OPTIMIZATION AND EXECUTION TECHNIQUES FOR QUERIES WITH EXPENSIVE METHODS

Joseph M. Hellerstein

Technical Report #1304

February 1996

OPTIMIZATION AND EXECUTION TECHNIQUES FOR QUERIES WITH EXPENSIVE METHODS

by

Joseph M. Hellerstein

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1995

Abstract

This dissertation describes techniques for optimizing and executing queries that can commonly arise in extensible database management systems. These systems allow knowledgeable users to define new types, as well as new *methods* (operators) for the types. This flexibility produces an attendant complexity, which must be handled in new ways for an extensible database management system to be efficient.

The focus of the dissertation is on optimizing and efficiently executing queries that contain time-consuming methods. In query optimization, the traditional focus has been on the choice of join methods and orders. Selections have been handled by "pushdown" rules, which apply selections in an arbitrary order before as many joins as possible. This works under the assumption that selection takes no time. However, users of extensible systems can embed complex methods in selections. Thus selections may take significant amounts of time, and the query optimization model must be enhanced. Query execution techniques also require enhancement to handle expensive methods, which should not be evaluated twice on the same inputs. Avoiding this redundancy efficiently requires new perspectives on well-known techniques for related — but subtly different — problems.

The dissertation first addresses optimization issues from a theoretical perspective. We develop an algorithm called *Predicate Migration*, and prove that it produces optimal plans for queries with expensive methods. We then describe our implementation of Predicate Migration in the commercial extensible database management system *Illustra*, and discuss practical issues that affect our earlier

assumptions. We compare Predicate Migration to a variety of simpler optimization techniques, and demonstrate that Predicate Migration is the best general solution to date.

We continue with techniques to avoid redundant computation of expensive methods. We develop a form of hybrid hashing called *Hybrid Cache*, which proves particularly effective for queries with expensive predicates. We isolate new tradeoffs between hashing and sorting, and demonstrate that sorting out-performs Hybrid Cache for expensive methods with large outputs. We conclude with an overview of results and directions for future work, focussing on research perspectives and our experience bringing research into practice in an industrial-strength system.

Acknowledgements

I would not have been able to complete this dissertation without the assistance and friendly supervision of Jeff Naughton. Jeff is a superb advisor and an excellent researcher. He is patient and extremely generous with his time, and he possesses a rare knack for providing students direction without undue coercion. He has always made me aware of the lessons offered by both success and failure, and he has done his best to teach me how to convert apparent failures into constructive results. Jeff is my role model as an advisor; I will do my best to guide my students as well as Jeff has guided me.

The seeds of this dissertation were planted in my Masters work at Berkeley, under the direction of Mike Stonebraker. Mike is a visionary scientist, and this dissertation owes a great deal to his vision: he was the one who originally suggested that I work on optimization of queries with expensive methods. Mike has been extremely good to me throughout my time in graduate school, both at Berkeley and afterwards.

I have been lucky to be involved with Illustra Information Technologies during my graduate career. The experience working with an industrial-strength system has been eye-opening, and I believe has added significant credence to my results. Very few companies would have allowed the research freedom that Illustra granted me. Special thanks are due to Mike Stonebraker and Paula Hawthorn for making this possible. Also, many people at Illustra shared their expertise, time and machines to help make this work possible. Particular thanks go to Wei Hong, Michael Ubell, Mike Olson, Jeff Meredith, Kevin Brown, and Nadene Re for their help.

I have been extremely fortunate to have done my Ph.D. at the best database research university in the world. David DeWitt, Mike Carey, Yannis Ioannidis and Raghu Ramakrishnan have all provided input on my work, and I believe that their wide perspective has been instrumental to my understanding of this work and the field of database research as a whole. None was officially my advisor, but I have looked to all of them for advice. The students at Madison have been terrific colleagues and friends. The Madison database crowd is so large that if I were to list all the people to whom I owe debts of gratitude I would not have room for the technical portion of this dissertation. My officemates Praveen Seshadri, Jussi Myllymaki and Voon-Fee Yong deserve special thanks for putting up with me. Praveen in particular has been a friend and a colleague, and sharing an office with him over the last few years has been one of the great pleasures of my time in graduate school.

Many other folks helped me with this dissertation. My friends from the POSTGRES group at Berkeley — Wei Hong, Jolly Chen, Mike Olson, Jeff Meredith and Mark Sullivan — were very helpful with work, and also were masters of diversionary tactics. My start in database research came during a "pre-doctoral" internship at IBM Almaden labs, during which I learned much of what I know about databases, research and software development. The folks there have been helpful and encouraging ever since. Hamid Pirahesh, Guy Lohman, and Laura Haas all contributed advice and assitance to this dissertation, and Jennifer Widom has been an additional mentor and role model. I have had helpful feedback on this work from other members of the database community as well, including Surajit Chaudhuri, Ravi Krishnamurthy, Goetz Graefe and Jim Gray. I was fortunate to get Eugene Lawler's input on questions relating to job scheduling. James Frew provided advice and code for example queries from the domain of Remote Sensing.

My good friend Irwin Goldman was kind enough to set aside his beets and serve as the external committee member on my defense. His enthusiasm and dedication to research, teaching, roots and bulbs is phenomenal. I owe a special debt of thanks to my roommate Earnie ("Bert") Relerford, who was a patient and kind friend during one of the toughest stretches of my graduate career.

My family has been my greatest source of help and support, and this dissertation is dedicated to

them. Working on my Ph.D. in the same town (even the same building!) as my parents has made the exercise a lot more pleasant than it could have been. My folks have been pillars of support, and their evident pride in their children has always been a source of inspiration; I hope that the completion of this dissertation brings them much *naches*. My sisters have been the best possible role models, as people and as academics. Their successes often seemed to set unreachable precedents, but they have always been generous with advice, anecdotes and encouragement. The final member of my family to thank is my fiancee, Adene Sacks, who has given me immense amounts of love, confidence, cheer and perspective that have kept me healthy and happy throughout this entire experience.

This work was funded in part by 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. This work was also funded by NSF grant IRI-9157357.

Contents

List of Tables

List of Figures

Chapter 1

Opening

One of the major themes of database research over the last 15 years has been the introduction of extensibility into Database Management Systems (DBMSs). Relational DBMSs have begun to allow simple user-defined data types and functions to be utilized in ad-hoc queries [Pir94]. 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 [Cat94]. More recently, Object-Relational DBMSs were developed to unify these supposedly distinct approaches to data management [Ill94, Kim93].

Much of the original research in this area focused on *enabling* technologies, *i.e.* system and language designs that make it possible for a DBMS to support extensibility of data types and methods. Considerably less research explored the problem of making this new functionality efficient.

This dissertation addresses basic efficiency problems that arise in extensible database systems. Specifically, it explores techniques for efficiently processing declarative queries that contain time-consuming methods. Such queries are natural in modern extensible database systems, which were expressly designed to support declarative queries over user-defined types and methods. Note that "expensive" time-consuming methods are natural for complex user-defined data types, which are often large objects that encode significant complexity of information (*e.g.*, arrays, images, sound, video,

maps, circuits, documents, fingerprints, etc.) In order to efficiently process queries containing expensive methods, new techniques are needed both in query optimization and query execution.

1.1 Background: Extensible DBMSs

As Relational DBMSs (RDBMSs) matured in the last decade and a half, it became clear that their inflexible type systems were causing problems for applications with complex data. A simple, fixed type system was originally seen as a positive feature, part of the clean mathematical basis of the relational model. However, as attempts were made to deploy RDBMS solutions for non-business applications, it became clear that the inflexibility of RDBMS type systems was a significant barrier to their broader acceptance.

RDBMSs have a clean, simple data model that naturally supports *declarative*, non-imperative queries and their optimization. Data is stored in relations or "tables". Tuples (the rows of a table) represent the basic records in the RDBMS, and each field in a tuple belongs to a named, typed column representing a particular attribute of the records in the table. The relational query languages support operations to *select* tuples from a table, *project* columns from a table, and *join* multiple tables on connecting predicates. Many RDBMS query languages also allow for tables to be aggregated into groups, over which aggregate functions like `COUNT`, `MIN`, `MAX` and `AVERAGE` can be computed.

However, the type system in an RDBMS is highly restricted. Each field in a tuple contains a datum, whose type must be one of the few *atomic* types supported by an RDBMS: integers, floating point numbers, character strings, monetary units and dates. The collection of types in these systems can not be extended, nor can the set of methods on the types, which usually includes only arithmetic operators, ordering and equality predicates, and simple string-matching predicates. Complex data objects such as audio/video, maps and arrays can not easily be encoded into tables of these atomic types. Moreover, it is extremely difficult to write meaningful queries on such data using simple alphanumeric methods. Even when this is possible, the queries are cumbersome to write and slow to execute.

As a solution to these problems, a main goal of extensible DBMS research was to support a user-extensible set of atomic types and methods. If users could store complex objects as fields in tables, then the query language could be used to support relational operations on the collections of objects, and user-defined methods could support the domain-specific operations on the objects themselves. The hope was that the basic mechanisms of an RDBMS to support selection, projection, join and aggregation queries could remain the same. This hope was largely borne out by the early extensible research systems such as POSTGRES [SK91] and Starburst [HFLP89]. However, by allowing users to define their own, potentially time-consuming methods, these systems generated new problems in query optimization and execution.

1.2 Query Optimization Issues

To illustrate the issues that can arise in processing queries with expensive methods, consider the following example over the satellite image data presented in the Sequoia benchmark for Geographic Information Systems [SFGM93]. The query retrieves names of digital "raster" images taken by a satellite; particularly, it selects the names of images from the first time period of observation that show vegetation in over 20% of their pixels:

Example 1.

```
SELECT   name
  FROM   rasters
 WHERE   rtime = 1
   AND   veg(raster) > 20;
```

In this example, the method veg reads in 16 megabytes of raster image data (infrared and visual data from a satellite), and counts the percentage of pixels that have the characteristics of vegetation (these characteristics are computed per pixel using a standard technique in remote sensing [Fre95].) The veg method is very time-consuming, taking many thousands of instructions and I/O operations to compute. It should be clear that the query will run faster if the selection rtime = 1 is applied before the veg selection, since doing so minimizes the number of calls to veg. A traditional optimizer would

order these two selections arbitrarily, and might well apply the veg selection first. Some additional logic must be added to the optimizer to ensure that selections are applied in a judicious order.

While selection ordering such as this is important, correctly ordering selections within a table-access is not sufficient to solve the general optimization problem of where to place predicates in a query execution plan. Consider the following example, which joins the rasters table with a table that contains notes on the rasters:

Example 2.

```
SELECT   rasters.name, notes.note
  FROM   rasters, notes
 WHERE   rasters.rtime = notes.rtime
   AND   notes.author = 'Clifford'
   AND   veg(rasters.raster) > 20;
```

Traditionally, an optimizer would plan this query by applying all the single-table selections in the WHERE clause before performing the join of rasters and notes. This heuristic, often called "predicate pushdown", is considered beneficial since early selections usually lower the complexity of join processing, and are traditionally considered to be trivial to check [Pal74]. However in this example the cost of evaluating the expensive selection predicate may outweigh the benefit gained by doing selection 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 time-consuming selection veg(rasters.raster) > 20 until after computing the join of rasters and notes.

In general it is not clear how joins and selections should be interleaved in an optimal execution plan, nor is it clear whether the migration of selections should have an effect on the join orders and methods used in the plan. The first few chapters of this dissertation explore these query optimization problems from both a theoretical point of view, and from the experience of producing an industrial-strength implementation for the Illustra Object-Relational DBMS.

1.3 Query Execution Issues

An orthogonal problem arises during query execution. Regardless of where selections are placed in a query plan, the query execution engine should not evaluate an expensive method twice on the same input. In the examples above, if the rasters table had duplicate values in the raster column, a traditional execution strategy would execute the veg method once per *tuple* of the rasters relation, rather than once per distinct *value* in the rasters.raster column. Such redundant computation must be avoided to achieve good query execution performance. The latter part of this dissertation explores a variety of techniques for efficiently avoiding redundant computation of methods.

1.4 Benefits for RDBMS: Subqueries

It is important to note that expensive methods do not exist only in next-generation Object-Relational DBMSs. Current relational languages, such as the industry standard, SQL [ISO93], have long supported expensive predicate methods 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 and execution 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 method. Thus the work presented in this dissertation is applicable to the majority of today's production RDBMSs, which support SQL subqueries but do not intelligently place subquery predicates in a query plan.

1.5 Outline of Dissertation

The remainder of this dissertation is divided into six chapters. Chapter 2 presents the theoretical underpinnings of *Predicate Migration*, an algorithm to optimally place expensive predicates in a query plan. Chapter 3 discusses three simpler alternatives to Predicate Migration, and uses performance of queries in Illustra to demonstrate the situations in which each technique works. Issues of method caching during query execution are discussed in Chapter 4, including the introduction of a variant of hybrid hashing which we call *Hybrid Cache*, and performance studies comparing sorting and hashing techniques for caching. Chapter 5 discusses the related work in the research literature. Concluding remarks appear in Chapter 6, and a bibliography is presented in the last chapter.

1.6 Environment for Experiments

In the course of the dissertation we will examine the behavior of various algorithms via both analysis and experiments. All the experiments presented in the dissertation were run in the commercial Object-Relational DBMS Illustra. A development version of Illustra was used, similar to the publicly released version 2.4.1. Except when otherwise noted, Illustra was run with its default configuration, with the exception of settings to produce traces of query plans and execution times. The machine used was a Sun Sparcstation 10/51 with 2 processors and 64 megabytes of RAM, running SunOS Release 4.1.3. One Seagate 2.1-gigabyte SCSI disk (model #ST12400N) was used to hold the databases. The binaries for Illustra were stored on an identical Seagate 2.1-gigabyte SCSI disk, Illustra's logs were stored on a Seagate 1.05-gigabyte SCSI disk (model #ST31200N), and 139 megabytes of swap space was allocated on another Seagate 1.05-gigabyte SCSI disk (model #ST31200N).

Due to restrictions from Illustra Information Technologies, most of the performance numbers presented in the dissertation are relative rather than absolute: the graphs are scaled so that the lowest data point has the value 1.0. Unfortunately, commercial database systems typically have a clause in their license agreements that prohibits the release of performance numbers. It is unusual for a database

vendor to permit publication of any performance results at all, relative or otherwise [CDKN94]. The scaling of results in this dissertation does not affect the conclusions drawn from the experiments, which are based on the relative performance of various approaches.

Chapter 2

Optimization: Theory

The field of query optimization has been explored in great detail in the relational database literature (see [JK84] for a survey.) The traditional focus of relational query optimization schemes has been on the choice of join methods and join orders. Selections have typically been handled in query optimizers by "predicate pushdown" rules, which apply selections in some arbitrary order before as many joins as possible. These rules work under the assumption that selection is essentially a zero-time operation. However, extensible database systems allow users to define time-consuming methods, which may be used in a query's selection and join predicates. Furthermore, SQL has long supported subquery predicates, which may be arbitrarily time-consuming to check. Thus selections should not be considered zero-time operations, and the model of query optimization must be enhanced.

In general it is not clear how joins and selections should be interleaved in an optimal execution plan, nor is it clear whether the migration of selections should have an effect on the join orders and methods used in the plan. This chapter 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 standard 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 chapter 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 Illustra, integrating it with Illustra's standard System R-style optimizer [SAC+79]. With modest overhead in optimization time, Predicate Migration can reduce the execution time of many practical queries by orders of magnitude. This is illustrated further below.

2.1 Background: Optimizer Estimates

To develop our optimizations, we must enhance the traditional model for analyzing query plan cost. This will involve some modification of the usual metrics for the expense and selectivity of relational operators. This preliminary discussion of our model will prove critical to the analysis below.

A relational query in a language such as SQL 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 *selection predicate* refers only to a single table.

2.1.1 Selectivity

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

$$selectivity(p) = \frac{cardinality(output(p))}{cardinality(input(p))}.$$

Typically these estimations are based on default values and statistics stored by the DBMS [SAC+79], although recent work suggests that inexpensive sampling techniques can be used [LNSS93, HOT88].

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

Table 1: Method expense parameters in Illustra.

Accurate selectivity estimation is a difficult problem in query optimization, and has generated increasing interest in recent years [IP95, FK94, IC91]. In Illustra, selectivity estimation for user-defined methods can be controlled through the `selfunc` flag of the `create function` command [Ill94]. In this dissertation we make the standard assumptions of most query optimization algorithms, namely that estimates are accurate and predicates have independent selectivities.

2.1.2 Differential Cost of User-Defined Methods

In an extensible system such as Illustra, arbitrary user-defined methods may be introduced into both selection and join predicates. These methods can be written in a general programming language such as C, or in a database query language, *e.g.*, SQL. In this section we discuss programming language methods; we handle query language methods and subqueries in Section 2.1.3.

Given that user-defined methods may be written in a general purpose language such as C, it is difficult for the database to correctly estimate the cost of predicates containing these methods, at least initially.[1] As a result, Illustra includes syntax to give users control over the optimizer's estimates of cost and selectivity for user-defined methods.

To introduce a method to Illustra, a user first writes the method in C and compiles it, and then issues Illustra SQL's `create function` statement, which registers the method with the database system. To capture optimizer information, the `create function` statement accepts a number of special flags, which are summarized in Table 1.

[1] After repeated applications of a method, one could collect performance statistics and use curve-fitting techniques to make estimates about the method's behavior.

The cost of evaluating a predicate on a single tuple in Illustra is computed by adding up the costs for all the expensive methods in the predicate expression. Given an Illustra predicate $p(a_1, \ldots, a_n)$, the expense per tuple is recursively defined as:

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

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 method.

Note that the expense of a method reflects the cost of evaluating the method on a single tuple, not the cost of evaluating it for every tuple in a relation. This "cost per tuple" metric is natural to method invocations, but it is less natural for predicates, since predicates take relations as inputs. Predicate costs are typically expressed as a function of the cardinality of their input. Thus the per-tuple cost of a predicate can be thought of as the *differential* cost of the predicate on a relation — *i.e.* the increase in processing time that would result if the cardinality of the input relation were increased by one. This is the derivative of the cost of the predicate with respect to the cardinality of its input.

2.1.3 Differential Cost of Query Language Methods

Since its inception, SQL has allowed 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 selection is the cost e_c of computing and

materializing the subquery once, and the cost e_s of scanning the subquery's result 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 computation and materialization cost, and consider an uncorrelated subquery's differential cost to be e_s.

Correlated subqueries must be recomputed for each tuple that is checked against the subquery predicate, and hence the differential cost for correlated subqueries is e_c. We ignore e_s here since scanning can be done during each recomputation, and does not represent a separate cost. Illustra also allows for user-defined methods that can be written in SQL; these are essentially named, correlated subqueries.

The cost estimates presented here for query language methods 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+84], 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 dissertation, but including them into the framework of the later sections merely requires more careful estimates of the subquery costs.

2.1.4 Estimates for Joins

In our subsequent analysis, we will be treating joins and selections 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 each of the join's input; that is, we need to estimate the differential cost of the join with respect to each input. We are given a join algorithm over outer relation R and inner relation S, with cost function $f(|R|, |S|)$, where $|R|$ and $|S|$ are the numbers of tuples in R and S respectively. From this information, we compute the differential cost of the join with respect to its outer relation as $\frac{\partial f}{\partial |R|}$; the differential cost of the join with respect to its inner relation is $\frac{\partial f}{\partial |S|}$. We will see in Chapter 3 that these partial differentials are constants for all the well-known join algorithms, and hence the cost of a

join per tuple of each input is typically well defined and independent of the cardinality of either input.

