problems faced by these systems is that the common relational heuristic of "selection pushdown" (see *e.g.* [Ull88]) is no longer advantageous in all situations. Selection pushdown requires query processing to perform selections before performing joins. This heuristic fails when expensive computations are embedded in selections: in such scenarios the selections may be more costly and less selective than joins, and thus it may be desirable to postpone checking selections until after performing joins. A variety of papers have dealt with this problem in various forms, and two basic algorithms for query planning have resulted, one first described in [CGK89], another in [Hel92]. As we will see in this paper, each of these approaches has limitations in practice, which necessitate some compromises. We add to these rather complex algorithms a family of simpler heuristics, illustrating for each one the class of queries which it can effectively optimize.

The intention of this work is to guide query optimizer developers in choosing a practical solution whose implementation and performance complexity is suited to their application domain. As a reference point, we describe our experience implementing the Predicate Migration algorithm [Hel92, HS93a] and simpler heuristics in the Montage Object-Relational DBMS (formerly called Miró [Sto93]). We compare the performance of the various heuristics on different classes of queries, attempting to highlight the simplest solution that works for each class.

Table 1 provides a quick reference to the algorithms, their applicability and limitations. When appropriate, the 'C Lines' field gives a rough estimate of the number of lines of C code (with comments) needed in Montage's System R-style optimizer to support each algorithm.

### 1.1 Related Work

The System R project faced the issue of expensive predicate placement early on, since their SQL language had the notion of subqueries, which (especially when correlated) are a form of expensive selection. At the time, however, the emphasis of their optimization work was on finding optimal join orders and methods, and there was no published work on placing expensive predicates in a query plan. System R and R* both had simple heuristics for placing an expensive subquery in a plan. In both systems, subquery evaluation was postponed until simple predicates were evaluated. In

*Current address: Department of Computer Sciences, 1210 W. Dayton St., Madison, WI, 53706. This work was initiated while the author was at the University of California, Berkeley. In part, 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.

| Algorithm | Works For... | C Lines | Comments |
|---|---|---|---|
| PushDown+ | queries without expensive predicates and queries without joins | 900 | OK for single table queries, and thus some OODBMSs. |
| PullUp | queries with either free or *very* expensive selections | 1400 | OK for MMDBMSs with standard primary join predicates. |
| PullRank | queries with at most one join and standard primary join predicates | 2000 | Also used as a preprocessor for Predicate Migration. |
| Predicate Migration | queries with standard primary join predicates | 3000 | Widely effective. Can cause enlargement of plan space. |
| LDL | queries where the optimal plan has no costly predicates over an inner | NA | Impractical to integrate with a System R optimizer. |
| Exhaustive | all queries, including those with expensive primary joins | 1100 | Prohibitive computational complexity. |

Table 1: Summary of Algorithms

R*, the issue of balancing selectivity and cost was discussed, but in the final implementation subqueries were ordered among themselves by a cost-only metric: the more difficult the subquery was to compute, the later it would be applied. Neither system considered pulling up subquery predicates from their lowest eligible position. [LH93]

An orthogonal issue related to predicate placement is the problem of rewriting predicates into more efficient forms [CS93, CYY+92]. In such semantic optimization work, the focus is on rewriting expensive predicates in terms of other, cheaper predicates. This is similar to the query rewrite facility of Starburst [PHH92]. It is important to note that this issue is indeed orthogonal to our problems of query planning — once predicates have been rewritten into cheaper forms, they still need to be optimally placed in a query plan.

### 1.2 Structure of the Paper

Section 2 provides background required for the rest of the paper. Section 3 revisits the two published algorithms for predicate placement, and highlights their limitations. Section 4 presents four different predicate placement heuristics and algorithms considered for Montage, and discusses the class of queries for which each is well suited. Section 5 describes our implementation experience. Section 6 summarizes and gives directions for future research.

## 2 Background: Terminology and the Benchmark Database

In order to discuss predicate placement in a practical context, we need to carefully define what we mean by a predicate. In a typical SQL system, the `WHERE` clause of a query is converted to conjunctive normal form, and each conjunct is considered a separate predicate. Predicates referencing a single table are *selection* predicates, and predicates referencing multiple tables are *join* predicates. There are two types of join predicates. A *primary* join predicate is one that is intrinsic to the evaluation of the chosen join method; typically it is a predicate matching an index, or a predicate over sorted or hashed attributes. Each join has at least one primary join predicate.[1] Additional join predicates are called *secondary*, and they are much like selections — they can be placed anywhere in the query plan, as long as they remain higher in the query plan tree than their associated primary join predicate. In the remaining sections, discussion of pulling up selections also applies to secondary join predicates.

In the course of the paper, we will be using SQL queries to demonstrate the strengths and limitations of our algorithms. The database schema for these queries is based on that of Hong and Stonebraker [HS93b], with cardinalities scaled up by a factor of 10. All tuples are 100 bytes wide. Attributes whose names start with the letter '`u`' are unindexed, while all other attributes have B-tree indices defined over them. Numbers in attribute names indicate the approximate number of times each value is repeated in the attribute. For example, each value in a column named `ua20` is duplicated about 20 times. Some physical characteristics of the relations appear in Table 2. The entire database, with indices and catalogs, was about 110 Megabytes in size.