We also need to characterize the selectivity of a join with respect to each of its inputs. Traditional selectivity estimation [SAC+79] computes the selectivity s_J of a join J of relations R and S as the expected number of tuples in the output of J (O_J) over the number of tuples in the Cartesian product of the input relations, *i.e.*, $s_J = |O_J|/|R \times S| = |O_J|/(|R| \cdot |S|)$. The selectivity $s_{J(R)}$ of the join with respect to R can be derived from the traditional estimation: it is the size of the output of the join relative to the size of R, *i.e.*, $s_{J(R)} = |O_J|/|R| = s_J \cdot |S|$. The selectivity $s_{J(S)}$ with respect to S is derived similarly as $s_{J(S)} = |O_J|/|S| = s_J \cdot |R|$.

Note that a query may contain multiple 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.* Merge join, hash join, and index nested-loop join all have primary join predicates implicit in their processing. Join predicates that are not applicable in processing the join are merely used to select from its output, and we refer to these as *secondary join predicates.* Secondary join predicates are essentially no different from selection predicates, and we treat them as such. These predicates may then be reordered and even pulled up above higher join nodes, just like selection 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]

2.2 Optimal Plans for Queries With Expensive Predicates

At first glance, the task of correctly optimizing queries containing 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 selection predicates could make traditional plan enumeration techniques prohibitively expensive. In

[2]Nested-loop join without an index is essentially a Cartesian product followed by selection, but inexpensive predicates on an unindexed nested-loop join may be considered primary join predicates, since they will not be pulled up. All expensive join predicates are considered secondary, since they are not essential to the join method and may be pulled up in the plan.

this section we prove the reassuring results that:

1. Given a particular query plan, its selection 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 number of operators in a plan.

2.2.1 Optimal Predicate Ordering in Table Accesses

We begin our discussion by focusing on the simple case of queries over a single table. Such queries can have an arbitrary number of selection predicates, each of which may be a complicated Boolean function over the table's range variables, possibly containing expensive subqueries or user-defined methods. 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 satisfied during the scan are applied first. This is because such predicates have essentially zero cost: they are not actually evaluated, rather the indices are traversed to retrieve only those tuples that qualify.[3] We will represent 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 (differential) 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

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

[3] It is possible to index tables on method values as well as on table attributes [MS86, LS88]. If a scan is done on such a "method" index, then predicates over the method may be satisfied during the scan *without invoking the method.* As a result, these predicates are considered to have zero cost, regardless of the method's expense.

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 by Monma and Sidney [MS79].

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

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

Proof. This result dates back to early work in Operations Research [Smi56], but we review it in our context for completeness.

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

$$\begin{aligned} e_1 &= e_{p_1}t + s_{p_1}e_{p_2}t + \ldots + s_{p_1}s_{p_2}\cdots s_{p_{k-1}}e_{p_k}t \\ &\quad + s_{p_1}s_{p_2}\cdots s_{p_{k-1}}s_{p_k}e_{p_{k+1}}t + \ldots + s_{p_1}s_{p_2}\cdots s_{p_{n-1}}e_{p_n}t. \end{aligned}$$

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

$$\begin{aligned} e_2 &= e_{p_1}t + s_{p_1}e_{p_2}t + \ldots + s_{p_1}s_{p_2}\cdots s_{p_{k-1}}e_{p_{k+1}}t \\ &\quad + s_{p_1}s_{p_2}\cdots s_{p_{k-1}}s_{p_{k+1}}e_{p_k}t + \ldots + s_{p_1}s_{p_2}\cdots s_{p_{n-1}}e_{p_n}t. \end{aligned}$$

By subtracting e_1 from e_2 and factoring we get

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

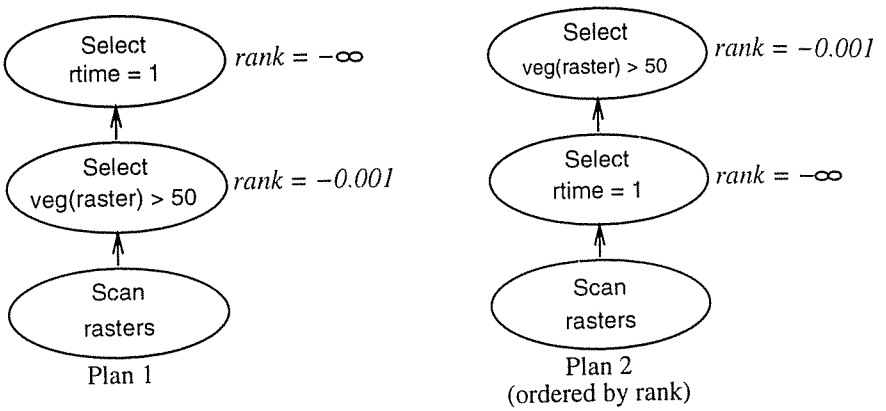

Figure 1: Two execution plans for Example 1.

Now recall that $rank(p_k) > rank(p_{k+1})$, *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 minimum-cost, a contradiction. ■

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 selections, we return to Example 1 from the introduction. We ran the query in Illustra without the *rank*-sort optimization, generating Plan 1 of Figure 1, and with the *rank*-sort optimization, 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

Query Plan	Optimization Time		Execution Time	
	CPU	Elapsed	CPU	Elapsed
Plan 1	0.01 sec	0.02 sec	2 min 18.09 sec	3 min 25.40 sec
Plan 2	0.10 sec	0.10 sec	0 min 0.03 sec	0 min 0.10 sec

Table 2: Performance of plans for Example 1.

execution times were measured for both runs, as illustrated in Table 2. We see that correctly ordering selections can improve query execution time by orders of magnitude, even for simple queries of two predicates and one relation.

2.2.2 Predicate Migration: Placing Selections Among Joins

In the previous section, we established an optimal ordering for selections. In this section, we explore the issue of ordering selections 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 polynomial-time algorithm to optimally place selections and secondary join predicates in a join plan. In Section 2.5 we show how to efficiently integrate this algorithm into a traditional optimizer, so that the optimal plan is chosen from the space of all possible join orders, join methods, and selection placements.

Definitions

The thrust of this section is to handle join predicates in our ordering scheme in the same way that we handle selection predicates: by having them participate in an ordering based on *rank*.

Definition 1 *A* plan tree *is a tree whose leaves are* scan *nodes, and whose internal nodes are either* joins *or* selections. *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 4 (page 30) 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 selection, it can be treated in an identical fashion. For each input to the join one can use the definitions of Section 2.1.4 to compute the differential cost of the join on that stream, the selectivity on that stream, and hence the *rank* of the join in that stream. These estimations require some assumptions about the join cost and selectivity modelling, which we revisit in Chapter 3. For the purposes of this chapter, however, we assume these costs and selectivities are estimated accurately.

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.*

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 $e_1 = te_p + ts_pe_q$. Swapping them results in the cost $e_2 = te_q + ts_qe_p$. 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. ∎

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.

---

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

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

Definition 3 *A plan stream is* semantically incorrect *if some predicate in the stream refers to attributes that do not appear in the predicate's input. Otherwise it is* semantically correct. *A plan tree is semantically incorrect if it contains a semantically incorrect stream; otherwise it is semantically correct.*

Trees can be rendered semantically incorrect by pushing a secondary join predicate below its corresponding primary join, or by pulling a selection 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 selections and secondary join predicates. We formalize this as follows:

Definition 4 *Two plan trees T and T' are* join-order equivalent *if they contain the same set of nodes, and there is a bijection 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.*

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

Our approach to 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 selections 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, 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 for a single stream under these kinds of precedence constraints can be done fairly simply. Their analysis is based on two key results:

1. A set S of plan nodes can be grouped into *job modules*, where a job module is defined as a subset of nodes $S' \subseteq S$ such that for each element n of $S - S'$, n has the same constraint relationship (must precede, must follow, or unconstrained) with respect to all nodes in S'. An optimal ordering for a job module forms a subset of an optimal ordering for the entire stream.

2. For a job module $\{p_1, p_2\}$ 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 that can optimize an arbitrarily constrained stream. The algorithm repeatedly isolates job modules in a stream, optimizing each job module individually, and using the resulting orders for job modules to find a total order for the stream. We use a version of their algorithm as a subroutine in our optimization algorithm:

Predicate Migration Algorithm: *To optimize a plan tree, push all predicates down as far as possible, 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.*

Pseudo-code for the Predicate Migration Algorithm is given in Figure 2, and we provide a brief explanation of the algorithm here. The constraints in a plan tree are not general series-parallel constraints, and hence our version of Monma and Sidney's Series-Parallel Algorithm Using Parallel Chains is somewhat simplified.