In the example queries below, we refer to user-defined functions. Numbers in the function names describe the cost of the function in terms of random (i.e. non-sequential) database I/Os. For example, the function costly100 takes as much time per invocation as the I/O time used by a query which touches 100 unclustered tuples in the database. In our experiments, however, the functions did not perform any computation; rather, we counted how many times each function was invoked,

[1] For nested loop join without an indexed inner, an arbitrary join predicate can be chosen as primary. Typically it should be one of minimal *rank*, as described in Section 4.1.

| Table | #Tuples | #8K Pgs |
|---|---|---|
| t1 | 45 900 | 574 |
| t2 | 77 310 | 957 |
| t3 | 8 730 | 110 |
| t4 | 40 150 | 498 |
| t5 | 71 560 | 886 |
| t6 | 2 980 | 39 |
| t7 | 34 390 | 427 |
| t8 | 65 810 | 815 |
| t9 | 97 230 | 1 203 |
| t10 | 28 640 | 356 |

Table 2: Benchmark Database

multiplied that number by the function's cost, and added the total to the measurement of the running time for the query. This allowed us to benchmark queries with very expensive functions in a reasonable amount of time; otherwise, comparisons of good plans to suboptimal plans would have been prohibitively time-consuming.

Our performance measurements were done in a development version of Montage, similar to the publicly available version 2.0. Montage was run on a Sparcstation 10/40, equipped with 32Mb of main memory, 139Mb of swap space, and 833Mb of file system space, running SunOS Release 4.1.3. All performance numbers reported below are relative, not absolute.

## 3 Predicate Placement Algorithms Revisited

Two basic approaches have been published for handling expensive predicate placement. The first approach was pioneered in the LDL logic database system [CGK89], and was later proposed for an extended relational model in [YKY+91]. We refer to this as the LDL algorithm. The other approach, called Predicate Migration, is outlined in [HS93a], and described in full in [Hel92]. Neither algorithm actually produces optimal plans in all scenarios. In this section we explore the limitations of each algorithm, and proposals for handling those limitations in practice.

### 3.1 The LDL Algorithm

To illustrate the LDL algorithm, consider the following example query:

```
SELECT  *
  FROM  R, S
 WHERE  R.c1 = S.c1
   AND  p(R.c2)
   AND  q(S.c2);
```

In the query, p and q are expensive user-defined boolean functions.

Assume that the optimal plan for the query is as

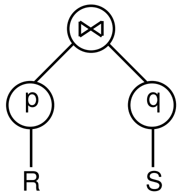


Figure 1: A query plan with expensive selections p and q.

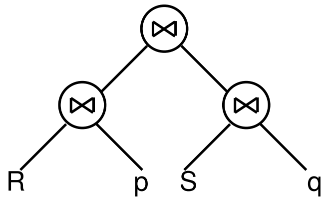


Figure 2: The same query plan, with the selections modeled as joins.

pictured in Figure 1, where both p and q are placed directly above the scans of R and S respectively. In the LDL algorithm, p and q are treated not as selections, but as joins with virtual relations of infinite cardinality. In essence, the query is transformed to:

```
SELECT  *
  FROM  R, S, p, q
 WHERE  R.c1 = S.c1
   AND  p.c1 = R.c2
   AND  q.c1 = S.c2;
```

with the joins over p and q having cost equivalent to the cost of applying the functions. At this point, the LDL approach applies a traditional join-ordering optimizer to plan the rewritten query. This does not integrate well with a System R-style optimization algorithm, however, since LDL increases the number of joins to order, and System R's complexity is exponential in the number of joins. Thus [KZ88] proposes using the polynomial-time IK-KBZ [IK84, KBZ86] approach for optimizing the join order. Unfortunately, both the System R and IK-KBZ optimization algorithms consider only left-deep plan trees, and *no left-deep plan tree can model the optimal plan tree of Figure 1.* That is because the plan tree of Figure 1, with selections p and q treated as joins, looks like the *bushy* plan tree of Figure 2. Effectively, the LDL approach is forced to always pull expensive selections up from the inner relation of a join, in order to get a left-deep tree. Thus the LDL approach can often err by making over-eager pullup decisions.

This deficiency of the LDL approach can be overcome in a number of ways. A System R optimizer can be modified to explore the space of bushy trees, but this increases the complexity of the LDL algorithm yet further. No known modification of the IK-KBZ optimizer can handle bushy trees. Yajima *et al.* [YKY+91] successfully integrate the LDL algorithm with an IK-KBZ optimizer, but they use an exhaustive mechanism that requires time exponential in the number of expensive selections.

### 3.2 Predicate Migration Revisited

The Predicate Migration algorithm presented in [HS93a] only works when join costs fit a linear cost model. In this section we update that cost model, and illustrate how that affects the practical applicability of Predicate Migration.

The "global" cost model presented in [HS93a] proved to be inaccurate at modelling query plans in practice, and was discarded in Montage. In the global cost model, the selectivity of a join node has the same effect on both inputs to the join. In practice this is inaccurate. Consider an equi-join of two relations $R$ and $S$ on primary keys, where $R$ has cardinality 100 and $S$ has cardinality 1000. According to [SAC+79], the selectivity of this join is $\frac{1}{1000}$, because we can expect each tuple of $R$ to find one match in $S$. But note that the join's output does not contain $\frac{1}{1000}$ of the tuples from either $R$ or $S$; it in fact selects *all* tuples of $R$, and one-tenth of the tuples of $S$. Thus in our estimates we need to allow the selectivity of a join node to be different for each input stream to the join, something the global cost model cannot capture.

Instead of the global cost model, Montage uses a more flexible estimate of selectivity that can be different for each input, and a simpler (non-"global") estimate of cost per tuple of each input. These modifications do not affect the Predicate Migration algorithm or the proofs of optimality in [Hel92]. Given two relations $R$ and $S$, and a join predicate $J$ of selectivity $s$ over them, we represent the selectivity of $J$ over $R$ as $s \cdot \{S\}$, where $\{S\}$ is the number of tuples that are passed into the join from $S$. Similarly we represent the selectivity of $J$ over $S$ as $s \cdot \{R\}$. In Section 5.2 we review the accuracy of these estimates in practice.

The cost of a selection predicate is its cost per tuple, as stored in the system metadata. The cost of a join predicate per tuple (the "differential" cost) also needs to be computed, and this is only possible if one assumes a linear cost model for joins. Particularly, for relations $R$ and $S$ the join costs must be of the form $k\{R\}+l\{S\}+m$; we do not allow any term of the form $j\{R\}\{S\}$. We proceed to demonstrate that this strict cost model is sufficiently robust to cover the usual join algorithms.

Recall that we treat traditional simple predicates as being of zero-cost; similarly here we ignore the CPU costs associated with joins. Taking this into account, the costs of merge and hash joins given in [Sha86] fit our criterion.[2] For nested-loop join with an indexed inner relation, the cost per tuple of the outer relation is the cost of probing the index (typically 3 I/Os or less), while the cost per tuple of the inner relation is essentially zero — since we never scan tuples of the inner relation that do not qualify for the join, they are filtered with zero cost. So nested-loop join with an indexed inner relation fits our criterion as well.

The trickiest issue is that of nested-loop join without an index. In this case, the cost is $j\{R\}|S| + k\{R\} + l\{S\}+m$, where $|S|$ is the number of blocks scanned from the inner relation $S$. Note that the number of blocks scanned from the inner relation is a constant *irrespective of expensive selections on the inner relation*. That is, in a nested-loop join, for each tuple of the outer relation one must scan every disk block of the inner relation, regardless of whether expensive selections are pulled up from the inner relation or not. So nested-loop join does indeed fit our cost model: $|S|$ is a constant regardless of where expensive predicates are placed, and the equation above can be written as $(j|S| + k)\{R\} + l\{S\} + m$.[3] Therefore we can accurately model the differential costs of all the typical join methods per tuple of an input.

Linear cost models have been evaluated experimentally for nested-loop and merge joins, and were found to be relatively accurate estimates of the performance of a variety of commercial systems [DKS92]. Unfortunately, our strict linear cost model does not apply to joins in which the primary join predicate is expensive. In such joins, the cost formula has an additional term $c_p\{R\}\{S\}$, where $c_p$ is the cost of evaluating the expensive join predicate on a tuple. In order to integrate Predicate Migration with such joins, one needs to sacrifice some accuracy in cost estimation. We postpone discussion of cost estimations for expensive primary joins until Section 5.2, since our estimation technique was chosen as a result of the experiments described below.

## 4 Predicate Placement Schemes, and The Queries They Optimize

In this section we analyze four algorithms for handling expensive predicate placement, each of which can be easily integrated into a System R-style query optimizer. We begin with the assumption that all predicates are initially placed as low as possible in a plan tree, since this is the typical default in existing systems.

### 4.1 PushDown with Rank-Ordering

In our version of the traditional selection pushdown algorithm, we add code to order selections. This

[2] Actually, we ignore the $\sqrt{S}$ savings available in merge join due to buffering. We thus slightly over-estimate the costs of merge join for the purposes of predicate placement.

[3] This assumes that plan trees are left-deep, which is true for most systems including Montage. Even for bushy trees this is not a significant limitation: one would be unlikely to have nested-loop join with a bushy inner, since one might as well sort or hash the inner relation while materializing it.

Query 1:

```
SELECT  t10.a1
  FROM  t10, t3
 WHERE  t3.a1 = t10.ua1
   AND  costly100(t10.ua1) < 0;
```

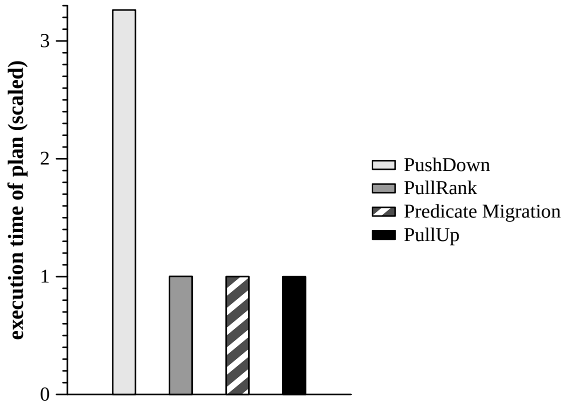


Figure 3: Query execution times for Query 1.

enhanced heuristic guarantees optimal plans for queries on single tables.

The cost of invoking each selection predicate on a tuple is estimated through system metadata. The selectivity of each selection predicate (*i.e.* the percentage of tuples expected to satisfy the predicate) is similarly estimated, and selections over a given relation are ordered in ascending order of the metric

$$rank = \frac{selectivity - 1}{cost}.$$

Such ordering is easily shown to be optimal for selections [HS93a], and intuitively makes sense: the lower the selectivity of the predicate, the earlier we wish to apply it, since it will filter out many tuples. Similarly, the cheaper the predicate, the earlier we wish to apply it, since its benefits may be reaped at a low cost.