The function `predicate_migration` first pushes all predicates down as far as possible. This preprocessing is typically automatic in most System R-style optimizers. The rest of `predicate_migration`

```
/* Optimally locate selections in a query plan tree. */
predicate_migration(tree)
{
  push all predicates down as far as possible;
  do {
       for (each stream in tree)
           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 N constrained to follow J,
        and N is not constrained to precede anything else)
      /* nodes following J form a job module */
      parallel_chains(all nodes constrained to follow J);
  }
  /* stream is now a job module */
  parallel_chains(stream);
  discard any constraints introduced by parallel_chains;
}

/* Monma and Sidney's Parallel Chains Algorithm */
parallel_chains(module)
{
  chain = {nodes in module that form a chain of constraints};
  /* By default, each node forms a group by itself */
  find_groups(chain);
  if (groups in module aren't sorted by their group's ranks)
    sort nodes in module by their group's ranks; /* progress! */
  /* the resulting order reflects the optimized module */
  introduce constraints to preserve the resulting order;
}

/* find adjacent groups constrained to be ill-ordered & merge them. */
find_groups(chain)
{
  initialize each node in chain to be in a group by itself;
  while (any 2 adjacent groups a,b aren't ordered by ascending group rank) {
    form 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 2: Predicate Migration Algorithm.

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 all streams are optimally ordered). The inner loop cycles through all the streams in the plan tree, applying a version of Monma and Sidney's Series-Parallel Algorithm using Parallel Chains.

The `series_parallel` routine traverses the stream from the top down, repeatedly finding modules of the stream to optimize. Given a module, it calls `parallel_chains` to order the nodes of the module optimally. When `parallel_chains` finds the optimal ordering for the module, it introduces constraints to maintain that ordering as a chain of nodes. Thus `series_parallel` uses the `parallel_chains` subroutine to convert the stream, from the top down, into a chain. Once the lowest join node of the stream has been handled by `parallel_chains`, the resulting stream has a chain of nodes and possibly a set of unconstrained selections at the bottom. This entire stream is a job module, and `parallel_chains` can be called to optimize the stream into a single ordering.

Our version of 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 the top down, `series_parallel` always provides correct input to `parallel_chains`.[5] The `parallel_chains` routine first finds groups of nodes in the chain that are constrained to be ordered sub-optimally (*i.e.* by descending *rank*). As shown by Monma and Sidney [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.

---

[5] Note also that for each module S' that `series_parallel` constructs from a stream S, each node of $S-S'$ is constrained in exactly the same way with respect to each node of S': every element of $S - S'$ is either a primary join predicate constrained to precede all of S', or a selection or secondary join predicate that is unconstrained with respect to all of S'. Thus `parallel_chains` is always passed a valid job module.

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.

2.3 Predicate Migration: Proofs of Optimality

Upon termination, the Predicate Migration Algorithm produces a semantically correct tree in which each stream is *well-ordered* according to Monma and Sidney; that is each stream, taken individually, is optimally ordered subject to its precedence constraints. We proceed to prove that the Predicate Migration Algorithm is guaranteed to terminate in polynomial time, and that the resulting tree of well-ordered streams represents the optimal choice of predicate locations for the entire plan tree.

Lemma 3 *Given a join node J in a module, adding a selection or secondary join predicate R to the stream does not increase the rank of J's group.*

Proof. Assume J is initially in a group of k members, $\overline{p_1 \cdots p_{j-1} J p_{j+1} \cdots p_k}$ (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 appropriate. If R is constrained with some member p_i of the group, it is constrained to be *above* p_i (by semantic correctness); no selection or secondary join 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 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 $rank(p_1) > rank(p_2)$, it is easy to show that $rank(p_1) > rank(\overline{p_1 p_2})$. 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 4 *For any join J and selection 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 selection 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 3, 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, and hence R would not have been pushed below J, a contradiction. ■

As a corollary to Lemma 4, we can modify the `parallel_chains` routine: instead of actually sorting a module, it can simply pull up each selection or secondary join above as many groups as possible, thus potentially lowering the number of comparisons in the routine. This optimization is implemented in Illustra.

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

Proof. From Lemma 4, 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 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 selection 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 selections, 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, with each stream well-ordered subject to 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 that:

1. There is only one semantically correct tree of well-ordered streams.
2. Among all semantically correct trees, some tree of well-ordered streams is of minimum cost.
3. Since the output of the Predicate Migration Algorithm is the semantically correct tree of well-ordered streams, it is a minimum cost semantically correct tree.

Theorem 2 *For every plan tree T_1 there is a unique semantically correct, join-order equivalent plan tree T_2 with only well-ordered streams. Moreover, among all semantically correct trees that are join-order equivalent to T_1, T_2 is of minimum cost.*

Proof. Theorem 1 demonstrates that for each tree there exists a semantically correct, 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*; equi-rank groups may be disambiguated via the IDs of the nodes in the groups.

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

Induction Hypothesis: For any tree with k join nodes or less, there is a unique semantically correct, join-order equivalent tree with well-ordered streams.

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

As illustrated in Figure 3, 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 selections 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 $R_{O^{+\prime}}$ 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.

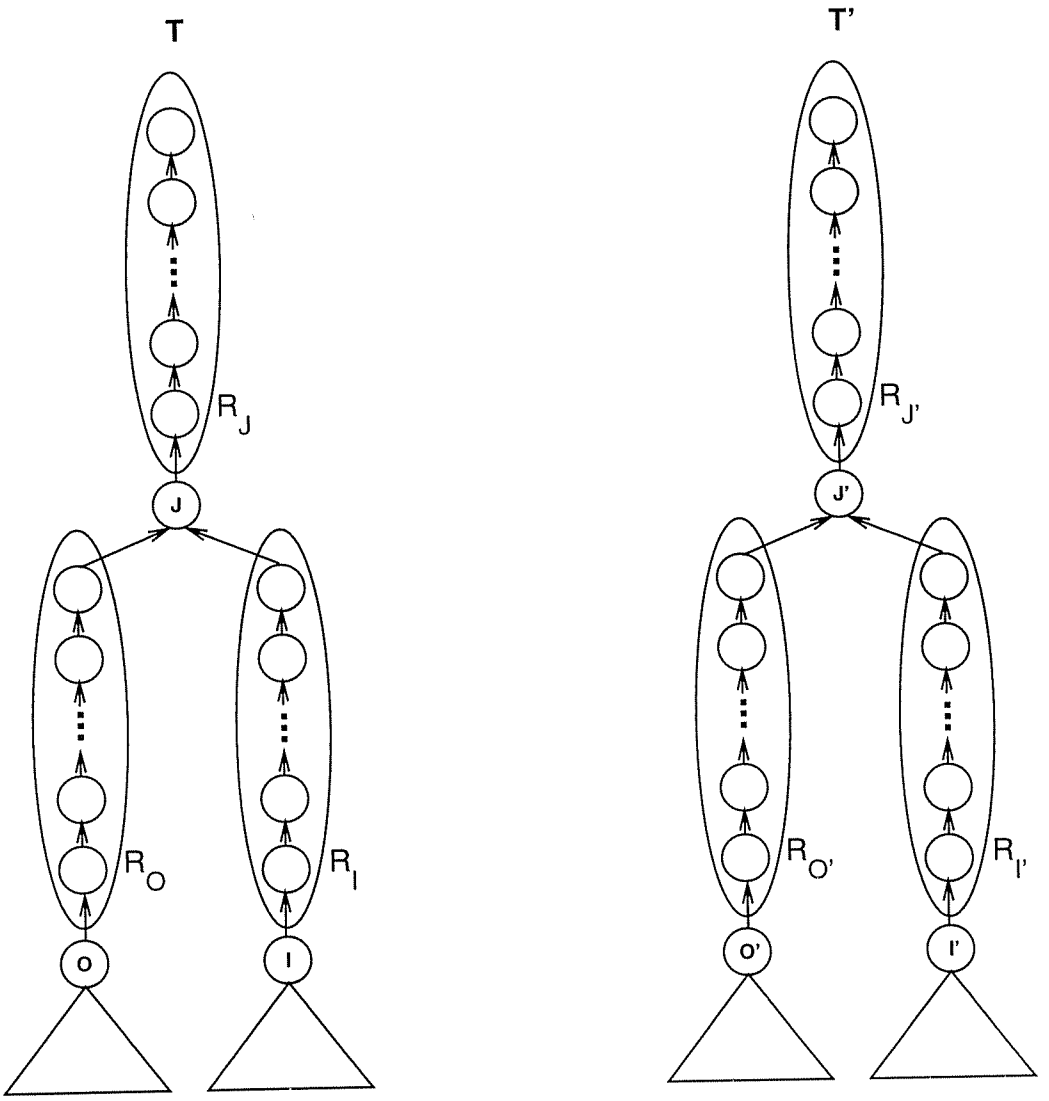

Figure 3: Two semantically correct, join-order equivalent plan trees with well-ordered streams.

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, it must be that $R_O \cup R_I \cup R_J = R_{O'} \cup R_{I'} \cup R_{J'}$. 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.* $R_J \neq R_{J'}$. Without loss of generality we can also assume that $R_J - R_{J'} \neq \emptyset$. Recalling that both trees are well-ordered, this implies that either

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

In either case, we see that R_J is a proper 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 uniqueness portion of 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'. Consider the lowest predicate p in R_J. Since $R_J \supset R_{J'}$, p cannot be in $R_{J'}$; assume without loss of generality that p is in $R_{O'}$. If we consider the stream in T containing O and J, p must have higher rank than J since the stream is well-ordered and p is mobile – *i.e.*, if p had lower rank than J it would be below J. Further, J must be in a group by itself in this stream, since p is directly above J and of higher rank than J. Now, consider the stream in T' containing O' and J'. In this stream, J' can have *rank* no greater than the rank of J, since J is in a group by itself and Lemma 3 tells us that adding nodes to a group can only lower the group's *rank*. Since p has higher *rank* than J, and the *rank* of J' is no higher than that of J, p must have higher rank than J'. This contradicts our earlier 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

Query Plan	Optimization Time		Execution time	
	CPU	Elapsed	CPU	Elapsed
Without Pred. Mig.	0.10 sec	0.10 sec	2 min 24.49 sec	3 min 33.97 sec
With Pred. Mig.	0.30 sec	0.30 sec	0 min 0.04 sec	0 min 0.10 sec

Table 3: Performance of plans for Example 2.

well-ordered streams.

It is easy to see that a minimum-cost tree is well-ordered, and hence that some well-ordered tree has minimum cost. Assume the contrary, *i.e.* there is a semantically correct, join-order equivalent tree T of minimum 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 minimum cost.

Since T_2 is the *only* semantically correct tree of well-ordered streams that is join-order equivalent to T_1, it follows that T_2 is of minimum cost. This completes the proof. ■

2.4 Example 2 Revisited

Theorems 1 and 2 demonstrate that the Predicate Migration Algorithm produces our desired minimum-cost interleaving of predicates. As a simple illustration of the efficacy of Predicate Migration, we go back to Example 2 from the introduction. Figure 4 illustrates plans generated for this query by Illustra running both with and without Predicate Migration. The performance measurements for the two plans appear in Table 3. It is clear from this example that failure to pull expensive selections above joins can cause performance degradation factors of orders of magnitude. A more detailed study of placing selections among joins appears in the next chapter.

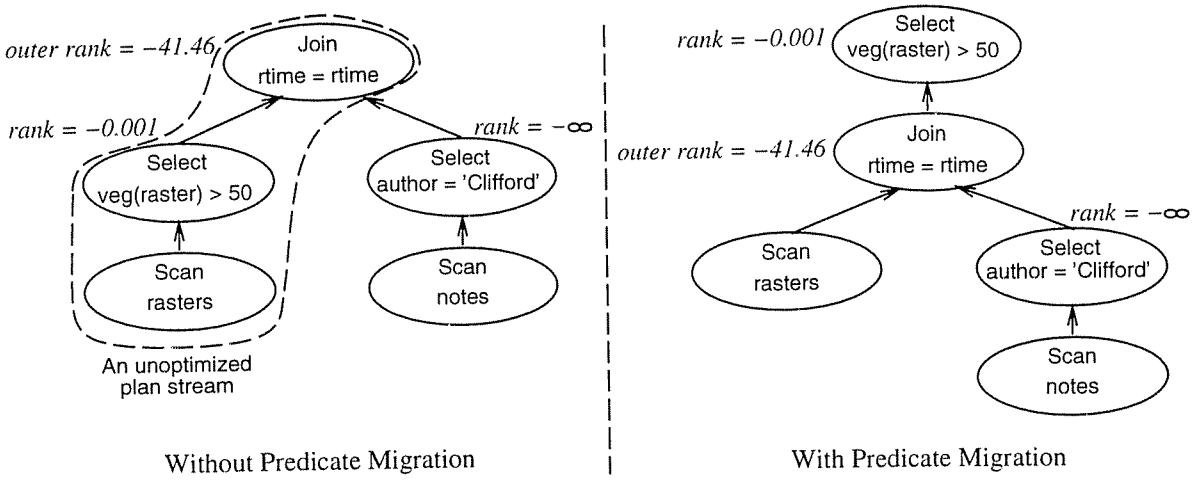

Figure 4: Plans for Example 2, with and without Predicate Migration.

2.5 Preserving Opportunities for Pruning

In the previous section we presented the Predicate Migration Algorithm, an algorithm for optimally placing selection 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 minimum-cost plan for the query.

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. When all plans for some subexpression of a query are generated, most of the plans are pruned out because they are of suboptimal cost. Unfortunately, this pruning does not integrate well with Predicate Migration.

To illustrate the problem, we consider an example. We have a query that joins three relations, A, B, C, and performs an expensive selection 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 subexpression. Assume that because selection predicate p has extremely high *rank*, it will always be pulled above all joins in any plan for this query. Then the join method that 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 plan for the subexpression $B \bowtie C$, not $B \bowtie \sigma_p(C)$. In general, the problem is that subexpressions 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: particularly, a subexpression may have its plans pruned if they will not be changed by Predicate Migration. For example, pruning can take place for subexpressions in which there are no expensive predicates. The following lemma helps to isolate more situations in which pruning may take place:

Lemma 5 *For a selection or secondary join predicate R in a subexpression, if the rank of R is greater than the rank of any join in any plan for the subexpression, then in the optimal complete tree R will appear above the highest join in a subtree for the subexpression.*

Proof. Recall that $rank(p_1) > rank(\overline{p_1 p_2})$. Thus in any full plan tree T containing a subtree T_0 for the subexpression, the highest-*rank* group containing nodes from T_0 will be of *rank* less than or equal to the *rank* of the highest-*rank* join node in T_0. A selection or secondary join predicate of higher *rank* than the highest-*rank* join node of T_0 is therefore certain to be placed above T_0 in T. ■

This lemma can be used to allow some predicate pullup to happen during join enumeration. If all the expensive predicates in a subexpression have higher *rank* than any join in any subtree for the subexpression, then the expensive predicates may be pulled to the top of the subtrees, and the subexpression without the expensive predicates may be pruned as usual. As an example, we return to our subexpression above containing the join of B and C, and the expensive selection $\sigma_p(C)$. Since we assumed that σ_p has higher *rank* than any join method for B and C, we can prune all subtrees for

$B \bowtie C$ (and $C \bowtie B$) except the one of minimal cost — we know that σ_p will reside above any of these subtrees in an optimal plan tree for the full query, and hence the best subplan for joining B and C is all that needs to be saved.[6]

Techniques of this sort, based on the observation of Lemma 5, will be used in Section 3.3.4 to allow Predicate Migration to be efficiently integrated into a System R-style optimizer. As an additional optimization, note that the choice of an optimal join algorithm is sometimes independent of the sizes of the inputs, and hence of the placement of selections. For example, if both of the inputs to a join are sorted on the join attributes, one may conclude that merge join will be a minimal-cost algorithm, regardless of the sizes of the inputs. This is not implemented in Illustra, but such cardinality-independent heuristics can be used to allow pruning to happen even when all selections cannot be pulled out of a subtree during join enumeration.

[6] Of course one may also choose to save particular subtrees for other reasons, such as "interesting orders" [SAC+79].

Chapter 3

Optimization: Practice

In the previous chapter we demonstrated that Predicate Migration produces provably optimal plans, under the assumptions of a theoretical cost model. In this chapter we consider bringing the theory into practice, by addressing a few important questions:

1. Are the assumptions underlying the theory correct?
2. Are there simple heuristics that work as well as Predicate Migration in general? In constrained situations?
3. How can Predicate Migration be efficiently integrated with a standard optimizer? Does it require significant modification to the existing optimization code?

The goal of this chapter is to guide query optimizer developers in choosing a practical optimization solution for queries with expensive predicates; in particular, one whose implementation and performance complexity is suited to their application domain. As a reference point, we describe our experience implementing Predicate Migration algorithm and three simpler heuristics in Illustra. We compare the performance of the four approaches on different classes of queries, attempting to highlight the simplest solution that works for each class.

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 when selection costs dominate. May be OK for MMDBMSs.
PullRank	queries with at most one join	2000	Also used as a preprocessor for Predicate Migration.
Predicate Migration	all queries	3000	Minor estimation problems. Can cause enlargement of System R 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	1100	Prohibitive computational complexity.

Table 4: Summary of algorithms.

Table 4 provides a quick reference to the algorithms, their applicability and limitations. When appropriate, the 'C Lines' field gives a rough estimate of the total number of lines of C code (with comments) needed in Illustra's System R-style optimizer to support each algorithm. Note that much of the code is shared across algorithms: for PullUp, PullRank and Predicate Migration, the code for each entry forms a superset of the code of the preceding entries.

3.1 Background: Analyzing Optimizer Effectiveness

This chapter analyzes the effectiveness of a variety of strategies for predicate placement. Predicate Migration produces optimal plans in theory, but we want to compare it with a variety of alternative strategies that — though not theoretically optimal — are easier to implement, and seem to be sensible optimization heuristics. Developing a methodology to carry out such a comparison is particularly tricky for query optimizers. In this section we discuss the choices for analyzing optimizer effectiveness in practice, and describe the motivation for our chosen evaluation approach.

3.1.1 The Difficulty of Optimizer Evaluation

Analyzing the effectiveness of an optimizer is a problematic undertaking. Optimizers choose plans from an enormous search space, and within that search space plans can vary in performance by orders of magnitude. In addition, optimization decisions are based on selectivity and cost estimations that are often erroneous [IC91, IP95]. As a result, even an exhaustive optimizer that compares all plans may not choose the best one, since its cost and selectivity estimates can be inaccurate.[1]

As a result, it is a truism in the database community that a query optimizer is "optimal enough" if it avoids the worst query plans and generally picks good query plans ([KBZ86],[ML86a],[ML86b],[SI92], etc.) What remains open to debate are the definitions of "generally" and "good" in the previous statement. In any situation where an optimizer chooses a suboptimal plan, a database and query can be constructed to make that error look arbitrarily detrimental. Database queries are by definition *ad hoc*, which leaves us with a significant problem: how does one intelligently analyze the practical efficacy of an inherently flawed algorithm over an infinite space of inputs?

3.1.2 Benchmarks

One way to avoid this problem is to restrict the set of inputs, *i.e.* define a plausible benchmark of a few queries. An optimizer can then be said to be effective if it picks optimal plans for the queries in the benchmark.

There are three benchmarking techniques that are commonly used for DBMS optimizer evaluation. The first is to evaluate the optimizer with "micro-benchmarks" or "validations", *i.e.* to test the cost and selectivity estimates and plan choices for basic operations like table scans or two-way joins, without trying to evaluate the optimizer's behavior on large complex queries. This is the technique used in the influential evaluation of the optimizer in the R* distributed DBMS [ML86a, ML86b], and it is very effective for isolating inaccuracies in an optimizer's cost model.

---

[1] In fact, the pioneering designs in query "optimization" were more accurately described by their authors as schemes for "query decomposition" [WY76] and "access path selection" [SAC+79].

A second technique is to develop "macro-benchmarks", which attempt to analyze the output of an optimizer *on average*, for many (possibly complex) queries. Typically this is done with a randomized benchmark generator, which produces both a random database and random queries (*e.g.*, [SG88], [Swa89], [IK90], [SI92], [HS93b], etc.) This approach is somewhat analogous to comparing multiple C++ compilers by compiling a set of randomly generated programs with each compiler, and running the resulting executables on a set of randomly generated inputs. While this technique can provide some indication of the overall effectiveness of different optimization strategies, it is unlikely to provide much insight into the workings of the competing approaches.

A third technique is to define an agreed-upon database and set of queries, such as the Wisconsin benchmark [BDT83], AS^{3}AP [TOB89], or TPC-D [Raa95]. Such *domain-specific* benchmarks [Gra91] are often based on models of real-world workloads. In terms of optimization, such benchmarks typically expose whether or not a system implements solutions to important details exposed by the benchmark, *e.g.* use of indices and reordering of joins in the Wisconsin benchmark, or intelligent rewriting of sub-queries in TPC-D. If an optimizer chooses a particular evaluation strategy then it does well, otherwise it does quite poorly. The evaluation of the optimizer in these benchmarks is binary, in the sense that typically the *relative* performance of the good and bad strategies is not interesting; what is important is that the optimizer choose the "correct access plan" [TOB89].

3.1.3 Analysis vs. Benchmarking for Expensive Methods

An alternative to benchmarking is to run queries that expose the *logic* that makes one optimization strategy work where another fails. This binary outlook is similar to the way in which the Wisconsin and TPC-D benchmarks reflect optimizer effectiveness, but is different in the sense that there is no claim that the workload reflects any typical real-world scenario. Rather than being a "performance study" in any practical sense, this is a form of *empirical algorithm analysis*, providing insight into the algorithms rather than a quantitative comparison. The conclusion of such an analysis is not to identify *which* optimization strategy should be used in practice, but rather to outline *when* and *why* each of

the various schemes succeeds and fails. This avoids the issue of identifying a "typical" workload, and hopefully presents enough information to predict the behavior of each strategy for any such workload.

Extensible database management systems are only now being deployed, so there is little consensus on the definition of a real-world workload containing expensive methods. As a result we decided not to define a benchmark for expensive methods — any such exercise would be premature at this stage of the deployment of extensible DBMS technology. We could have devised micro-benchmarks to test cost and selectivity estimation techniques for expensive predicates. However, the focus of our work was on optimization strategies rather than estimation techniques, so we have left this exercise for future work, as discussed in Section 6.3.

In terms of benchmarks this leaves only one remaining option: a randomized macro-benchmark. Such a benchmark could provide average-case analyses of the different strategies. However, even if we were able to define a random distribution of queries that would reflect "typical" use of expensive methods, a macro-benchmark would not have provided us insight into the reasons why each optimization strategy succeeded or failed. To return to our C++ analogy, it would be similarly difficult to learn much about the competing C++ compilers by comparing the performance of the random executables on random inputs. Since our goal was to further our understanding of the optimization strategies in practice, we chose to do an algorithm analysis rather than a benchmark.

3.1.4 Experiments in This Chapter

This chapter presents an empirical algorithm analysis of Predicate Migration and alternate approaches, to illustrate the scenarios in which each approach works and fails. In order to do this, we picked the simplest queries we could develop that would illustrate the tradeoffs between different choices in predicate placement. As we will see, approaches other than Predicate Migration can fail even on simple queries. This is indicative of the difficulty of predicate placement, and we believe justifies our decision to forgo a larger-scale macro-benchmark.

Table	#Tuples	#8K Pgs
T1	2 980	75
T2	8 730	216
T3	28 640	705
T4	34 390	847
T5	40 150	988
T6	45 900	1 134
T7	65 810	1 618
T8	71 560	1 759
T9	77 310	1 900
T10	97 230	2 389

Table 5: Benchmark database.

In the course of chapter, we will be using the performance of SQL queries run in Illustra to demonstrate the strengths and limitations of the algorithms. The database schema for these queries is based on the randomized benchmark of Hong and Stonebraker [HS93b], with the cardinality distribution scaled up by a factor of 10. All tuples contain 100 bytes of user data. The tables are named with numbers in ascending order of cardinality; this will prove important in the analysis below. 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 5. The entire database, with indices and catalogs, was about 155 megabytes in size.

In the example queries below, we refer to user-defined methods. Numbers in the method names describe the cost of the methods in terms of random (*i.e.* non-sequential) database I/Os. For example, the method costly100 takes as much time per invocation as the I/O time used by a query that touches 100 unclustered tuples in the database. In our experiments, however, the methods did not perform any computation; rather, we counted how many times each method was invoked, multiplied that number by the method's cost, and added the total to the measurement of the running time for the query. This allowed us to measure the performance of queries with very expensive methods in a reasonable amount of time; otherwise, comparisons of good plans to suboptimal plans would have been prohibitively

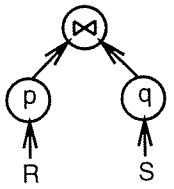

Figure 5: A query plan with expensive selections p and q.

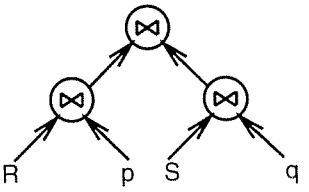

Figure 6: The same query plan, with the selections modeled as joins.

time-consuming.

3.2 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 proposed for extensible relational systems by Yajima, *et al.* [YKY+91]. We refer to this as the LDL approach. The other approach is Predicate Migration, which was first presented in 1992 [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.

3.2.1 The LDL Approach

To illustrate the LDL approach, 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 methods.

Assume that the optimal plan for the query is as pictured in Figure 5, where both p and q are placed directly above the scans of R and S respectively. In the LDL approach, 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 methods. 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 5.* That is because the plan tree of Figure 5, with selections p and q treated as joins, looks like the *bushy* plan tree of Figure 6. 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 approach yet further. No known modification of the IK-KBZ optimizer can handle bushy trees. Yajima *et al.* [YKY+91] successfully integrate the LDL approach with an IK-KBZ optimizer, but they use an exhaustive mechanism that requires time exponential in the number of expensive selections.

3.2.2 Predicate Migration Revisited

The Predicate Migration algorithm presented in Chapter 2 only works when the differential join costs are constants. In this section we examine how that cost model fits the standard join algorithms that are commonly used.

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 3.4.1 we review the accuracy of these estimates in practice.

The cost of a selection predicate is its differential cost, as stored in the system metadata. A constant differential cost of a join predicate per input also needs to be computed, and this is only possible if one assumes that the cost model is well-behaved: in particular, 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. In terms of I/O, the costs of merge and hash joins given by Shapiro, *et al.* [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]

---

[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 many systems including Illustra. 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.

Therefore we can accurately model the differential costs of all the typical join methods per tuple of an input.

These "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].

3.3 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.

3.3.1 PushDown with Rank-Ordering

In our version of the traditional selection pushdown algorithm, we add code to order selections. This 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 is similarly estimated, and selections over a given relation are ordered in ascending order of *rank*. As we saw in Chapter 2, such ordering is optimal for selections, and intuitively it 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.[4]

Thus a crucial first step in optimizing queries with expensive selections is to order selections by

[4]This intuition applies when $0 \leq$ *selectivity* ≤ 1. Though the intuition for *selectivity* > 0 is different, *rank*-ordering is optimal regardless of selectivity.

Query 1:

```
SELECT   T3.a1
  FROM   T3, T2
 WHERE   T2.a1 = T3.ua1
   AND   costly100(T3.ua1) < 0;
```

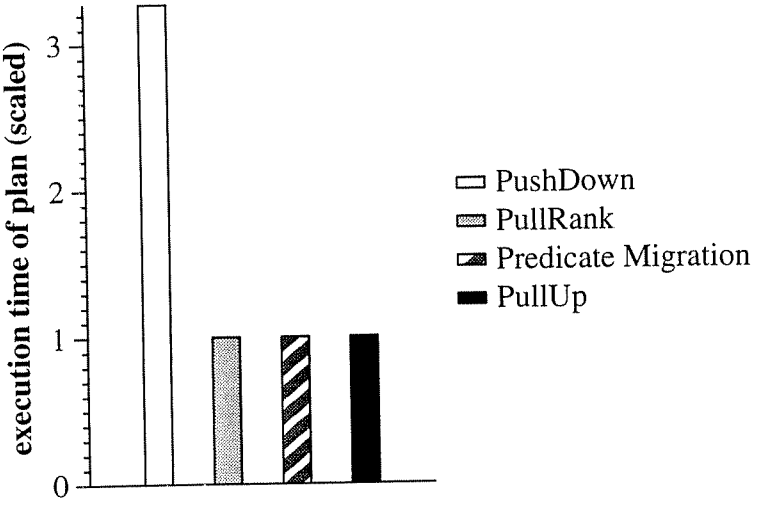

Figure 7: Query execution times for Query 1.

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 Object-Oriented DBMSs (OODBMSs), in which the currently typical ad-hoc query is a collection scan, not a join.[5] For systems supporting joins, however, PushDown may often produce very poor plans, as shown in Figure 7. 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.

3.3.2 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

[5]This may change in the future, since most of the OODBMS vendors plan to support the OQL query language, which includes facilities for joins [Cat94].

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 Management Systems (MMDBMSs), for example, or in disk-based systems that store small amounts of data on which very complex operations are performed. Even in such systems, however, PullUp can produce very poor plans if join selectivities are greater than 1. This problem can be avoided by using method caching, described in Chapter 4.

Query 2 (Figure 8) is the same as Query 1, except T10 is used instead of T2. This minor change causes PullUp to choose a suboptimal plan. Recall that table names reflect the relative cardinality of the tables, so in this case T10.ua1 has more values than T3.ua1, and hence the join of T10 and T3 has selectivity 1 over T3. As a result, pulling up the costly selection does not decrease the number of calls to costly100, and increases the cost of the join of T10 and T3. All the algorithms pick the same join method for Query 2, but PullUp incorrectly places the costly predicate above the join.

Note, however, the unusual graph in Figure 8: all of the bars are about the same height! The

Query 2:

```
SELECT   T3.a1
  FROM   T3, T10
 WHERE   T10.a1 = T3.ua1
   AND   costly100(T3.ua1) < 0;
```

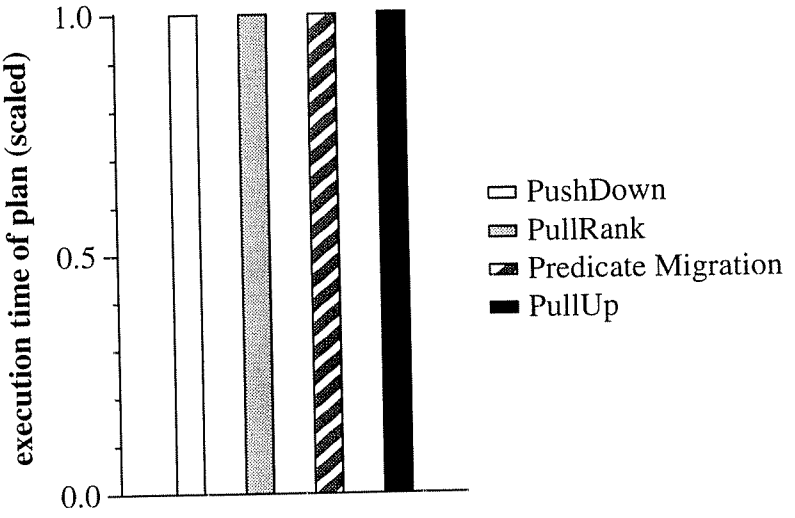

Figure 8: Query execution times for Query 2.

point of this experiment is to highlight the fact that the error made by PullUp is insignificant. This is because the costly100 method requires 100 random I/Os per tuple, while a join typically costs at most a few I/Os per tuple, and usually much less. Thus in this case the extra work performed by the join in Query 2 is insignificant compared to the time taken for computing the costly selection. The lesson here is that *in general, over-eager pullup is less dangerous than under-eager pullup*, since join is usually less expensive than an expensive predicate. As a heuristic, it is safer to overdo a cheap operation than an expensive one.

On the other hand, one would like to make as few over-eager pullup decisions as possible. As we see in Figure 9, over-eager pullup can indeed cause significant performance problems for some queries, especially if the predicates are not very expensive. Thus while it may be a "safer bet" in general to be over-eager in pullup rather than under-eager, neither heuristic is generally effective. In the remaining heuristics, we attempt to find a happy medium between PushDown and PullUp.

Query 3:

```
SELECT   T3.a1
  FROM   T3, T10
 WHERE   T10.a1 = T3.ua1
   AND   costly1(T3.ua100) < 0;
```

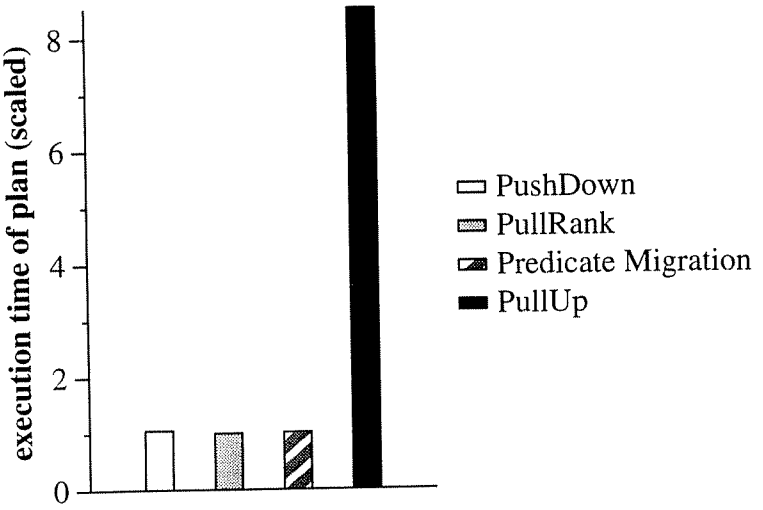

Figure 9: Query execution times for Query 3.

3.3.3 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 stream, generating an inner-*rank* and outer-*rank* for the join. Any selections in the inner stream that are of higher *rank* than the join's inner-*rank* are pulled above the join. Similarly, any selections in the outer stream that are of higher *rank* than the join's outer *rank* are pulled up. As indicated in Lemma 5, PullRank never pulls a selection above a join unless the selection is pulled above the join in the optimal plan tree.

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 3.2.2.

Query 4:

```
SELECT   T2.a100
  FROM   T2, T1, T3
 WHERE   T3.ua1 = T1.a1
   AND   T2.ua100 = T3.a1
   AND   costly100(T2.a100) < 10;
```

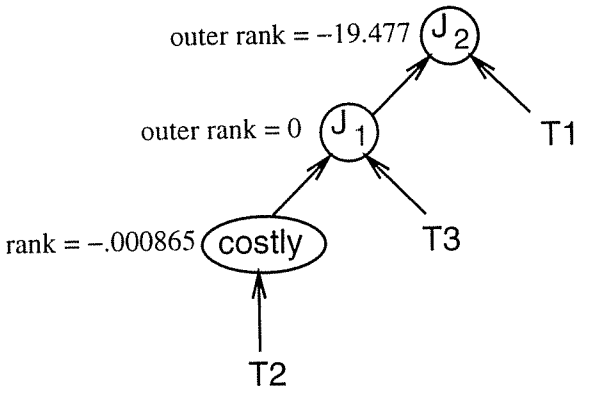

Figure 10: A three-way join plan for Query 4.

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 10. 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 appropriate to pull the selection above the pair $\overline{J_1 J_2}$. PullRank does not consider such multi-join pullups. In general, *if nodes are constrained to be in decreasing order of rank while ascending a stream in a plan tree, then it may be necessary to consider pulling up above groups of nodes,* rather than one join node at a time. PullRank fails in such scenarios.

As a corollary to Lemma 5, it is easy to show that PullRank is an optimal algorithm for queries with only one join and selections on a single stream. PullRank can be made optimal for all single-join queries as well: given a join of two relations R and S, while optimizing the stream containing R PullRank can treat any selections from S that are above the join as primary join predicates; this allows PullRank to view the join and the selections from S as a group. A similar technique can be used while optimizing the stream containing S. Unfortunately, PullRank does indeed fail in many multi-join scenarios, as

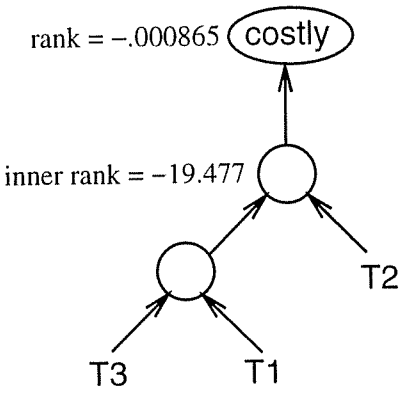

Figure 11: Another three-way join plan for Query 4.

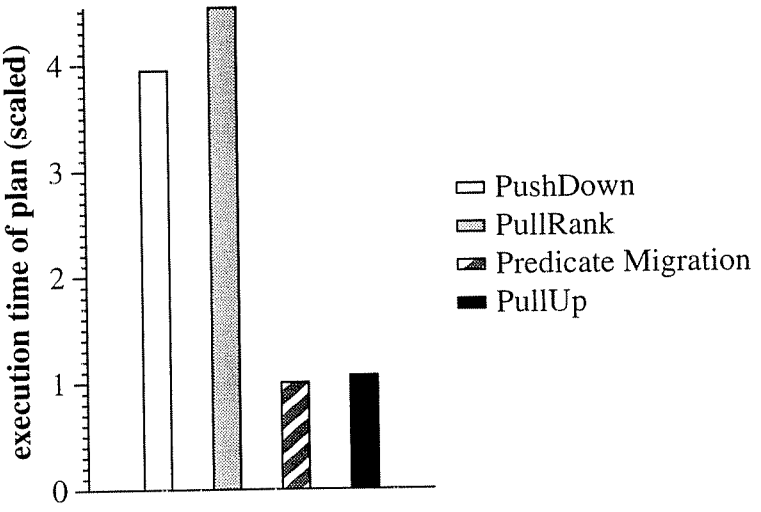

Figure 12: Query execution times for Query 4.

illustrated in Figure 12, since there is no way for it to form a group of joins. Since PullRank cannot pull up the selection in the plan of Figure 10, it chooses a different join order in which the expensive selection can be pulled to the top (Figure 11). This join order chosen by PullRank is not a good one, however, and results in the poor performance shown in Figure 12. The best plan used the join order of Figure 10, but with the costly selection pulled to the top.

3.3.4 Predicate Migration

The details of the Predicate Migration algorithm are presented in Chapter 2. 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 10). 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:

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

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 plan for a subexpression, we mark that subexpression as *unpruneable*. Subsequently when constructing plans for larger subexpressions, we mark a subexpression unpruneable if it contains an unpruneable subplan within it. The System R algorithm is then modified to save not only those subplans that are min-cost or "interestingly ordered"; it also saves all plans for unpruneable subexpressions. 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.

Query 5:

```
SELECT   T2.a100
  FROM   T2, T1, T3, T4
 WHERE   T3.ua1 = T1.a1
   AND   T2.ua20 = T3.a1
   AND   costly100(T2.ua100, T4.a1) = 0
   AND   costly100(T2.ua20) < 10;
```

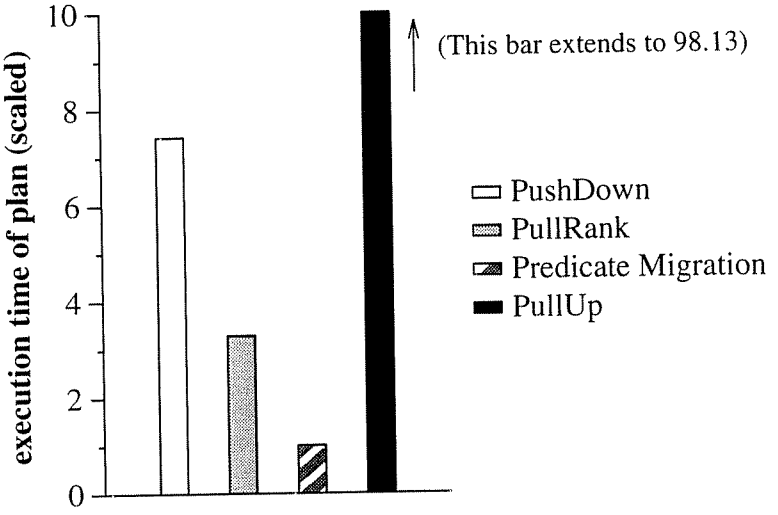

Figure 13: Query execution times for Query 5.

Note that Predicate Migration requires minimal modification to an existing System R optimizer: PullRank must be invoked for each subplan, pruning must be modified to save unpruneable plans, and before choosing the final plan Predicate Migration must be run on each remaining plan. This minimal intrusion into the existing optimizer is extremely important in practice. Many commercial implementors are wary of modifying their optimizers, because their experience is that queries that used to work well before the modification may run differently, poorly, or not at all afterwards. This can be very upsetting to customers [Loh95, Gra95]. Predicate Migration has no effect on the way that the optimizer handles queries without expensive predicates. In this sense it is entirely safe to implement Predicate Migration in systems that newly support expensive predicates; no old plans will be changed as a result of the implementation.

A drawback of Predicate Migration is the need to consider unpruneable plans. In the worst case, every subexpression has a plan in which an expensive predicate does not get pulled up, and hence every subexpression 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 untoward difficulty in practice. The payoff of this investment in optimization time is apparent in Figure 13.[6] Note that Predicate Migration is the only algorithm to correctly optimize each of Queries 1-5.

3.4 Theory to Practice: Implementation Issues

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

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

[6] In this particular query the plan chosen by PullUp took almost 100 times as long as the optimal plan, although for purposes of illustration the bar for the plan is truncated in the graph. This poor performance happened because PullUp pulled the costly selection on T2 above the costly join predicate. The result of this was that the costly join predicate had to be evaluated on all tuples in the Cartesian product of T4 and the subtree containing T2, T1, and T3. This extremely bad plan required significant effort in caching at execution time (as will be discussed in Chapter 4) to avoid calling the costly selection multiple times on the same input.

Illustra 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 Illustra actually proved more time-consuming than writing it initially for POSTGRES. Since Illustra 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 Illustra version of Predicate Migration, about one month was spent upgrading the optimization code for Illustra. 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. 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 minimum-cost plan. In the course of running the comparisons for this paper, a variety of subtle optimizer bugs were found, mostly in cost and selectivity estimation. The most difficult to uncover was that the original "global" cost model for Predicate Migration, presented in [HS93a], was inaccurate in practice. Typically, bugs were exposed by running the same query under the various different optimization heuristics, and comparing the estimated costs and actual running times of the resulting plans. When Predicate Migration, a supposedly superior optimization algorithm, produced a higher-cost or more time-consuming query plan than a simple heuristic, it usually meant that there was a bug in the optimizer.

The lesson to be learned here is that comparative benchmarking is absolutely crucial to thoroughly

debugging a query optimizer and validating its cost model. It has been noted fairly recently 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 [Raa95] — 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, producing minimal-cost plans that were generally as fast or faster in practice than those produced by the simpler heuristics.

3.4.1 Influence of Performance Results on Estimates

The results of our performance experiments influenced the way that we estimate selectivities in Illustra. In Section 3.2.2 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 that 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 Illustra, we calculate $|R|$ on the fly as needed, based on the number of selections over R at the time that 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. In part, this heuristic is based on the existence of method caching in Illustra, as described in Chapter 4 — as long as methods are cached, pulling a selection above a join incorrectly cannot increase the number of times the selection method is actually computed, it can only increase the number of duplicate input values to the selection.

The potential for over-eager pullup in Predicate Migration is similar, though not as flagrant, as the over-eager pullup in the LDL approach. Observe that if the Predicate Migration approach pulls up

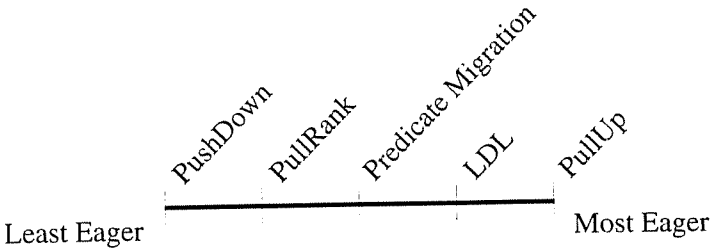

Figure 14: Eagerness of pullup in algorithms.

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 Illustra, and note that unlike LDL, Illustra only exhibits this over-eagerness in a particular situation:

1. when there are expensive selections on both inputs to a join, and
2. some of the expensive selections on the outer input should be pulled up, and
3. some of the expensive selections on the inner input should *not* be pulled up, and
4. treating the inner input before the outer results in selectivity estimations that cause over-eager pullup of some of those selections from the inner input.

3.5 Conclusions on Optimization

Some roughness remains in our selectivity estimations. 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 14.

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 Illustra 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 streams 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.

Chapter 4

Execution: Method Caching

It is undesirable to repeat complex computations. In this chapter we consider query evaluation techniques that prevent a method from being computed more than once on the same input.

As an example, consider the following SQL query, which displays a thumbnail sketch of photos used in advertisements:

```
SELECT   thumbnail(product_image)
  FROM   advertisements
 WHERE   product_name = 'Brownie';
```

The product_image field is a reference to a multi-kilobyte image object. The thumbnail method reads in this object, and produces a small version of the image. Reading in the object can be quite time-consuming, and hence thumbnail can be very expensive. Since a given product may have many different advertisement layouts with the same photo, thumbnail may need to be computed many times on the same image object. It would be wasteful to actually invoke thumbnail on each reference to an image; instead, we would like to invoke thumbnail on the first reference processed for every image, and then *cache* the result of thumbnail on that image for subsequent references.

This chapter explores techniques to do this method caching efficiently. We begin by considering algorithms to do caching for a single expensive method. We review two well-known caching algorithms — memoization and sorting – and present a third approach based on hybrid hashing, which proves

to be preferable to the alternatives in many cases. There are tradeoffs between sorting and hybrid hashing, though, which we explore. In later sections we propose techniques to handle queries with multiple method caches. We conclude with a discussion of the challenge of integrating method caching with query optimization.

4.1 Background

4.1.1 To Cache or Not to Cache?

Methods with duplicate-free inputs derive no benefit from caching, and one should not pay for the overhead of caching in such scenarios. In some cases a query optimizer can ascertain that an input is duplicate-free and can avoid caching, *e.g.*, when the input column(s) form a primary key for the input relation. But there can also be scenarios that can go undetected by an optimizer, in which an input happens to have zero or very few duplicates, and the cost of caching outweighs its benefit.

We will see in this chapter that an appropriately chosen caching technique always has a cost that is at most a fraction of an I/O per tuple. This is a small overhead to incur, especially as compared to the risk of recomputing an expensive method that may take the time of many I/Os. We will also see that a poor choice of caching techniques can make caching extremely expensive. As a result, we recommend caching in any situation where one cannot guarantee a duplicate-free input, but we also recommend that the technique used for caching be chosen carefully based on the analyses in this chapter. We will return to this point in Section 4.7.

4.1.2 A Note on Method Semantics over Time

Methods can be categorized in one of four ways, based on whether or not they produce the same answer on the same input at different points in time. We present these categories in descending order of the amount of time a method maintains the same answer on the same input; this corresponds to the duration that a method cache can be used to produce correct answers.

Most straightforward data analysis or manipulation methods are independent of time, and can be cached indefinitely (*infinite* cache life). Other methods, such as methods that refer to a transaction identifier, may only be cached for the duration of a transaction (*per-transaction* cache life). Some methods, such as subqueries, may only be cached for the duration of a single query language statement, since they access data that may change from query to query (*per-statement* cache life). Occasionally a user will define a method that cannot be cached at all, such as a method that checks the time of day, or that generates a random number (*uncacheable*). A query containing such a method is non-deterministic, since the query is not guaranteed to return the same value given different choices of access methods used to access the data — *e.g.*, the order of scanning a table may affect the output of the query. However, such methods can be valuable to users in certain situations.

The techniques of this chapter are applicable to methods with cache lives that are per-statement or longer. However, SQL3 (and Illustra) only distinguish between two of these classes of methods. SQL3's `create function` command allows methods to be declared `variant` or `not variant`, with `variant` as the default. A `variant` function is uncacheable; a function that is `not variant` is assumed to have infinite cache life. SQL3 makes no syntactic provisions for identifying per-statement or per-transaction cache lives [ISO94, Ill94].

4.2 Two Traditional Techniques

This section presents two previously proposed techniques for caching the results of expensive methods. Each of these algorithms (as well as the hybrid hashing algorithm below) can be used for expensive methods that appear anywhere in a query. The algorithms each take a relation as input and produce a relation as output, and therefore should be thought of as a node in a plan tree, much like Join or Sort (see, *e.g.*, Figure 25 of Section 4.5). If the method being cached is a predicate, then after determining the result of the method the algorithm can discard tuples whose result is FALSE. Logically two operations are occuring — caching followed by selection — but for convenience we fold the selection into the cache

operator. For non-predicate methods, the algorithm simply passes the results of the method to the next operator in the query pipeline.

4.2.1 Main-Memory Hashtables: Memoization

A well-known technique for caching the results of computation is to build a hash table in memory. In the programming language and logic programming literature, this technique is often referred to as *memoization* [Mic68]. The algorithm is sketched in Figure 15. For our advertisements example, this hashtable would be keyed on a hash function of product_image values (which might be object identifiers or file names), and would store (product_image,thumbnail) pairs. During query processing, before computing thumbnail for a given product_image, the hashtable would be checked for the presence of that product_image. If a match were found, the resulting thumbnail would be taken from the hashtable. Otherwise, the thumbnail would be computed, and an entry would be made in the hashtable for the new (product_image, thumbnail) pair.

Memoization is easy to implement, and well-suited for workloads with small numbers of values. Note that it can work well even when many tuples are involved, since the processing per tuple is fairly simple. Unfortunately, the technique breaks down when faced with many values, since the main-memory hash table quickly becomes very large, and the operating system is forced to page it to disk. Since hash accesses by definition have low locality, the operating system paging schemes manage the memory very poorly. This will be demonstrated dramatically in Section 4.4 below.

4.2.2 Sorting: The System R Approach

In Section 1.4 it was pointed out that SQL subqueries are a form of expensive method. The authors of the pioneering System R optimization paper [SAC+79] proposed a scheme to avoid redundant computation of correlated subqueries on identical inputs; their idea is directly applicable to expensive methods in general. They noted that

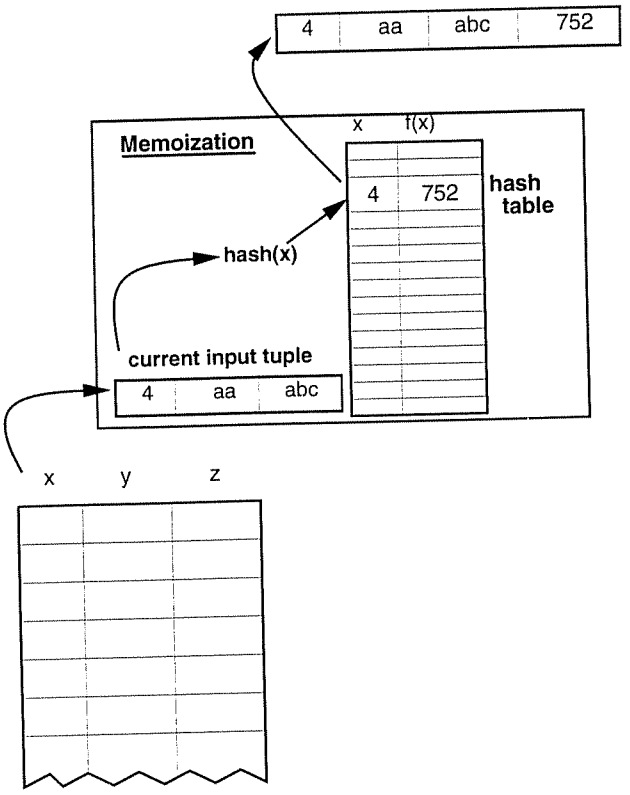

Figure 15: Memoization

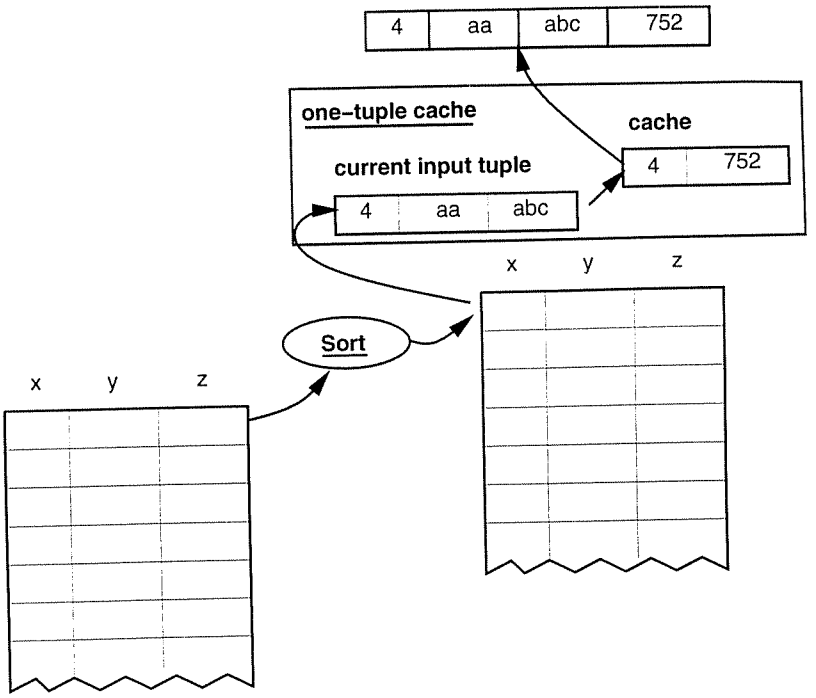

Figure 16: Sorting

> if the referenced relation is ordered on the referenced column, the re-evaluation [of the subquery] can be made conditional, depending on a test of whether or not the current referenced value is the same as the one in the previous candidate tuple. If they are the same, the previous evaluation result can be used again. In some cases, it might even pay to sort the referenced relation on the referenced column in order to avoid re-evaluating subqueries unnecessarily [SAC+79].

In essence, the System R solution is to sort the input relation on the input columns (if it is not already so sorted), and then maintain a cache of *only the last input/output pair* for the method. This prevents redundant computation: after processing tuples with value x_0, if a new value x_1 appears there is no need to maintain the result of the method on x_0, since it is guaranteed that no more x_0s will appear. This is illustrated in Figure 16.

There is a tradeoff between memoization and sorting. If there are few distinct-valued inputs to a method, then the memoization hash table fits in memory, and there is no I/O cost for arbitrary-sized relations. If there are many distinct input values, memoization's I/O cost in paging can be extremely high. By contrast, the I/O cost of sorting a relation is always moderate, regardless of the number of distinct values in the sort columns. Put succinctly, the cost of memoization is based on the number of values, while the cost of sorting is based on the number of tuples. Given an expensive method over a large relation, a good choice between memoization and sorting depends on the number of values in the input to the method: for few values, memoization is preferable, while for many values, sorting is preferable. Since estimating the number of values can be difficult [HNSS95], choosing between memoization and sorting can be tricky. The next algorithm we present makes this decision unnecessary, since it extends memoization to avoid paging.

4.3 Hybrid Cache

The third technique we consider is unary hybrid hashing, which has been used in the past to perform grouping for aggregation or duplicate elimination [Bra84]. Unary hybrid hashing is based on the hybrid hash join algorithm [DKO+84]. We introduce some minor modifications to unary hybrid hashing, and since we are applying it to the problem of caching we call our variant *Hybrid Cache*. Our modifications to standard unary hybrid hashing are discussed further in Section 4.3.2.

The basic idea of Hybrid Cache is to do memoization, but to manage the input stream so that when the main-memory hash table reaches a maximum size, it grows no further. This is accomplished by staging tuples with previously unseen input values to disk, and rescanning them later.

The Hybrid Cache algorithm has three phases. In the first or "growing" phase, tuples of the input relation are streamed in from some source (a base relation, a relational expression, etc.) As the tuples stream in, a main-memory hash table is developed and utilized exactly as in memoization. However, when the main memory hash table reaches a maximum size h (to be discussed in the next section), the

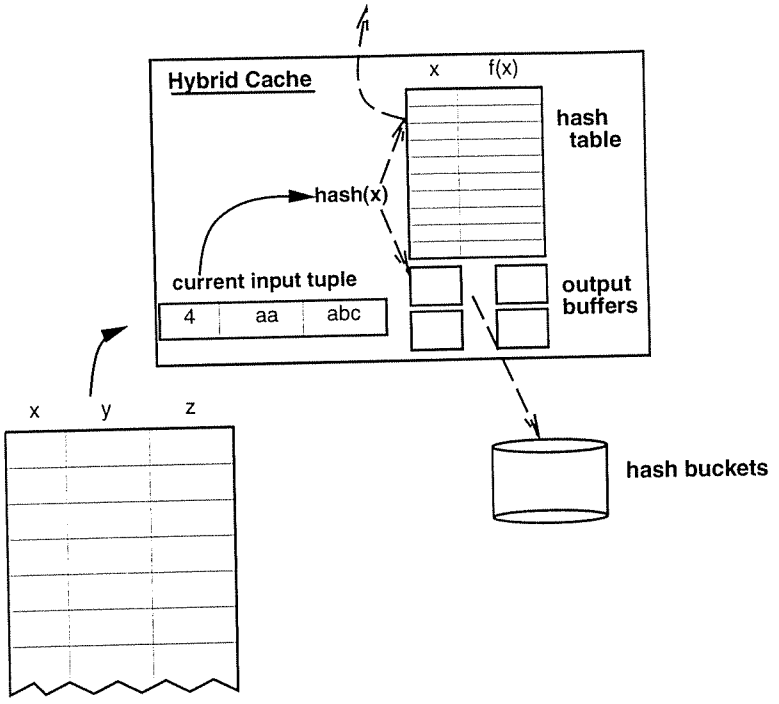

Figure 17: The staging phase of Hybrid Cache.

algorithm enters its second phase, the "staging" phase. In this phase, each tuple of the input relation is considered as it arrives. If a match is found in the hash table, it is used just as in memoization. If no match is found, the tuple is hashed to one of a number of *buckets*, and placed in a buffer for that bucket. The hash function and number of buckets are chosen so that each bucket has approximately the right number of *values* (not tuples!) to fill memory with a memoization hashtable. When a bucket's buffer fills, the buffer is written to the end of a contiguous area on disk allocated for that bucket. The staging phase of Hybrid Cache is illustrated in Figure 17.

When the input relation is consumed, the main-memory hash table is deallocated and the algorithm enters its third or "rescanning" phase. In this phase, the algorithm repeatedly chooses a new bucket, scans in the bucket and handles the function invocations via naive memoization. When the bucket is consumed, the hashtable is deallocated and a new bucket is chosen. This process repeats until no buckets remain on disk. The stage of the algorithm is illustrated in Figure 18.

The Hybrid Cache algorithm is attractive for a number of reasons. First, it works at least as well

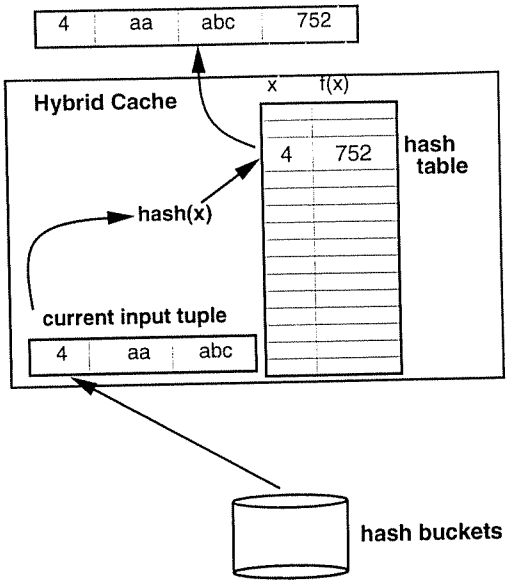

Figure 18: The rescanning stage of Hybrid Cache.

as memoization for inputs of few values: all of the input can be handled in the growing phase, and the last two phases are not required. Second, like sorting it deals gracefully with inputs of many values, without requiring paging. Third, the amount of main memory needed for hashing is proportional to the number of *values* in the input, while the amount of memory needed for sorting is proportional to the number of *tuples* in the input. Thus hashing can handle more input tuples in memory than sorting can, which can make hashing faster than sorting for inputs with many duplicates. This will become more apparent in Section 4.4, in which we present a performance study comparing the three techniques proposed so far.

4.3.1 Memory Allocation During Growing and Staging

A question remains from the above description of Hybrid Cache: during the growing and staging phases, how many pages of memory should be used for the hashtable, and how many for output buffers?

Assume there are M pages of physical memory available to the algorithm. Given the (average) width of the input values and the output of the method, we can compute N, the number of hashtable entries that fit in a page. Ideally, each hash bucket staged to disk should have as many values as make

a hashtable that just fits in memory; *i.e.*, MN values.

We want to know the number of pages h to use for the first in-memory hashtable; the remaining $M - h$ pages will be used as output buffers. If there are v input values total, there should be $(v - hN)/MN$ buckets on disk, requiring $(v - hN)/MN$ pages in memory for output buffers. Now if we want to fill memory while scanning the input, we let $h + (v - hN)/MN = M$. Then by using some simple algebra h can be calculated as a function of v:

$$h = \frac{(M^2N - v)}{N(M - 1)}$$

To ensure that the buffers fit in memory, we require the constraint that $h \geq 0$, which leads us via some additional algebra to the constraint that $v/N \leq M^2$. That is, the number of pages required to place all input values in the hashtable (with their outputs) must be less than the square of the size of memory. This is a well-known constraint on both hashing and sorting, which can be overcome by recursive application of either partitioning (for hashing) or merging (for sorting) [DKO+84, Knu73].

The number of input values v can be estimated by using stored statistics [SAC+79] or via sampling [HOT88, HNSS95]. Unfortunately this estimation is subject to error, so we must consider how the algorithm will behave if estimates are imperfect. If v is estimated too high, h will be too small — *i.e.*, memory will be underutilized during the first two phases of Hybrid Cache. If v is estimated too low, the buckets on disk may contain too many values, and will require recursive partitioning on rescan. But note that if we have upper and lower bounds on the input values (which is typical in most database statistics), v and the hash function can be chosen to ensure that repartitioning will never be required, since the number of values per bucket can easily be capped. Thus the worst behavior in hybrid cache is that it will underutilize memory during the first phase.

4.3.2 Improvements on Unary Hashing

This section points out two optimizations for unary hashing. These optimizations are not specific to caching, and can be applied to any unary hybrid hash algorithm.

The Hybrid Cache algorithm described above already provides one of the optimizations over traditional hybrid hashing techniques. In most hybrid hash algorithms, the first hashtable in memory is populated with all values which hash to bucket 0. If there is skew in the hash function, or if the number of values in the input is incorrectly estimated, bucket 0 may be too large for memory. If this is the case, then the first hashtable will require paging — recursive partitioning cannot be used on the first hashtable. By contrast, in Hybrid Cache we grow the first hashtable so that it exactly fits in memory, while all hash buckets (including bucket 0) are sent to disk. This modification guarantees that the first hashtable in memory will not grow too large.

The second optimization ensures that Hybrid Cache will do as well or better than memoization in all situations. Given the descriptions above of Hybrid Cache and memoization, there is a scenario in which memoization is preferable to Hybrid Cache. The remainder of this section presents an optimization to fix this problem. We did not implement this optimization in Illustra, but we include it here for completeness.

Consider the situation in which the input relation has exactly enough values so that the memoization hashtable fills but does not overflow physical memory. In this case, memoization performs no I/O, and is very efficient. Hybrid Cache is not as efficient in this scenario, since it must use some portion of memory ($M - h$ pages) for output buffers. As a result, Hybrid Cache cannot fit the entire set of input values into its hashtable during the growing phase, and some fraction of the values must be sent to buckets on disk. If there are many tuples that have these values, Hybrid Cache can be much slower than memoization. There is thus a small window where memoization is preferable to Hybrid Cache.

The Hybrid Cache algorithm can be modified to correctly handle the memoization window. In the growing phase we allow Hybrid Caching to fill *all* of physical memory with the hashtable, rather than

just filling h pages. When the hashtable is M pages big and a new input value is seen, we isolate $M - h$ pages of the hashtable. For each input/output pair in this portion of the hashtable, we use the hash function to compute the bucket on disk to which the input corresponds. Having computed the appropriate buckets for all such pairs, we sort the pairs by bucket and then write the pairs to special *cache* regions of their associated buckets on disk. We then deallocate those $M - h$ pages of the hashtable and begin the staging phase, using those pages for output buffers. Later, during the rescanning phase, we initialize the main-memory hashtable for each bucket with the contents of its cache region before beginning to scan tuples from the bucket.

This modification can provide performance benefits even for inputs that do not fall in the memoization window. Consider an input relation that requires staging to disk. During the growing phase, some input value v and its corresponding output is stored in one of the last $M - h$ pages of the hashtable. Until the growing phase completes, any tuple with input value v can be passed to the output without staging. This is in contrast to the unmodified version of Hybrid Cache, which would have staged all tuples of input value v to disk. Note that this optimization comes at a very low execution overhead, namely writing $M - h$ pages of memory to disk. It does requires a somewhat more complicated implementation, however.

4.3.3 Hybrid Cache and Hybrid Hash Join

Hybrid Cache treats its input in much the same way that hybrid hash join treats its "probing" relation. In fact, our implementation of Hybrid Cache in Illustra reuses much of the existing hybrid hash join code for the probing relation. The main-memory hashtable of Hybrid Cache is analogous to the "building" relation in hybrid hash join, but note that Hybrid Cache does not suffer from the "well-known" problems of hybrid hash join:

1. In hybrid hash join, if some join value has many duplicates in the building relation, then all tuples containing this value must be brought into a main-memory hashtable at once, possibly resulting in paging. In Hybrid Cache, even if some input value has many duplicates it only

requires one entry in the main-memory hashtable. The distinction is that hash join requires materializing *tuples* from the building relation in the main-memory hashtables, while Hybrid Cache materializes only *distinct values* in the main-memory hashtables.

2. Hybrid hash join is very sensitive to optimizer estimation errors. In particular, if it underestimates the number of values in the building relation, or chooses a poor hash function, the first bucket of the building relation is likely to be too large and have to be paged, resulting in as much as a random I/O operation (seek and write) per tuple of the first bucket of the probing relation [NKT88]. By contrast, Hybrid Cache never over-utilizes memory, since its first bucket is dynamically grown to the appropriate size. Both algorithms can underutilize memory for the first bucket if estimates are incorrect, but this is less dangerous than over-utilizing memory. The penalty that results from under-utilization is that the first bucket is too small, and too many input tuples are staged to disk and rescanned. Since staging is done in page-sized blocks, the I/O penalty is amortized over a number of tuples, and is therefore less egregious than the penalty paid for over-utilizing memory.

Hybrid hash join can be modified to have the first bucket of its building relation be grown dynamically, as we do for Hybrid Cache. This addresses the second problem above, but does not solve the first problem, since duplicate tuples in the building relation need to be inserted into the hashtable even after memory has filled. More effective (though somewhat complex) solutions to the problems of hybrid hash join have been proposed by Nakayama, *et al.* [NKT88].

4.3.4 Sort vs. Hash Revisited

In analyzing hashing and sorting, Graefe presents the interesting result that hash-based algorithms typically have "dual" sorting algorithms that perform comparably [Gra93]. However, we observe in this section that one of his dualities is based on an assumption that does not apply to the problem of caching. This highlights an advantage of using Hybrid Cache for caching, and also exposes an

important distinction between hashing and sorting that was not noted previously.

Graefe distinguishes between hashing and sorting by observing that hashing techniques utilize an amount of main memory proportional to the cardinality of their *output*, while sorting techniques utilize an amount proportional to the cardinality of their *input*, which is likely to be larger. This is analogous to – but significantly different from – our earlier observation that hashing is *value-based*, while sorting is *tuple-based*. (Note that in the case of grouping, these two analyses are identical, since the cardinality of the output is equal to the number of values in the input). Graefe's analysis of the situation leads him to a modification of sorting via replacement selection [Knu73], which allows sort-based grouping to utilize an amount of memory closer to the cardinality of its output rather than that of its input.

However, note that *the cardinalities of the input and output are identical for caching*: one output tuple is produced for every input tuple.[1] Graefe attempts to make sorting competitive with hashing by getting the sort cost to be proportional to the size of the output. In the case of caching, this does not improve the performance of sort. The operative distinction between sorting and hybrid hashing is not based on input cardinality vs. output cardinality, rather it is that hashing is value-based, while sorting is tuple-based. As a result, hashing is often preferable to sorting for the purpose of caching, as we proceed to demonstrate.

4.4 Performance Study

We implemented memoization and Hybrid Cache in our version of Illustra. Sorting code was already provided in the system to support sort-merge join and SQL's `ORDER BY` clause, so to use sorting for method caching we simply added code for a one-tuple cache over the sort node. We ran experiments using a synthetic table T of 2 million tuples, each having 14 integer-valued fields (56 bytes of user data per tuple). These experiments required Illustra to use more buffer pool space than it is configured

[1] In the case of predicate methods, a tuple is discarded only after the method's result is produced for the tuple. Thus the caching algorithm does indeed produce one output for every input, but subsequently the predicate may discard some of the results as they get propagated.

with by default, so for these experiments we quadrupled Illustra's default MI_BUFFERS setting to 256.

4.4.1 Experiment 1: Effects of Duplicates

Our first set of experiments explore the behavior of memoization, sorting, and Hybrid Cache as the percentage of duplicate values in the input is increased. We consider queries of the form:

```
SELECT   *
  FROM   T
 WHERE   xfalse(T.c);
```

where xfalse is an expensive method that always returns FALSE. We use xfalse so that no tuples satisfy the selection predicate, and hence no time is spent propagating output. We ran this query 7 times per algorithm, and each time varied the column referenced by xfalse. The first column referenced had no duplicate values (*i.e.* 2 million distinct values), the second column had 10 copies of each value (*i.e.* 200,000 values), the third column had 100 copies per value (20,000 values), the fourth column had 1,000 copies per value (2,000 values), the fifth column had 10,000 copies per value (200 values), the sixth column 100,000 copies per value (20 values), and the final column had 1,000,000 copies per value (2 values). The table was sorted randomly so that duplicate copies were interspersed throughout each column. As in Chapter 3, the expensive method did not actually do any expensive computation, so that experimentation could happen at a reasonable pace.

The results of the experiments are shown in Figure 19, which presents the running time of the query minus the time spent in xfalse. Figure 19 demonstrates that memoization is not acceptable when the hashtable does not fit in memory; operating system control over paging results in thrashing. In the case of 1 copy per value, the memoization technique ran for well over half a day, while the other techniques each took under a half an hour.[2] From 100 duplicates onward, memoization and Hybrid Cache behaved identically, since Hybrid Cache did not need to stage any tuples to disk. In subsequent

[2]Actually, the performance result of memoization for the case of 1 copy per tuple is a lower bound on the running time. The first point of the memoization curve in Figure 19 represents the time spent by memoization before it ran out of paging space and crashed. If our machine had been configured with more swap space, it would have run for even longer before completing.

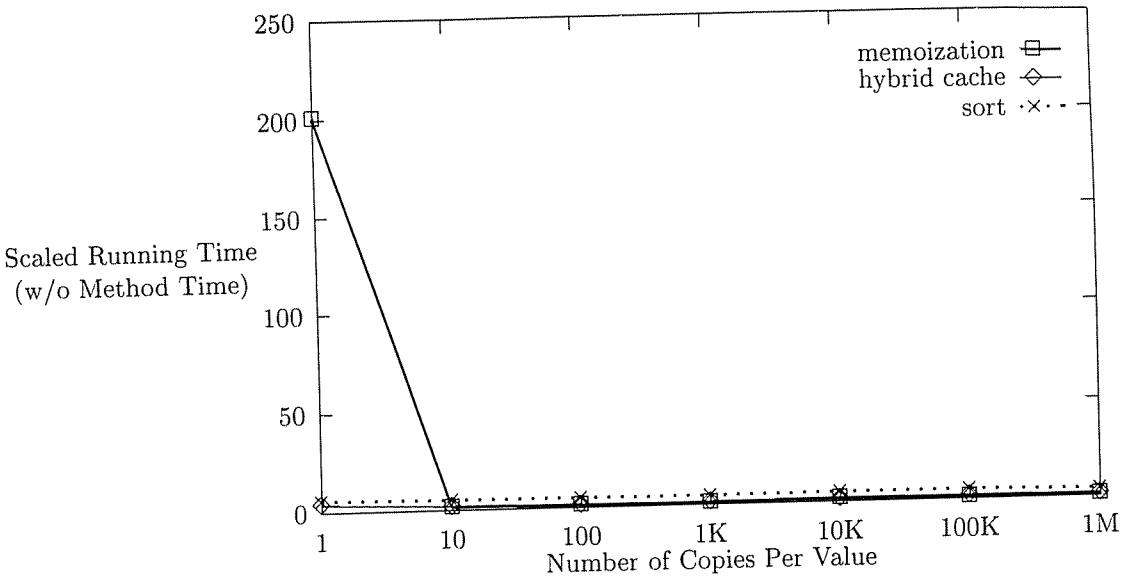

Figure 19: Running time of memoization, Hybrid Cache and sorting.

experiments we ignore memoization, since it is clearly not a general-purpose technique, and can never out-perform Hybrid Cache.

Figure 20 presents the results of the same experiments, ignoring memoization and focussing on the distinction between sorting and Hybrid Cache. We see that the performance of sorting is not competitive with that of Hybrid Cache. The main reason for this is illustrated in Figure 21, which shows the number of temporary-file I/Os that sorting and hashing require to stage and rescan tuples. This depicts what we expect from the preceding discussion: the I/O behavior of sorting is a function of the number of *tuples* in the input, while the I/O performance of Hybrid Cache — a hash-based algorithm — is a function of the number of *values* in the input. The running time of sorting does improve somewhat as the number of duplicates is increased, but this is due to speedups in the sorting of runs in memory. If the in-memory cost of sorting were minimized (as it is for the 1M experiment), sorting would be about as efficient as the worst-case performance of Hybrid Cache (1 copy). As a result, even if sorting performed optimally in memory, Hybrid Cache would be the algorithm of choice for these experiments because of its I/O behavior. Like memoization, Hybrid Cache is optimized for

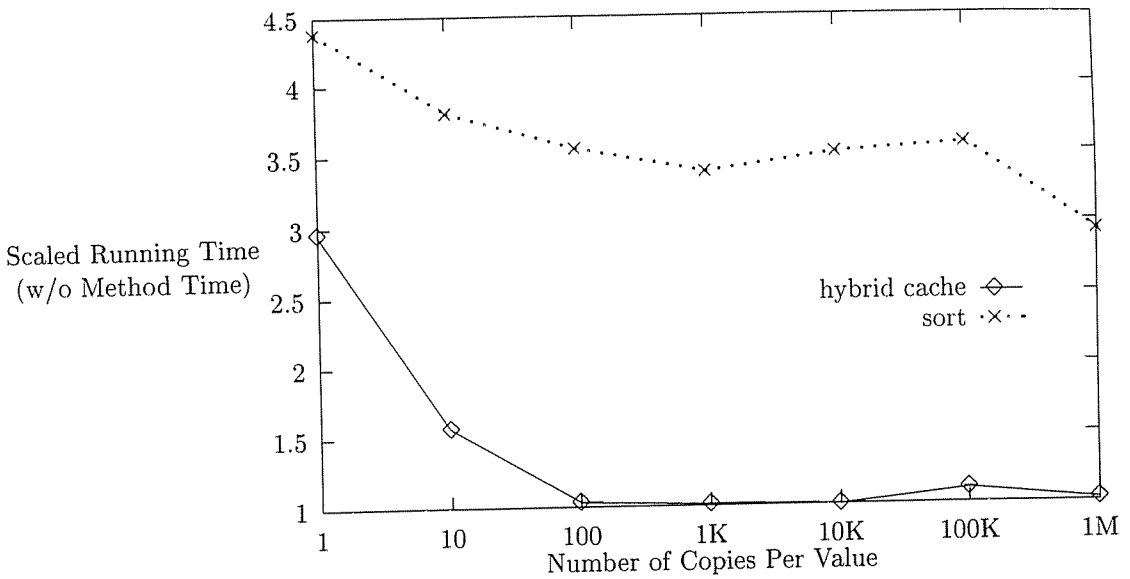

Figure 20: Running time of Hybrid Cache and sorting.

inputs of few values, and like sorting it scales well to inputs with many values.

4.4.2 Effects of Output Size

The previous experiment shows Hybrid Cache out-performing sorting for an expensive predicate method. One important property of predicate methods is that their outputs are of Boolean type, and hence require only a few bits to represent. What if the output of a method is large? In the remaining experiments, we examine how this issue affects the performance of sorting and Hybrid Cache.

Expensive methods can appear in other places than predicates. For example, they can be used in the `SELECT`, `ORDER BY` and `GROUP BY` lists of SQL queries. In such scenarios, the method outputs can be large. A typical example of such a method is decompress, which takes as input a reference to a large object, and produces as output an even larger object. Note that the output of decompress is a value, not a reference. As a result, the output is materialized in memory, and does not have an associated object or tuple identifier.

Our second set of experiments explores the performance of Hybrid Cache and sorting for methods

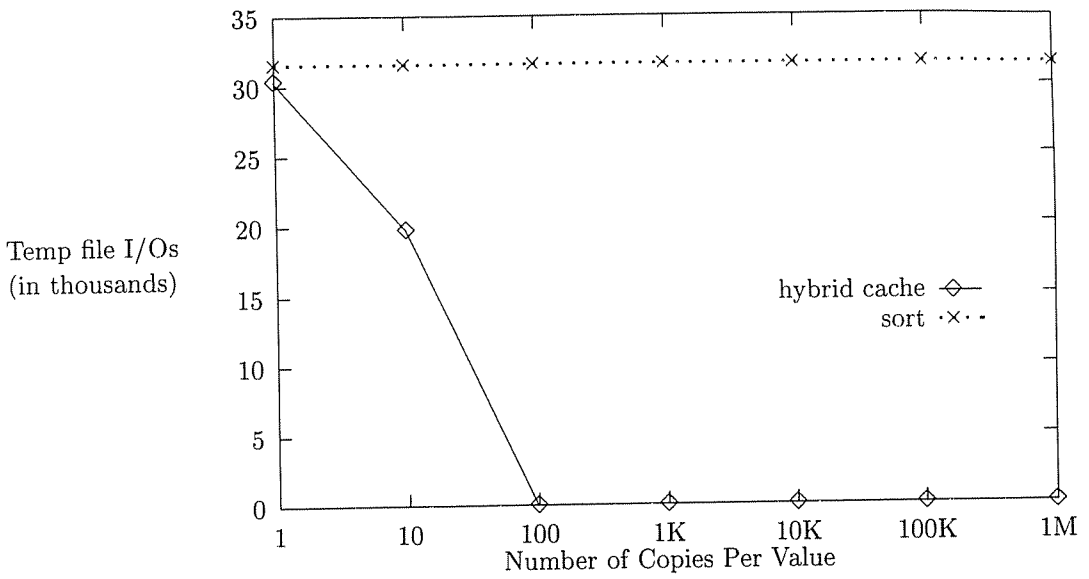

Figure 21: Temp-file I/O for Hybrid Cache and sorting.

producing outputs of various sizes. We consider queries of the form:

```
CREATE  VIEW   V AS
        SELECT  xbig(T.c)
          FROM  T;

        SELECT  xbig
          FROM  V
         WHERE  false(big);
```

where false is an inexpensive method that always returns FALSE, and xbig is an expensive method that returns an object that is 2 kilobytes big. The column T.c is varied among the differing numbers of copies, as in the previous set of experiments. We use false to prevent any tuples from needing to be passed to the output, and we disable Illustra's standard optimization of "flattening" the view into the outer query. As a result, xbig needs to be produced for every tuple of T.

Figure 22 presents the performance of sorting and Hybrid Cache for outputs of size 2 kilobytes. Unlike in our previous experiments, Hybrid Cache is no longer always superior to sorting; in fact, sorting is effective until the number of duplicates becomes quite high. Once again, the performance can largely be explained in terms of the I/O to temporary files, as shown in Figure 23. These graphs

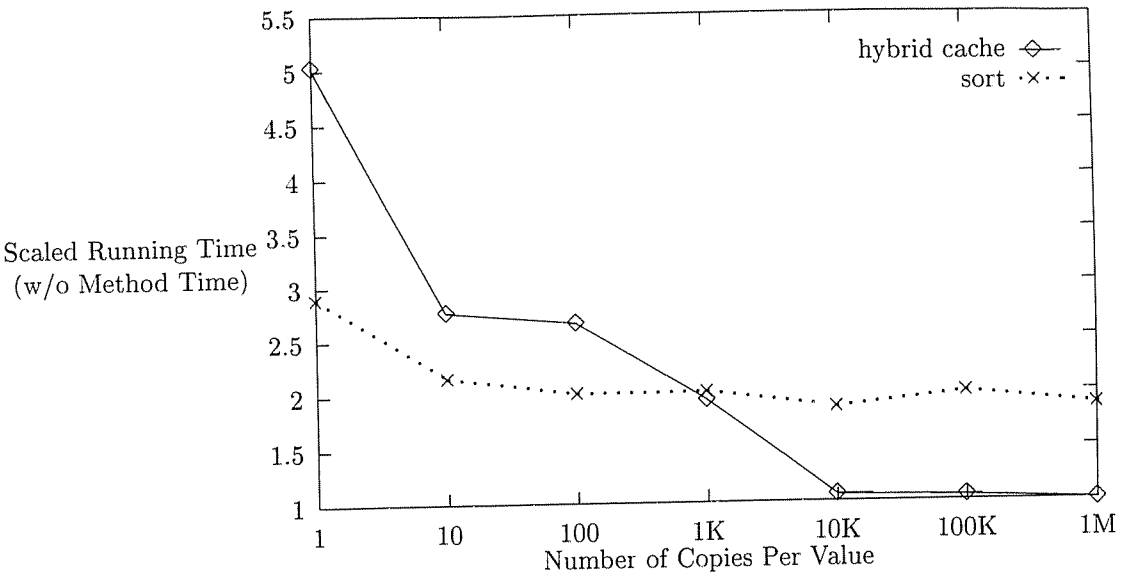

Figure 22: Running time of Hybrid Cache and sorting, 2K outputs.

illustrate a second important distinction between sorting and Hybrid Cache: the I/O behavior of sorting is insensitive to the size of the output, but the I/O behavior of Hybrid Cache degrades when the output size is increased. This is because of the different ways that sorting and Hybrid Cache manage main memory. As illustrated in Figure 16 (page 61), sorting with a one-tuple cache can utilize almost all of main memory for its input tuples, leaving just a single input/output value pair in memory. By contrast, Figure 17 (page 63) illustrates the fact that Hybrid Cache must share main memory among both inputs *and outputs* of the method. As output size is increased, Hybrid Cache can fit fewer and fewer entries into main memory. This means that more and more buckets are required to partition the input. One aspect of this problem is that very few values are placed in the hashtable during the growing phase, and as a result almost all tuples of the input must be staged to disk and subsequently rescanned.

A more significant aspect of this problem is that as output size is increased, the number of values per page (factor N of section 4.3.1) becomes smaller. As a result, the number of pages required for hashing more quickly approaches the limit of the square of the size of memory. When this limit is

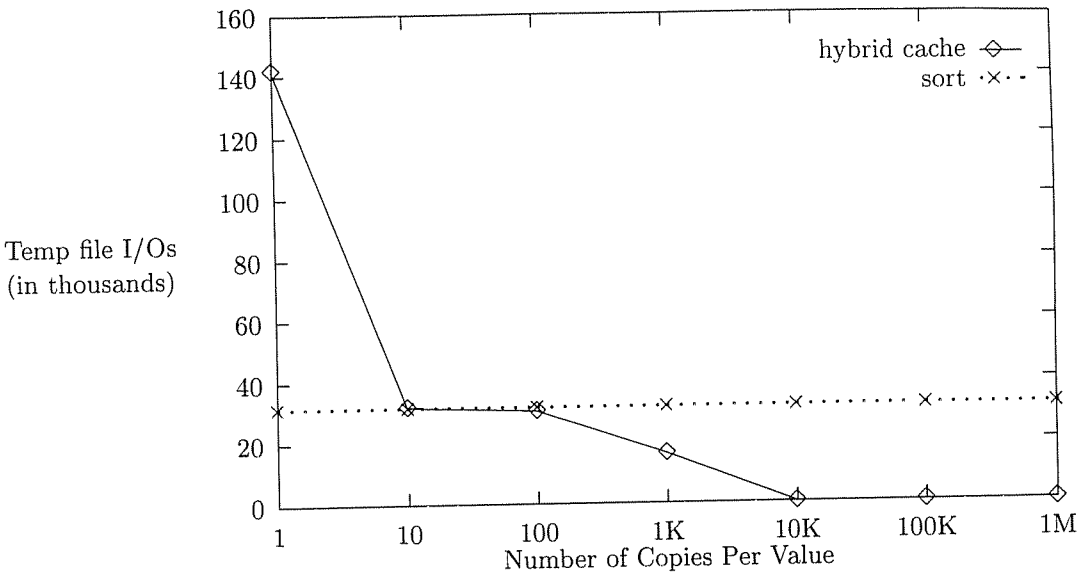

Figure 23: Temp-file I/O for Hybrid Cache and sorting, 2K outputs.

exceeded, recursive partitioning is required to avoid overflowing memory, and this requires extra I/O for recursively staging and rescanning the contents of buckets. This is illustrated in Figure 23, which shows the temporary-file I/Os required to stage and rescan tuples in both Hybrid Cache and sorting. In the case of one copy per value, Hybrid Cache is forced to do recursive partitioning, and must stage and restage tuples to disk multiple times. As the number of duplicates increases, Hybrid Cache no longer needs to do recursive partitioning, and can pass more and more tuples to the output during the growing phase, until at 10,000 copies per tuple every value fits in the hashtable during the growing phase.

The curve for Hybrid Cache in Figure 23 is curious, in that it levels off between 10 and 100 copies. This is explained by examining the number of tuples passed to the output during the growing phase of each query, as shown in Table 6. In the case of 10 copies, the number of tuples passed to the output (and hence never staged) represents just .55% of the input relation, and in the case of 100 copies, the number of tuples passed to the output represents only 5.45% of the input relation. In both cases, the fraction of the input that is not staged to the output is negligible, which explains why the two data

# of copies	1	10	100	1,000	10,000 and up
# of tuples	1,009	10,090	100,900	1,009,000	2,000,000
% of table	0.06%	0.55%	5.45%	54.5%	100%

Table 6: The number of tuples passed to the output during the growing stage of Hybrid Cache, for 2-kilobyte method outputs.

points are approximately at the same height. However, since this percentage grows by a factor of 10 as we move right in Table 6, it quickly becomes larger than one and prevents any input tuples from being staged.

Another curious point is that while Hybrid Cache and sorting do about the same number of temporary-file I/Os for 10 and 100 copies per value, sorting out-performs Hybrid Cache in terms of elapsed time for these inputs. The reason for this is that sorting requires fewer temporary files on disk then Hybrid Cache, since the Hybrid Cache buckets are small due to the large output size of the method. The time difference between Hybrid Cache and sorting is largely due to the cost incurred by Hybrid Cache in maintaining all its temporary files. Illustra allocates temporary files from the operating system, which places a limit on the number of open files allowed per process. As a result, when there are more temporary files than the operating system's open-file limit, Illustra must close and re-open temporary files to keep the number of open files below the limit. The difference in performance between Hybrid Cache and sorting at 10 and 100 copies is due to this overhead of closing and opening temporary files. Hybrid Cache could be improved in this regard by having the DBMS manage its temporary files without using the file system.

4.4.3 Summary of Experimental Results

Our experiments demonstrate that there are two basic factors that affect Hybrid Cache and Sorting: the percentage of distinct input values, and the size of method outputs. Hybrid Cache handles duplicates well and large outputs poorly; sorting handles duplicates poorly and large outputs well.

This tradeoff is illustrated in Figure 24. An input relation is shown, along with a "virtual" column

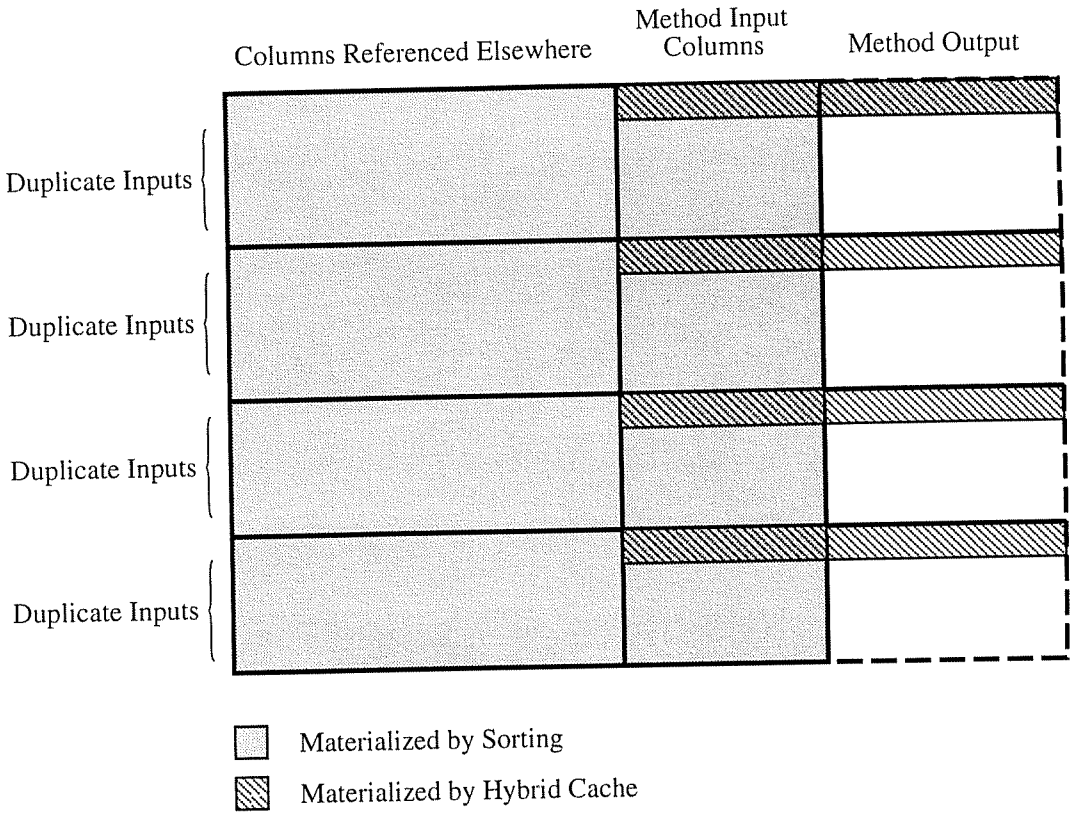

Figure 24: Portions of the input relation materialized in memory.

representing the result of the method computation. (For purposes of illustration, the input is depicted as if it were sorted on the inputs to the method.) The differently shaded areas represent the portions of the input materialized in main memory by sorting and Hybrid Cache. Sorting materializes all of the input tuples and only one row of the virtual output column. Hybrid Cache materializes only the distinct values of columns that are referenced by the method, along with their corresponding output values. The tradeoff is depicted spatially in Figure 24: the performance of the two algorithms varies with the size of the regions that they must materialize in memory. Whichever algorithm uses less memory can pass more tuples to the output without staging, and is less likely to require costly additional I/Os to handle main-memory limitations.

As mentioned above, predicate methods have output values that are very small. As a result, Hybrid Cache is typically a better choice than sorting for expensive predicate methods. For other expensive methods, such as those in `SELECT`, `ORDER BY`, and `GROUP BY` lists, a choice between Hybrid Cache and sorting should be made based on the size of the distinct method outputs as compared to the expected memory needed for tuples with duplicate input values.

4.5 Caching Multiple Methods

The previous sections present a detailed analysis and performance study of caching the results of a single method. A system that uses the appropriate choice of Hybrid Cache and sort for each expensive method is already well-tuned for the sorts of queries possible in today's extensible systems. In this section we propose some additional techniques, which can be used in high-performance systems to further streamline the execution of some queries with multiple expensive methods.

A query can contain multiple expensive methods, either in separate expressions (*e.g.* in two different select list expressions or predicates), or composed in a single expression. The caching techniques in the previous section can be chained together to handle such scenarios. Methods in separate expressions may be handled naturally by a sequence of Hybrid Cache and sort nodes, as illustrated in Figure 25. A