Thus a crucial first step in optimizing queries with expensive selections is to order selections by rank. This represents the minimum gesture that a system can make towards optimizing such queries, providing significant benefits for any query with multiple selections. It can be particularly useful for current OODBMSs, in which the typical ad-hoc query is a collection scan, not a join. For systems supporting joins, however, PushDown may often produce very poor plans, as shown in Figure 3. All the remaining algorithms order their selections by rank, and we will not mention selection-ordering explicitly from this point on. In the remaining sections we focus on how the other algorithms order selections with respect to joins.

### 4.2 PullUp

PullUp is the converse of PushDown. In PullUp, all selections with non-trivial cost are pulled to the very top of each subplan that is enumerated during the System R algorithm; this is done before the System R algorithm chooses which subplans to keep and which to prune. The result is equivalent to removing the expensive predicates from the query, generating an optimal plan for the modified query, and then pasting the expensive predicates onto the top of that plan.

PullUp represents the extreme in eagerness to pull up selections, and also the minimum complexity required, both in terms of implementation and running time, to intelligently place expensive predicates among joins. Most systems already estimate the selectivity of selections, so in order to add PullUp to an existing optimizer, one needs to add only three simple services: a facility to collect cost information for predicates, a routine to sort selections by rank, and code to pull selections up in a plan tree.

Though this algorithm is not particularly subtle, it can be a simple and effective solution for those systems in which predicates are either negligibly cheap (*e.g.* less time-consuming than an I/O) or extremely expensive (*e.g.* more costly than joining a number of relations in the database). It is difficult to quantify exactly where to draw the lines for these extremes in general, however, since the optimal placement of the predicates depends not only on the costs of the selections, but also their selectivities, and on the costs and selectivities of the joins. Selectivities and join costs depend on the sizes and contents of relations in the database, so this is a data-specific issue. PullUp may be an acceptable technique in Main Memory Database Systems, for example, or in disk-based systems which store small amounts of data on which very complex operations are performed. Even in such systems, however, PullUp can produce very poor plans if primary joins are expensive, or if join selectivities are greater than 1. The latter problem can be avoided by using function caching, described in section 5.1.

Query 2 (Figure 4) is the same as Query 1, except `t9` is used instead of `t3`. This minor change causes PullUp to choose a suboptimal plan. In this case, since `t9.ua1` has more values than `t10.ua1`, the join of `t9` and `t10` has selectivity 1 over `t10`. As a result, pulling up the costly selection provides no benefit, and increases the cost of the join of `t9` and `t10`. All the algorithms pick the same join method, but PullUp incorrectly places the costly predicate above the join. Note, however, that this error is nearly insignificant, especially in comparison to the error made by PushDown in Query 1. This is because the costly100 function requires 100 random I/Os per tuple, while a join typically costs at most a few I/Os per tuple. The lesson here is that *in general, if primary joins are not expensive, over-eager pullup is less dangerous than under-eager pullup.* As a heuristic, it is safer to overdo a cheap operation than an expensive

Query 2:

```
SELECT  t10.a1
  FROM  t10, t9
 WHERE  t9.a1 = t10.ua1
   AND  costly100(t10.ua1) < 0;
```

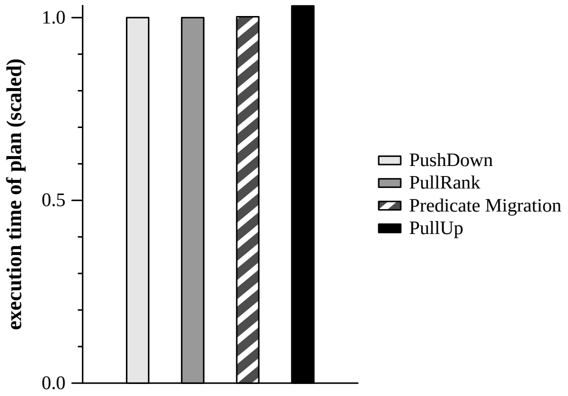


Figure 4: Query execution times for Query 2.

Query 3:

```
SELECT  t10.a1
  FROM  t9, t10
 WHERE  t9.a1 = t10.ua1
   AND  costly1(t10.ua1) < 0;
```

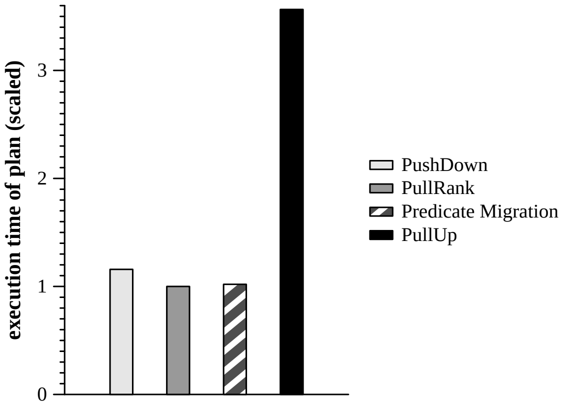


Figure 5: Query execution times for Query 3.

one.

On the other hand, one would like to make as few over-eager pullup decisions as possible. As we see in Figure 5, over-eager pullup can cause significant performance problems for some queries, even though it is a "safer bet" in general than under-eager pullup. In the remaining heuristics, we attempt to find a happy medium between PushDown and PullUp.

### 4.3 PullRank

Like PullUp, the PullRank heuristic works as a subroutine of the System R algorithm: every time a join is constructed for two (possibly derived) input relations, PullRank examines the selections over the two inputs. Unlike Pullup, PullRank does not always pull selections above joins; it makes decisions about selection pullup based on rank. The cost and selectivity of the join are calculated for both the inner and outer input, generating an inner-rank and outer-rank for the join. Any selections on the inner input that are of higher rank than the join's inner-rank are pulled above the join. Similarly, any selections on the outer input that are of higher rank than the join's outer rank are pulled up.

This algorithm is not substantially more difficult to implement than the PullUp algorithm — the only addition is the computation of costs and selectivities for joins, as described in Section 3.2. Because of its simplicity, and the fact that it does rank-based ordering, we had hoped that PullRank would be a very useful heuristic. Unfortunately, it proved to be ineffective in many cases. As an example, consider the plan tree for Query 4 in Figure 6. In this plan, the outer rank of $J_1$ is greater than the rank of the costly selection, so PullRank would not pull the selection above $J_1$. However, the outer rank of $J_2$ is low, and it may be

Query 4:

```
SELECT  t3.a100
  FROM  t3, t6, t10
 WHERE  t10.ua1 = t6.a1
   AND  t3.ua1 = t10.a1
   AND  costly100(t3.a1) < 10;
```

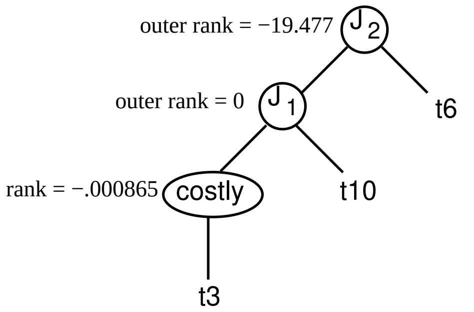


Figure 6: A three-way join plan for Query 4.

appropriate to pull the selection above the pair $J_1 J_2$. PullRank does not consider such multi-join pullups. In general, *if join nodes are decreasing in rank while ascending a root-to-leaf path in a plan tree, then it may be necessary to consider pulling up above groups of joins,* rather than one join at a time. PullRank fails in such scenarios.

PullRank is an optimal algorithm for queries with only one join. Unfortunately, PullRank does indeed fail in many multi-join scenarios, as illustrated in Figure 8. Since PullRank cannot pull up the selection in the plan of Figure 6, it chooses a different join order in which the expensive selection can be pulled to the top (Figure 7). This join order chosen by PullRank is not a good one, however, and results in the poor performance shown in Figure 8. The best plan used the join order of Figure 6,

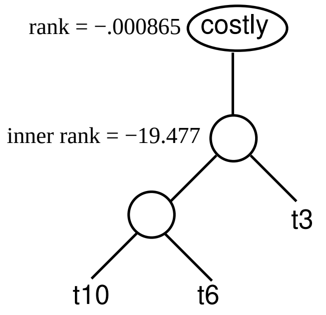


Figure 7: Another three-way join plan for Query 4.

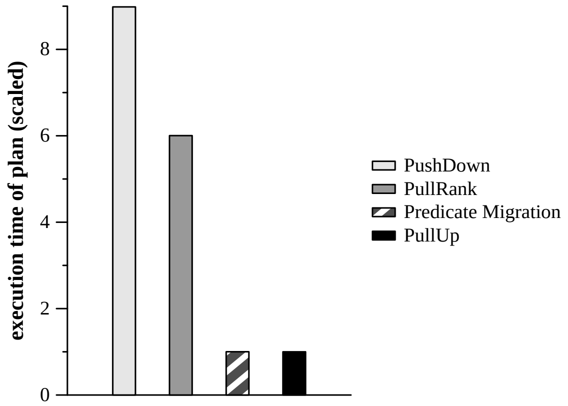


Figure 8: Query execution times for Query 4.

but with the costly selection pulled to the top.

### 4.4 Predicate Migration

The details of the Predicate Migration algorithm are presented in [Hel92] , and we only review them here. The Predicate Migration algorithm repeatedly applies the Series-Parallel Algorithm using Parallel Chains [MS79] to each root-to-leaf path in the plan tree until no progress is made. In essence, Predicate Migration augments PullRank by also considering the possibility that two primary join nodes in a plan tree may be out of rank order, *e.g.* join node $J_2$ may appear just above node $J_1$ in a plan tree, with the rank of $J_2$ being less than the rank of $J_1$ (Figure 6). In such a scenario, it can be shown that $J_1$ and $J_2$ should be treated as a group for the purposes of pulling up selections — they are composed together as one operator, and the group rank is calculated:

$$\begin{aligned}\mathrm{rank}(J_1 J_2) &= \frac{\mathit{selectivity}(J_1 J_2) - 1}{\mathit{cost}(J_1 J_2)} \\ &= \frac{\mathit{selectivity}(J_1)\cdot \mathit{selectivity}(J_2) - 1}{\mathit{cost}(J_1) + \mathit{selectivity}(J_1)\cdot \mathit{cost}(J_2)}.\end{aligned}$$

Selections of higher rank than this group rank are pulled up above the pair. The Predicate Migration algorithm forms all such groups before attempting pullup.

Predicate Migration is integrated with the System R join-enumeration algorithm as follows. We start by running System R with the PullRank heuristic, but one change is made to PullRank: when PullRank finds an expensive predicate and decides *not* to pull it above a join in a subplan, we mark that subplan as *unpruneable*. Subsequently when constructing larger subplans, we mark a subplan unpruneable if it contains an unpruneable subplan within it. The System R algorithm is then modified to save not only those subplans which are min-cost or "interestingly ordered"; it also saves those subplans which are unpruneable. In this way, we assure that if multiple primary joins should become grouped in some plan, we will have maximal opportunity to pull expensive predicates over the group. At the end of the System R algorithm, a set of plans is produced, including the cheapest plan so far, the plans with interesting orders, and the unpruneable plans. Each of these plans is passed through the Predicate Migration algorithm, which optimally places the predicates in each plan. After reevaluating the costs of the modified plans, the new cheapest plan is chosen to be executed.