```
SELECT   thumbnail(product_image)
  FROM   advertisements
 WHERE   ad_text similar '.*optic.*';
```

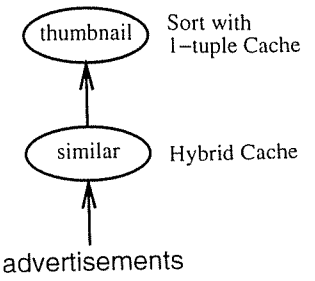

Figure 25: A query with 2 expensive methods.

```
SELECT   name
  FROM   advertisements
 WHERE   percentage(quantize(product_image), 'red') < 50;
```

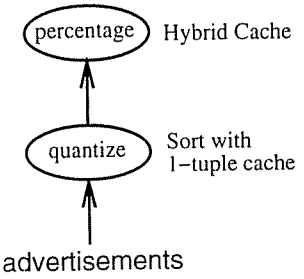

Figure 26: A query with nested expensive methods.

similar technique can be used for nested methods, as illustrated in Figure 26. Note that for predicates with nested methods, only the topmost node of the nesting actually performs the selection. Methods nested lower down simply produce their method outputs for consumption by higher levels.

4.5.1 Sharing One Cache Among Multiple Methods

An important optimization to note is that *two methods with the same input can be computed by the same cache node.* Hybrid Cache and sorting both work by organizing their input tuples based on the input value to the method; methods with identical input values can thus be computed jointly. For example, the query of Figure 26 could be cached with a single Hybrid Cache node storing (product_image, percentage(quantize(product_image), 'red')) pairs.

Additional optimizations are possible when the inputs are similar but not identical. Consider sorting first. Assume we have two methods f and g, where the set of input variables to f is I_f, and the set of input variables to g is I_g, and $I_f \subset I_g$. In this case, both methods may be cached by sorting the input tuples once on I_g, with the sort keys ordered so that elements of I_f precede those of $I_g - I_f$.

For example, if the input to f was {product_image}, and the input to g was {ad_text, product_image}, the sorting should be done with on (product_image, ad_text) pairs. If I_f is not a proper subset of I_g but overlaps I_g significantly, we can still compute both with a single sort. We sort the input relation on $I_f \cup I_g$ with the sort keys ordered to have $I_f \cap I_g$ first. The in-memory cache for sorting will need to hold more than one input/output pair — in the worst case it will have to hold as many pairs as there are distinct values of the input for the same values of $I_f \cap I_g$.

Hybrid Cache can be modified to perform similar optimizations. Given two methods f and g with input variables I_f and I_g respectively such that I_f and I_g have a large intersection, the input can be hashed on $I_f \cap I_g$, but the inputs stored in the main-memory hashtables include all of the variables in $I_f \cup I_g$. This may result in hash collisions at the main-memory hashtable, which can degrade CPU performance if there are many distinct instantiations of I_g that have the same values for I_f. This technique works whether or not I_f is a subset of I_g.

Note that users can force two methods to be cached together by writing a new method which combines the two original methods into one. For nested methods (*e.g.*, $f(g(x))$, this simply entails writing a new method which is equivalent to the composition of the old methods (*e.g.*, $fg(x)$); for multiple methods that are not nested ($f(x), g(y)$), the new method must return a composite object containing the results of both of the methods. This observation can be handy for users of systems which do not implement cache-sharing.

4.5.2 Cache Ordering

When multiple expensive predicate methods appear in a query, the query optimizer orders them appropriately based on their relative costs and selectivities. However, when multiple expensive methods appear in a single predicate, or when multiple expensive methods appear in a `SELECT` list, `ORDER BY` list or `GROUP BY` list, no selection takes place and even advanced optimization schemes such as Predicate Migration will order the methods arbitrarily.

Yet there is often a choice of how to order cache nodes in a query plan. For example, methods from

different SELECT list expressions can be ordered arbitrarily amongst themselves, as can methods that are at the same level of nesting in a nested expression (*e.g.*, g and h in the expression $f(g(x), h(y))$). A poor choice of ordering can be dangerous, even though no selection is involved. Non-predicate cache nodes pass the outputs of their methods to the next operator in the pipeline. These outputs can be quite large, and the next cache node in the pipeline — whether it uses sorting or Hybrid Cache – may need to stage some of these large outputs to disk. It is therefore beneficial to order these cache nodes in increasing width of their output tuples. As an additional optimization, if the output of some cache node is very wide and multiple cache nodes follow it in the output, these outputs should be staged to disk once, referenced via pointers during the rest of the query plan, and rescanned only when they are next required.

4.5.3 Benefits of Semi-Join

A further, somewhat complex optimization is possible when a long chain of method caches appears in a query, and it is not advisable to merge the chain into fewer caches. Recall that the I/O cost of both sorting and Hybrid Cache results from staging tuples to disk and then rescanning them. To minimize this cost, we can first project the input to the caches so that only the columns referenced by the methods are passed into the caches; additionally, duplicates can be removed from this projection. The chain of caches can then be used to compute the methods for the distinct inputs, and the result of this chain can be rejoined with the original input relation to regenerate the projected columns and the correct number of duplicates. This is depicted in Figure 27 for the query of Figure 25.

There are multiple ways to achieve a query plan such as the one of Figure 27. A cost-based optimizer could be extended to produce the plan for the query. Alternatively, the plan can be achieved via rewriting the query as:

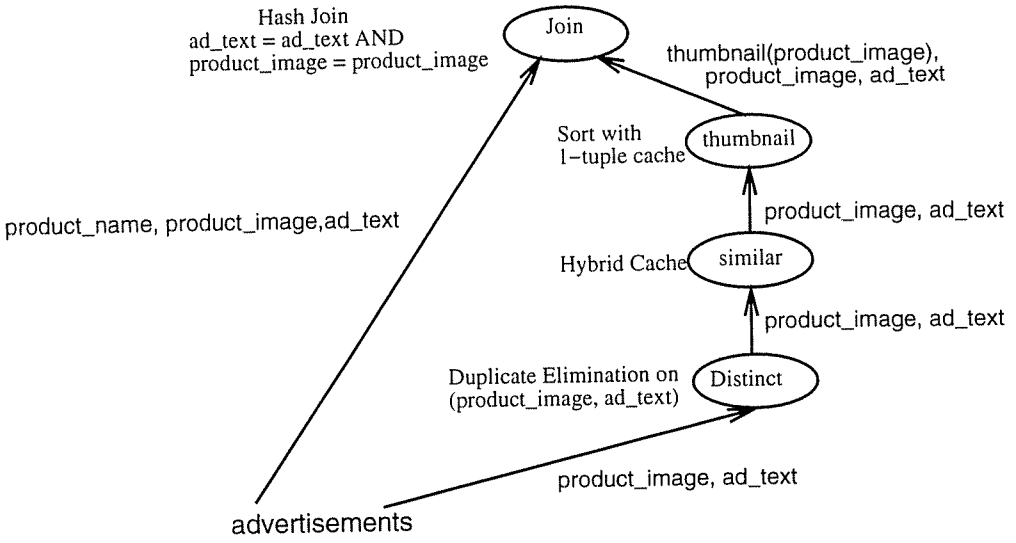

Figure 27: An alternative plan for the query of Figure 25.

```
CREATE  VIEW   vals AS
SELECT DISTINCT  product_image, ad_text
          FROM   advertisements;

CREATE  VIEW   methods AS
       SELECT  thumbnail(product_image), product_image, ad_text
         FROM  vals
        WHERE  ad_text similar '.*optic.*';