A drawback of Predicate Migration is the need to consider unpruneable plans. In the worst case, there is an expensive predicate in every subplan that does not get pulled up, so that every subplan is marked unpruneable. In this scenario the System R algorithm exhaustively enumerates the space of join orders, never pruning any subplan. This is still preferable, however, to the LDL approach of adding joins to the query, and has not caused us untoward difficulty in practice. Even in the worst-case scenario where no subplans can be pruned, Montage plans a 5-way join with expensive predicates in under 8 seconds on our SparcStation 10. The payoff of this investment in optimization time is apparent in Figure 9.[4] Note also that Predicate Migration is the only algorithm to correctly optimize each of Queries 1-4.

## 5 Theory to Practice: Implementation Issues

The four algorithms described in the previous section were implemented in the Montage DBMS. In this section we discuss the implementation experience, and some issues which arose in our experiments.

The Montage "Object-Relational" DBMS is based on the publicly available POSTGRES system [SK91].

[4] For Query 5, PullAll used up all available swap space and never completed. This happened because PullAll pulled the costly selection on `t3` above the costly join predicate. The result of this was to call the costly join predicate on all tuples in the cross-product of `t7` and the subtree containing `t3`, `t6`, and `t10`. In addition to resulting in many function calls, this extremely bad plan filled up our entire swap space with predicate cache entries.

Query 5

```
SELECT   t3.a100
  FROM   t3, t6, t10, t7
 WHERE   t10.ua1 = t6.a1
   AND   t3.ua1 = t10.a1
   AND   costly100(t3.a1, t7.a100) = 0
   AND   costly100(t3.a1) < 10;
```

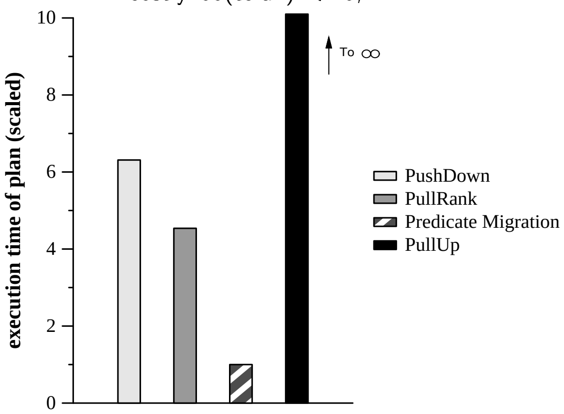


Figure 9: Query execution times for Query 5.

Montage extends POSTGRES in many ways, most significantly (for our purposes) by supporting an extended version of SQL and by bringing the POSTGRES prototype code to an industrial grade of performance and reliability.

The full Predicate Migration algorithm was originally implemented by the author in POSTGRES, an effort that took about two months of work — one month to implement the PullRank heuristic, and another month to implement the Predicate Migration algorithm. Refining and upgrading that code for Montage actually proved more time-consuming than writing it initially for POSTGRES. Since Montage SQL is a significantly more complex language than POSTQUEL, some modest changes had to be made to the code to handle subqueries and other SQL-specific features. More significant, however, was the effort required to debug, test, and tune the code so that it was robust enough for use in a commercial product.

Of the three months spent on the Montage version of Predicate Migration, about one month was spent upgrading the optimization code for Montage. This involved extending it to handle SQL subqueries, making the code faster and less memory-consuming, and removing bugs that caused various sorts of system failures. Another week was required to implement predicate caching (described below), and the remaining time was spent fixing subtle optimization bugs.

Debugging a query optimizer is a difficult task, since an optimization bug does not necessarily produce a crash or a wrong answer; it often simply produces a suboptimal plan. It can be quite difficult to ensure that one has produced a minimal-cost plan. In the course of running the comparisons for this paper, a variety of subtle optimizer bugs were found, including the observation that the global cost model is inaccurate in practice. Typically, bugs were exposed by running the same query under the various different optimization heuristics, and comparing the estimated costs and running times of the resulting plans. When Predicate Migration, a supposedly superior optimization algorithm, produced a higher-cost query plan than a simple heuristic, it typically meant that there was a bug in the optimizer.

The lesson to be learned here is that benchmarking is absolutely crucial to thoroughly debugging a query optimizer. It has been noted that a variety of commercial products still produce very poor plans even on simple queries [Nau93]. Thus benchmarks — particularly *complex query* benchmarks such as TPC-D [TPC93] — are critical debugging tools for DBMS developers. In our case, we were able to easily compare our Predicate Migration implementation against various heuristics, to ensure that Predicate Migration always did at least as well as the heuristics. After many comparisons and bug fixes we found Predicate Migration to be stable, generally producing plans that were as cheap or cheaper than those produced by the simpler heuristics.

## 5.1 Predicate Caching

It is undesirable to repeat complex computations. In Montage, we avoid this through a predicate caching scheme, similar to the one proposed in [HS93a], but different in a few key ways. Contrary to the assertions of [HS93a], predicate caching is not a requirement for using Predicate Migration. Choosing to use predicate caching merely requires changes in rank calculations, to reflect the selectivity of a join on *values* rather than on tuples. Given a join predicate $J$ of selectivity $s$ over $R.c1$ and $S.c2$, the selectivity of $J$ over $R$ is $s \cdot \mathit{number\_of\_values}(S.c2)$, and the selectivity of $J$ over $S$ is $s \cdot \mathit{number\_of\_values}(R.c1)$. In addition, we bound selectivities under predicate caching by 1. This reflects the savings of predicate caching: even if the output of the join has more tuples than either input, it has no values that do not appear in the inputs, and thus it cannot produce more than 100% of the values from each input.