       SELECT  methods.thumbnail
         FROM  advertisements,methods
        WHERE  advertisements.ad_text = methods.ad_text
          AND  advertisements.product_image = methods.product_image;
```

Such rewriting can be done by the user, or by a system with a query rewrite engine [PHH92].

Chimenti, *et al.* [CGK89] note that expensive selection predicates can be viewed as joins between the input relation and an infinite logical relation of input/output pairs for the selection method. In this spirit, one can think of our last optimization as forming the *semi-join* [BC81] of the input relation with the (infinite) Cartesian product of the (infinite) logical relations of the methods, and then re-joining the result of the semi-join with the original relation. This is depicted in Figure 28. Semi-joins as used for method caches are quite similar to the semi-joins of distributed database systems, but the motivation

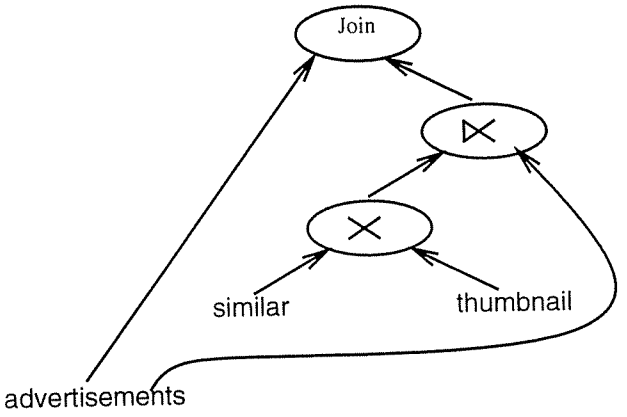

Figure 28: The query plan of Figure 27 depicted as a semi-join.

for using them is different. In distributed systems, semi-joins minimize communication costs. In method caching, this semi-join-like technique minimizes staging costs. In both cases, the benefits of semi-joins need to be balanced against their costs, namely scanning the input relation twice, removing duplicates, and doing an additional join [ML86b]. Note that these costs certainly outweigh the benefit derived by a single cache node, and thus semi-join techniques can only be beneficial for multi-method queries.

4.6 Integration with Query Optimization

By implementing method caching in a query execution engine, one changes the costs of query plans, since methods are only invoked once per value rather than once per tuple. The query optimizer must be modified to reflect this change, so that it picks the least expensive query that the execution engine can run.

The total costs of methods in a query plan with caching can be computed by multiplying the differential cost of the method times the number of distinct values in its input. Estimating the number of values can be difficult in general, though many approaches have been proposed [SAC+79, FM85, HNSS95]. Predicate Migration and related heuristics are typically used to place expensive predicates in a query plan. However, the estimates for Predicate Migration presented in Section 2.1 do not handle

the implications of caching. Modifying them to do so requires two changes, to estimates of cost and selectivity.

First, we need to find a common unit with which to express expense. Given caching, methods have differential costs per *value*, while joins have differential costs per tuple of a stream. To resolve this discrepancy, we convert method costs to be expressed per tuple. This is done by multiplying the differential method cost times the ratio of values to tuples in a stream; this ratio can be computed quite accurately per column (or set of columns) of the base relations via system statistics [SAC+79, FM85]. We assume that this ratio remains constant throughout the query plan, *i.e.* that any predicate of selectivity s reduces the number of values in each column of its input by a factor of $1 - s$.

Second, in order to preserve this assumption we must modify our estimation of join selectivity with respect to one input stream, to ensure that duplicates in the other stream do not affect the values-to-tuples ratio. To do this, we estimate the selectivity of a join based on the number of *values* in the other stream, rather than on the number of tuples. That is, given a join predicate J of selectivity s over $R.c1$ and $S.c2$, the selectivity of J over R is estimated as $s \cdot number\_of\_values(S.c2)$, and the selectivity of J over S is estimated as $s \cdot number\_of\_values(R.c1)$. 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 stream, 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.[3]

Given these two modifications, and the assumption that the values-to-tuples ratio remains constant in a stream, the proofs of Chapter 2 continue to hold for systems with caching. There are certainly situations where the assumption is incorrect, just as there are situations where the assumption of independence of predicates is incorrect. However, given the difficulty of correctly estimating the number of values in a column of a relational expression, and the difficulty of finding an effective technique for predicate placement, we feel that this assumption is workable for today's systems.

---

[3]This bound is actually implicit in the widely used System R technique for join selectivity estimation, which is based on the number of values in the join column.

4.7 Conclusions

This chapter studies the problem of avoiding redundant method invocation during query processing. Three algorithms are considered for caching method results: memoization, sorting, and a minor variation of hybrid hashing which we call Hybrid Cache. Memoization is shown to be ineffective in general. Hybrid Cache is shown to dominate the other algorithms for expensive predicate methods. Non-predicate expensive methods can have arbitrary-sized outputs, and we demonstrate that for methods with large outputs sorting is often preferable to Hybrid Cache. Our variations on hybrid hashing apply not only to caching, but also to other applications of unary hybrid hashing such as grouping for aggregation and duplicate elimination.

We also present a number of more advanced techniques for handling multiple methods in a query, which are designed to improve upon the solution of chaining multiple Hybrid Cache and/or sort nodes together. Implementing these techniques is a second-order issue, since they are applicable only to complex queries with multiple expensive methods. For a high-performance extensible system, however, handling such queries as efficiently as possible can prove to be very important. Finally, we note that caching should be taken into account by a query optimizer, and we show how to integrate caching with Predicate Migration.

In conclusion, we recommend that both sorting and Hybrid Cache be implemented in extensible database systems. It is attractive to simply use sorting for all expensive methods, since code for sorting is typically already present in most systems. We discourage this decision, however, since sorting is out-performed by Hybrid Cache for the common case of expensive predicates whose inputs have few values and many tuples. If methods are always extremely expensive, then the difference between sorting and Hybrid Cache will usually be imperceptible. But for methods which are only moderately expensive, appropriate use of Hybrid Cache will provide noticeable improvements in performance for inputs with many duplicates. For handling multiple methods, it is important and relatively simple to identify that multiple methods on the same inputs can be cached by a single operator. Further

optimizations, especially those involving semi-joins, require significantly more work and are much less generally applicable; as a result, they may only be appropriate for high performance systems. If necessary, database users can achieve these optimizations in more basic systems by rewriting their queries or methods.

Chapter 5

Related Work

5.1 Optimization and Expensive Methods

Stonebraker 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 dissertation, although we vary slightly in the definition of cost metrics for expensive functions.

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 the notion of *rank*. However, their scheme uses *rank* to reorder joins rather than selections. Their techniques do not consider the possibility of expensive selection predicates, and only reorder nodes of a single path in a left-deep query plan tree. Furthermore, their schemes are a proposal for a completely new method for query optimization, not an extension that can be applied to the plans of any query optimizer.

Various papers in the Artificial Intelligence literature [Han77, Nat87, Smi89] use metrics similar to *rank* to determine evaluation order of AND/OR trees in logic programming systems. These papers are applications of the early Operations Research work of W. E. Smith [Smi56], and do not address the problems that arise with joins, including the issues of constraints on ordering and integrating the orderings of multiple streams.

The notion of expensive selections was considered in the context of the LDL logic programming system [CGK89]. Their solution was to model a selection on relation R as a join between R and a virtual relation of infinite cardinality containing the entire logical predicate of the selection. By modeling selections 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 selections as joins can make query optimization prohibitively expensive for a large set of queries, including queries on a single relation. Predicate Migration and the other heuristics we considered do not cause traditional optimizers to exhibit this exponential growth in optimization time.

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 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. Another semantic rewriting technique called "Predicate Move-Around" [LMS94] suggests rewriting SQL queries by copying predicates and then moving the copies across query blocks. The authors conjecture that this technique may be beneficial for queries with expensive predicates. All of these ideas are similar to the query rewrite facility of Starburst [PHH92], and are heuristics rather than cost-based optimizations. It is important to note

that these issues are indeed orthogonal to our problems of predicate placement — once queries have been rewritten into cheaper forms, they still need to have their predicates optimally placed into a query plan.

Kemper, Steinbrunn, *et al.* [KMPS94, SPMK95] address the problem of planning disjunctive queries with expensive predicates; their work is not easily integrated with a traditional (conjunct-based) optimizer. Like most System R-based optimizers, Illustra focusses on Boolean factors (conjuncts). Within Boolean factors, the operands of OR are ordered by the metric *cost/selectivity*. This can easily be shown to be the optimal ordering for a disjunction of expensive selections; the analysis is similar to the proof of Lemma 1 of Section 2.2.1.

The optimization work in this dissertation has been previously published in earlier forms [Hel92, HS93a, Hel94].

5.2 Method and Subquery Caching

Chapter 4 focusses on techniques for caching method results while computing a single query language statement. Alternate techniques can be used for caching methods in a persistent manner to be reused across multiple queries over a period of time (for methods with per-transaction or infinite cache lives). Graefe provides an annotated bibliography of these ideas in Section 12.1 of his query processing survey [Gra93]. These persistent caches are akin to materialized views or method indices — from the point of view of a single query, they represent (partial or complete) *precomputation* of methods, rather than caches. Policies for generating persistent caches are thus more closely related to problems of database design than to query processing. Query processing over persistent caches is relatively simple: one performs a join of the input relation and the persistent cache by using a standard join algorithm. Similar techniques can be used for avoiding recomputation of common relational subexpressions [Sel88]. If the cache is incomplete, an outer join [Dat83] may be used — method inputs that are not found in the cache are "preserved" by the outer join, and for these inputs the method must be computed later in

the query. This latter computation can directly benefit from the query processing techniques described in Chapter 4, as can algorithms for initially generating persistent caches.

Chapter 6

Conclusion

This dissertation presents techniques for efficiently handling queries that utilize a core feature of extensible database systems: user-defined methods and types. Efficiently evaluating queries with expensive methods required further research into query optimization and execution, which led to the theoretical contributions of Predicate Migration, and to new insights on hashing and sorting for method caching. In addition to developing new techniques for handling these problems, we were able to fairly easily integrate our techniques into an industrial-strength solution for Illustra. This illustrates the viability of the work, both in terms of its practical performance and in terms of the simplicity of implementation. Although this dissertation represents many years of work, the actual implementation time required for a full commercial implementation of all the best algorithms was on the order of a few person-months total.

6.1 Contributions

The contributions of this dissertation are a combination of new algorithms and analyses of the algorithms' behavior and tradeoffs with respect to alternative solutions. For optimizing queries with expensive predicates, we presented the Predicate Migration Algorithm, and proved that it produces

optimal plans. We implemented Predicate Migration and three simpler predicate placement techniques in Illustra, and studied the effectiveness of each solution as compared to its implementation complexity. Developers can choose optimization solutions from among these techniques depending on their workloads and their willingness to invest in new code. Our experience was that Predicate Migration was not difficult to implement. Since it should provide significantly better results than any known non-exhaustive algorithm, we believe that it is the appropriate solution for any general-purpose extensible DBMS.

In the realm of query execution, we presented a new variant of hybrid hashing called Hybrid Cache, and demonstrated that it is very effective for queries with expensive predicates. For queries with expensive non-predicate methods, we showed that a choice between Hybrid Cache and sorting should be made based on the number of duplicate values in the input and the size of the output of the methods. Our analysis of the tradeoffs between Hybrid Cache and sorting highlighted new distinctions between hashing and sorting. First, we identified that hashing has performance based on the number of *values* in its input, whereas sorting has performance based on the number of *tuples* in its input. Second, we found that the performance of hashing is negatively affected by the size of the method output stored in the hash table, whereas sorting is not affected in this way. This observation applies not only to caching, but also to grouping for aggregation, since aggregate functions can produce large outputs in a system that supports extensible user-defined aggregate functions. Finally, we suggested some additional optimizations that can be applied for queries with multiple expensive methods. We view these optimizations as being less critical to performance than the implementation of Hybrid Cache, but they can be of benefit in some workloads.

6.2 Lessons

The research and implementation effort that went into this dissertation provided numerous lessons. First, it became clear that query optimization research requires a combination of theoretical insight

and significant implementation and testing. Our original cost models for Predicate Migration were fundamentally incorrect, but neither we nor many readers and reviewers of the early work were able to detect the errors. Only by implementing and experimenting were we able to gain the appropriate insights. As a result, we believe that implementation and experimentation are critical for query optimization: query optimization researchers must implement their ideas to demonstrate viability, and new complex query benchmarks are required to drive both researchers and industry into workable solutions to the many remaining challenges in query optimization.

The tradeoffs between hashing and sorting are well known and have been studied deeply by a number of people. Yet in applying these techniques to the problem of caching we discovered additional twists on the old tradeoffs. This came as a surprise: we did not expect the caching work to be particularly interesting, and considered leaving it as a simple analytic discussion of how both hashing and sorting could be applied to the problem of caching. Contrary to our expectations, the issues involved proved to be subtle and quite interesting. This reinforced the rewards of doing detailed systems research, and the necessity for an open mind when revisiting old problems in new guises.

6.3 Open Questions

This dissertation leaves a number of interesting questions open, both in optimization and execution. In optimization, it would be satisfying to understand the ramifications of weakening the assumptions of the Predicate Migration cost model. Is our roughness in estimation required for a polynomial-time predicate placement algorithm, or is there a way to weaken the assumptions and still develop an efficient algorithm?

This question can be cast in different terms. If one combines the LDL approach for expensive methods [CGK89] with the IK-KBZ *rank*-based join optimizer [IK84, KBZ86], one gets a polynomial-time optimization algorithm that handles expensive selections. However as we pointed out in Section 3.2 this technique fails because IK-KBZ does not handle bushy join trees. So an isomorphic question to

the one above is whether the IK-KBZ join-ordering algorithm can be extended to handle bushy trees in polynomial time. A positive result in this regard would be very exciting; negative results seem more probable, but would likely shed light on a number of issues.

Although Predicate Migration does not significantly affect the asymptotic cost of exponential join-ordering query optimizers, it can noticeably increase the memory utilization of the algorithms by preventing pruning. A useful extension of this work would be a deeper investigation of techniques to allow even more pruning than already available via PullRank during Predicate Migration.

Our work on predicate placement has concentrated on the traditional non-recursive select-project-join query blocks that are handled by most cost-based optimizers. It would be interesting to try and extend our work to more difficult sorts of queries. For example, how can Predicate Migration be applied to recursive queries and common subexpressions? When should predicates be transferred across query blocks? When should they be duplicated across query blocks, as sketched by Levy, *et al.* [LMS94]? These questions will require significant effort to answer, since there are no generally accepted techniques for cost-based optimization in these scenarios, even for queries without expensive predicates.

As noted in Chapter 3, the problem of benchmarking optimization schemes for expensive methods remains open. One benchmark that could be of use is a randomized macro-benchmark, to illustrate the effectiveness of the various solutions over queries from some typical distribution. Another benchmark that would be helpful would be a domain-specific benchmark with expensive predicates from real-world applications such as Asset Management [Ols95] or Remote Sensing [Doz92]; of course this would be most convincing if it were agreed upon by the users in these domains. A very interesting micro-benchmark would be to analyze the sensitivity of Predicate Migration and other schemes to estimation errors — this is of particular importance since cost and selectivity estimation for user-defined methods in Illustra is out of the control of the database implementor.

An alternative way to approach the problem of cost and selectivity estimation is to try and develop schemes to dynamically adjust these statistics as queries are executed. Method invocation times could

be measured and fit to cost functions based on the size of the input to the methods. Similarly, statistics could be kept on predicate method evaluations, to dynamically refine selectivity estimates based on input distributions.

Finally, additional performance work could be done to explore some of the remaining details in method caching, particularly for queries with multiple methods. The techniques proposed in the dissertation for sharing caches and for query rewriting merit further investigation. Another issue remaining is that we have only studied how to do caching "in the large", for situations where disk I/O may be required. Additional work may be required to study the problem of optimally doing caching for inputs that fit in memory. Such work will become increasingly important as the cost of RAM decreases and ever larger amounts of main memory are available.

6.4 Suggestions for Implementation

Practitioners are often wary of adding large amounts of code to existing systems. Fortunately, the work in this dissertation can be implemented incrementally, providing increasing benefits as more pieces are added. We sketch a plausible upgrade path for existing extensible systems:

1. Sorting is available in almost all DBMSs, and can be used directly to handle method caching. This is a fairly trivial optimization to implement, requiring two small changes to an existing system:

   - the optimizer must identify expensive methods and insert sort nodes into plans just below the method invocations, as needed
   - the executor must check a one-tuple cache before invoking the method, and update the cache after each invocation.

2. As a first step in optimization, selections should be ordered by *rank* (the PushDown+ heuristic). Since typical optimizers already estimate selectivities, this only requires estimating selection

costs, and sorting selections.

3. As a further enhancement in optimization, the PullUp heuristic can be implemented. This requires code to pull selections above joins, which is a bit tricky since it requires handling projections and column renumbering. Note that PullUp is not necessarily more effective than PushDown for all queries, but is probably a safer heuristic if methods are expected to be very expensive.

4. The PullUp heuristic can be be modified to do PullRank, by including code to compute the differential costs for joins. Since PullRank can leave methods too low in a plan, it is in some sense more dangerous than PullUp. On the other hand, it can correctly optimize all two-table queries, which PullUp can not guarantee. A decision between PullUp and PullRank is thus somewhat difficult. However, PullRank is a preprocessor for a full implementation of Predicate Migration, and hence a step in the right direction.

5. Predicate Migration itself, while somewhat complex to understand, is actually not difficult to implement. It requires implementation of the Predicate Migration Algorithm, PullRank, and also code to mark subplans as unpruneable.

6. Finally, execution can be further improved by implementing Hybrid Cache for expensive predicate methods. If hybrid hash join has already been implemented, Hybrid Cache is easier to add. If hybrid hash join has not been implemented, then adding Hybrid Cache is a first step in that direction.

6.5 Closing

In the coming years, database management systems will be called upon to handle many new types of non-standard data, both from domains we have already identified, and also undoubtedly from domains we have yet to consider. Extensible database systems will be required so that this multiplicity of unforseeable new data types can be handled easily in a single system.

This dissertation presents a significant portion of the technology required to make such extensible systems process queries efficiently. Our experience with Illustra suggests that implementing this new technology is a manageable and highly beneficial undertaking for any extensible database management system.

Bibliography

[BC81] P. A. Bernstein and D. W. Chiu. Using Semijoins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, 1981.

[BDT83] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems, a systematic approach. In *Proc. 9th International Conference on Very Large Data Bases*, Florence, Italy, October 1983.

[Bra84] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proc. 10th International Conference on Very Large Data Bases*, pages 323–333, Singapore, August 1984.

[Cat94] R. G. G. Cattell. *The Object Database Standard: ODMG-93 (Release 1.1).* Morgan Kaufmann, San Mateo, CA, 1994.

[CDKN94] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A Status Report on the OO7 OODBMS Benchmarking Effort. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 414–426, Portland, October 1994.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an Open Architecture for LDL. In *Proc. 15th International Conference on Very Large Data Bases*, pages 195–203, Amsterdam, August 1989.

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

[CYY+92] H. Chen, X. Yu, K. Yamaguchi, H. Kitagawa, N. Ohbo, and Y. Fujiwara. Decomposition — An Approach for Optimizing Queries Including ADT Functions. *Information Processing Letters*, 43(6):327–333, 1992.

[Dat83] C. J. Date. The Outer Join. In *Proceedings of the Second International Conference on Databases*, Cambridge, England, September 1983.

[Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. 13th International Conference on Very Large Data Bases*, pages 197–208, Brighton, September 1987.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 1–8, Boston, June 1984.

[DKS92] W. Du, R. Krishnamurthy, and M. Shan. Query Optimization in Heterogeneous DBMS. In *Proc. 18th International Conference on Very Large Data Bases*, pages 277–291, Vancouver, August 1992.

[Doz92] J. Dozier. Access to Data in NASA's Earth Observing System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, San Diego, page 1, June 1992.

[FK94] C. Faloutsos and I. Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 4–13, Minneapolis, May 1994.

[FM85] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31, 1985.

[Fre95] J. Frew. Personal correspondence, July 1995.

[Gra91] J. Gray, editor. *The Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan-Kaufmann Publishers, Inc., 1991.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra95] J. Gray. Personal correspondence, July 1995.

[Han77] M. Z. Hanani. An Optimal Evaluation of Boolean Expressions in an Online Query System. *Communications of the ACM*, 20(5):344–347, may 1977.

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

[Hel94] J. M. Hellerstein. Practical Predicate Placement. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 325–335, Minneapolis, May 1994.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 377–388, Portland, May-June 1989.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[HOT88] W. Hou, G. Ozsoyoglu, and B. 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.

[HS93a] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 267–276, Washington, D.C., May 1993.

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

[IC91] Y. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 268–277, Denver, June 1991.

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

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 312–321, Atlantic City, May 1990.

[Ill94] Illustra Information Technologies, Inc. *Illustra User's Guide, Illustra Server Release 2.1*, June 1994.

[IP95] Y. Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 233-244, San Jose, May 1995.

[ISO93] ISO_ANSI. Database Language SQL ISO/IEC 9075:1992, 1993.

[ISO94] ISO. ISO_ANSI Working Draft: Database Language SQL (SQL3) TC X3H2-94-331; ISO/IEC JTC1/SC21/WG3, 1994.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

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

[Kim93] W. Kim. Object-Oriented Database Systems: Promises, Reality, and Future. In *Proc. 19th International Conference on Very Large Data Bases*, pages 676–687, Dublin, August 1993.

[KMPS94] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing Disjunctive Queries with Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 336–347, Minneapolis, May 1994.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., 1973.

[KZ88] R. Krishnamurthy and C. Zaniolo. Optimization in a Logic Based Language for Knowledge and Data Intensive Applications. In J. W. Schmidt, S. 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.

[LDH+84] G. M. Lohman, D. Daniels, L. M. Haas, R. Kistler, and P. G. Selinger. Optimization of Nested Queries in a Distributed Relational Database. In *Proc. 10th International Conference on Very Large Data Bases*, pages 403–415, Singapore, August 1984.

[LH93] G M. Lohman and L. M. Haas. Personal correspondence, November 1993.

[LMS94] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query Optimization by Predicate Move-Around. In *Proc. 20th International Conference on Very Large Data Bases*, pages 96–107, Santiago, September 1994.

[LNSS93] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science*, (116):195–226, 1993.

[Loh95] G. M. Lohman. Personal correspondence, July 1995.

[LS88] C. Lynch and M. Stonebraker. Extended User-Defined Indexing with Application to Textual Databases. In *Proc. 14th International Conference on Very Large Data Bases*, pages 306–317, Los Angeles, August-September 1988.

[Mic68] D. Michie. "Memo" Functions and Machine Learning. *Nature*, (218):19–22, April 1968.

[ML86a] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Local Queries. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 84–95, Washington, D.C., May 1986.

[ML86b] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. 12th International Conference on Very Large Data Bases*, pages 149–159, Kyoto, August 1986.

[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 K. R. Dittrich and U. Dayal, editors, *Proc. Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar, September 1986.

[Nat87] K. S. Natarajan. Optimizing Backtrack Search for All Solutions to Conjunctive Problems. In *Proceedings IJCAI-87*, pages 955–958, Milan, 1987.

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

[NKT88] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proc. 14th International Conference on Very Large Data Bases*, pages 468–477, Los Angeles, August-September 1988.

[Ols95] M. A. Olson. Cover Your Assets. In *Proc. ACM-SIGMOD International Conference on Management of Data*, San Jose, page 453, May 1995.

[Pal74] F. P. Palermo. A Data Base Search Problem. In J. T. Tou, editor, *Information Systems COINS IV*. Plenum Press, New York, 1974.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. 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.

[Pir94] H. Pirahesh. Object-Oriented Features of DB2 Client/Server. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, page 483, May 1994.

[Raa95] F. Raab. *"TPC Benchmark D – Standard Specification, Revision 1.0"*. Transaction Processing Performance Council, May 1995.

[SAC+79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 22–34, Boston, June 1979.

[Sel88] T. Sellis. Multiple Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Storage Benchmark. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 2–11, Washington, D.C., May 1993.

[SG88] A. Swami and A. Gupta. Optimization of Large Join Queries. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 8–17, Chicago, June 1988.

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

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

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, 1991.

[Smi56] W. E. Smith. Various Optimizers For Single-Stage Production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.

[Smi89] D. E. Smith. Controlling Backward Inference. *Artificial Intelligence*, 39:145–208, 1989.

[SPMK95] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing Joins in Disjunctive Queries. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[Sto91] M. 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.

[Swa89] A. Swami. Optimization of Large Join Queries: Combining Heuristics with Combinatorial Techniques. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 367–376, Portland, May-June 1989.

[TOB89] C. Turbyfill, C. Orji, and D. Bitton. AS^3AP - A Comparative Relational Database Benchmark. In *Proc. IEEE Compcon Spring '89*, February 1989.

[WY76] E. Wong and K. Youssefi. Decomposition - A Strategy for Query Processing. *ACM Transactions on Database Systems*, 1(3):223–241, September 1976.

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