The implementation of predicate caching was relatively simple. In Montage, associated with each expensive predicate is a main-memory dynamic hash table, which stores the return value of the predicate for each binding of its input variables. For example, consider the following SQL query:

```
SELECT   *
  FROM   emp
 WHERE   beard_color(emp.picture) = 'red';
```

Montage stores a hash table keyed on pictures (actually, on 4-byte "handles" to the pictures), with entries being either true, false or NULL (for beardless people). Note that it does not cache the results of the expensive function beard_color (as proposed in [HS93a]); instead it caches the results of the entire predicate. This is important because the return types of functions within predicates may be arbitrarily large derived objects — in the case of subquery functions, for example, they may be sets. As an example of how predicate caching works for subqueries, consider the following:

```
SELECT   name, gpa
  FROM   student
 WHERE   student.mother  IN
         (SELECT   name
            FROM   professor
           WHERE   professor.dept = student.dept);
```

Montage does not cache the result of the subquery for each student tuple. It caches the entire IN predicate. The hash table for the IN predicate is keyed on (student.mother, student.dept) pairs, again with true, false, or NULL entries. Note that attributes of professor are not variables here. Each time we run the subquery we may have different values for student.mother and student.dept, but we will have the same set of values for all the attributes of professor. Essentially, professor is a set-valued constant in the predicate.

As an additional optimization, one need not do predicate caching when it is not beneficial. For each predicate in a query, one can maintain estimates of the number of distinct values of the inputs to the predicate, as well as the number of tuples that will be passed into the predicate. If the ratio of distinct values to tuples is 1, then each input tuple will require explicit computation of the predicate, and predicate caching can provide no benefit. In such cases predicate caching should be avoided. This optimization is planned for Montage, but has not been implemented yet.

Other alternatives exist to our predicate caching implementation, though ours seems to perform reasonably well for our purposes. One can do per-function caching instead, as proposed in [Jhi88] and [HS93a]. Function or predicate caches can be limited in size, using any of a variety of replacement schemes. Queries can be rewritten with Magic-Sets techniques [BMSU86, MFPR90] to avoid the issue of caching entirely, at the expense of extra joins and common subexpressions. Such alternatives do not form a focus of this paper, as this space of possible implementations is large and orthogonal to the space of optimization techniques explored here. Although this is certainly a topic meriting further research, for our purposes we merely wish to point out that it is easy and beneficial to implement a reasonable solution.

### 5.2 Influence of Performance Results on Estimates

The results of our performance experiments influenced the way that we estimate selectivities in Montage, and also the way that we estimate the costs for expensive primary joins. In Section 3.2 we estimated the selectivity of a join over table $S$ as $s \cdot \{R\}$. This was a rough estimate, however, since $\{R\}$ is not well defined — it depends on the selections which are placed on $R$. Thus $\{R\}$ could range from the cardinality of $R$ with no selections, to the minimal output of $R$ after all eligible selections are applied. In Montage, we calculate $\{R\}$ on the fly as needed, based on the number of selections over $R$ at the time we need to compute the selectivity of the join. Since some predicates over $R$ may later be pulled up, this potentially under-estimates the selectivity of the join for $S$. Such an under-estimate results in an under-estimate of the rank of the join for $S$, possibly resulting in over-eager pullup of selections on $S$. This rough selectivity estimation was chosen as a result of our performance observations: it was decided that estimates resulting in somewhat over-eager pullup are preferable to estimates resulting in under-eager pullup. This heuristic is based on the existence of predicate caching in Montage, and also on the assumption that expensive primary joins will not be nearly as common as expensive secondary joins and selections.

A similar issue arises when we heuristically try to cope with expensive primary join predicates in Montage. Given a join with cost $j\{R\}\{S\} + k\{R\} + l\{S\} + m$, consider the differential cost per tuple of $S$ (the cost per tuple of $R$ is analogous). The differential cost per tuple of $S$ is $j\{R\} + l$, but since there may be predicates above relation $R$ which are subject to pullup, we do not know how to estimate $\{R\}$. Again, in Montage we compute $\{R\}$ as the current state of $\{R\}$ while planning, whatever it may be. This means that we may be under-estimating $\{R\}$, since some predicates may later be pulled up from $\{R\}$. By under-estimating $\{R\}$, we are under-estimating the cost of the join per tuple of $S$, which again may cause us to be over-eager in pulling up selections from $S$.

This potential for over-eager pullup is similar, though not as flagrant, as the over-eager pullup in the LDL algorithm. Observe that if the Predicate Migration approach pulls up from inner inputs first, then ranks of joins for inner inputs may be underestimated, but ranks of joins for outer inputs are accurate, since pullup from the inner input has already been completed. This will produce over-eager pullup from inner tables, and accurate pullup from outer tables, as in LDL. This is the approach taken in Montage, and note that unlike LDL, Montage only exhibits this over-eagerness when there are expensive selections on both inputs to a join.

## 6 Conclusion

This paper presents a number of predicate placement algorithms, outlining the class of queries for which each is sufficiently intelligent. Of the algorithms we

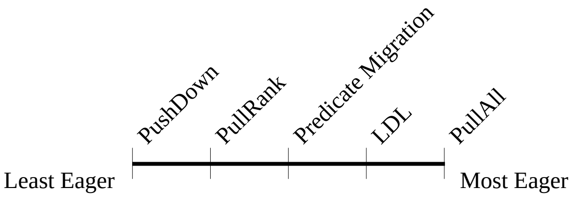


Figure 10: Eagerness of Pullup in Algorithms

tested in Montage, only Predicate Migration provided good query plans over a wide range of queries with expensive predicates. The remaining heuristics each have their realm of applicability, however, and should be considered for systems which need to optimize limited classes of queries.

Adapting the Predicate Migration algorithm for use in a commercial system required some re-working of its cost model. Particularly, it was found that the "global" cost model of [HS93a] was inapplicable in practice. This paper presents a more realistic model for costs and selectivities, and integrates that model into the Predicate Migration algorithm. Some roughness remains in our selectivity estimations, and in our cost estimations for expensive joins. When forced to choose, we opt to risk over-eager pullup of selections rather than under-eager pullup. This is justified by our example queries, which showed that leaving selections too low in a plan was more dangerous than pulling them up too high. The algorithms considered form a spectrum of eagerness in pullup, as shown in Figure 10.

Implementing an effective predicate placement scheme proved to be a manageable task, and doing so exposed many over-simplifications that were present in the original algorithms proposed. This highlights the complex issues that arise in implementing query optimizers. Perhaps the most important lesson we learned in implementing Predicate Migration in Montage was that query optimizers require a great deal of testing before they can be trusted. In practice this means that commercial optimizers should be subjected to complex query benchmarks, and that query optimization researchers should invest time in implementing and testing their ideas in practice.

The limitations of the previously published algorithms for predicate placement are suggestive. Both algorithms suffer from the same problem: the choice of which predicates to pull up from one side of a join both depends on and influences the choice of which predicates to pull up from the other side of the join. This interdependency of separate branches in a query tree suggests a fundamental intractability in predicate placement, that may only be avoidable through the sorts of compromises found in the existing literature.

## 7 Acknowledgments

This paper was made possible through the open-mindedness of Mike Stonebraker, Paula Hawthorn, and the rest of Montage Software. Their willingness to allow research in a commercial setting is a model for the database community. Wei Hong was an invaluable resource in all aspects of the work. Jeff Naughton provided regular advice and support, and was alternately patient and probing as the situation demanded. The author benefited once again from Mike Stonebraker's clear thinking and rich experience. Mike Olson provided essential technical support when the author was frantic, thousands of miles from the reset button. Thanks to Guy Lohman and Laura Haas for historical information on IBM systems, to Eben Haber and Janet Wiener for editorial comments, and to Madhusudhan Talluri, Jeff Naughton and Shivakumar Venkataraman for sharing machine cycles. Finally, thanks to the entire staff of Montage for their help with work and diversion.

## References

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and other Strange Ways to Implement Logic Programs. In *Proc. 5th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–15, Cambridge, March 1986.

[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.

[CS93] Surajit Chaudhuri and Kyuseok Shim. Query Optimization in the Presence of Foreign Functions. In *Proc. 19th International Conference on Very Large Data Bases*, pages 526–541, Dublin, August 1993.

[CYY+92] Hanxiong Chen, Xu Yu, Kazunori Yamaguchi, Hiroyuki Kitagawa, Nobuo Ohbo, and Yuzuru Fujiwara. Decomposition — An Approach for Optimizing Queries Including ADT Functions. *Information Processing Letters*, 43(6):327–333, 1992.

[DKS92] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query Optimization in Heterogeneous DBMS. In *Proc. 18th International Conference on Very Large Data Bases*, Vancouver, August 1992.

[Hel92] Joseph M. Hellerstein. Predicate Migration: Optimizing Queries With Expensive Predicates. Technical Report Sequoia 2000 92/13, University of California, Berkeley, December 1992.

[HS93a] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

[HS93b] Wei Hong and Michael Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases, An International Journal*, 1(1):9–32, January 1993.

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

[Jhi88] Anant Jhingran. A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures. In *Proc. 14th International Conference on Very Large Data Bases*, Los Angeles, August-September 1988.

[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.

[KZ88] Ravi Krishnamurthy and Carlo Zaniolo. Optimization in a Logic Based Language for Knowledge and Data Intensive Applications. In Joachim W. Schmidt, Stefano Ceri, and M. Missikoff, editors, *Proc. International Conference on Extending Data Base Technology*, Advances in Database Technology - EDBT '88. Lecture Notes in Computer Science, Volume 303, Venice, March 1988. Springer-Verlag.

[LH93] Guy M. Lohman and Laura M. Haas. Personal correspondence, November 1993.

[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.

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

[Nau93] Jeff Naughton. Presentation at Fifth International High Performance Transaction Workshop, September 1993.

[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.

[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.

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[SK91] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10), 1991.

[Sto93] Michael Stonebraker. The Miró DBMS. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

[TPC93] TPC. TPC Benchmark$^{TM}$ D (Decision Support). Working Draft 6.0, Transaction Processing Performance Council, August 1993.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[YKY+91] Kenichi Yajima, Hiroyuki Kitagawa, Kazunori Yamaguchi, Nobuo Ohbo, and Yuzura Fujiwara. Optimization of Queries Including ADT Functions. In *Proc. 2nd International Symposium on Database Systems for Advanced Applications*, pages 366–373, Tokyo, April 1991.