MULTI-QUERY OPTIMIZATION AND APPLICATIONS

Submitted in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

by

PRASAN ROY

Department of Computer Science and Engineering

Indian Institute of Technology - Bombay

2000

Approval Sheet

The thesis entitled MULTI-QUERY OPTIMIZATION AND APPLICATIONS

by PRASAN ROY

is approved for the degree of DOCTOR OF PHILOSOPHY.

Examiners

_____

_____

_____

_____

Supervisor

_____

Chairman

_____

Date : _____

Place : _____

Abstract

Complex queries are becoming commonplace with the growing use of decision support systems. These complex queries often have a lot of common sub-expressions, either within a single query, or across multiple such queries. The focus of this work is to speed up query execution by exploiting these common subexpressions.

Given a set of queries in a batch, multi-query optimization aims at exploiting common subexpressions among these queries to reduce evaluation cost. Multi-query optimization has hitherto been viewed as impractical, since earlier algorithms were exhaustive, and explore a doubly exponential search space. We present novel heuristics for multi-query optimization, and demonstrate that optimization using these heuristics provides significant benefits over traditional optimization, at a very acceptable overhead in optimization time.

In online environments, where the queries are posed as a part of an ongoing stream instead of in a batch, individual query response times can be greatly improved by caching final/intermediate results of previous queries, and using them to answer later queries. An automatic caching system that makes intelligent decisions on what results to cache would be an important step towards knobs-free operation of a database system. We describe an automatic query caching system called *Exchequer* which is closely coupled with the optimizer to ensure that the caching system and the optimizer make mutually consistent decisions, and experimentally illustrate the benefits of this approach.

Further, because the presence of views enhances query performance, materialized views are increasingly being supported by commercial database/data warehouse systems. Whenever the data warehouse is updated, the materialized views must also be updated. We show how to find

an efficient plan for maintenance of a *set* of views, by exploiting common subexpressions between different view maintenance expressions. These common subexpressions may be materialized temporarily during view maintenance. Our algorithms also choose additional subexpressions/indices to be materialized permanently (and maintained along with other materialized views), to speed up view maintenance. In addition to faster view maintenance, our algorithms can also be used to efficiently select materialized views to speed up query workloads.

Contents

List of Figures

Chapter 1

Introduction

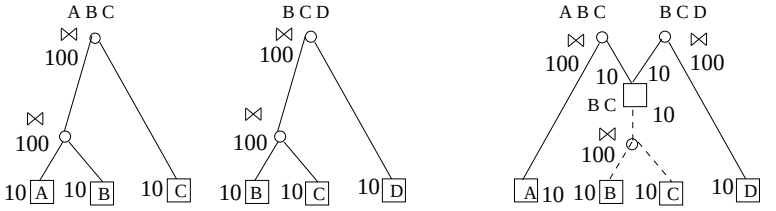
Complex queries are becoming commonplace, especially due to the advent of automatic tools that help analyze information from large data warehouses. These complex queries often have several subexpressions in common since i) they make extensive use of views which are referred to multiple times in the query and ii) many of them are correlated nested queries in which parts of the inner subquery may not depend on the outer query variables, thus forming a common subexpression for repeated invocations of the inner query.

1.1 Problem Overview and Motivation

The focus of this thesis is to speed up query processing by sharing computation within or across queries by materializing intermediate results. This can be done at three levels: transient, dynamic and permanent.

1.1.1 Transient Materialization

Given a batch of queries to be executed, the results computed during the execution can be materialized on the disk as they are computed when refered for the first time, reused on later references instead of being recomputed, and discarded at the end of the execution. This is termed transient materialization.



(a) Traditional execution: No sharing
Total Cost = 460

(b) Execution with BC shared
Total Cost = 370

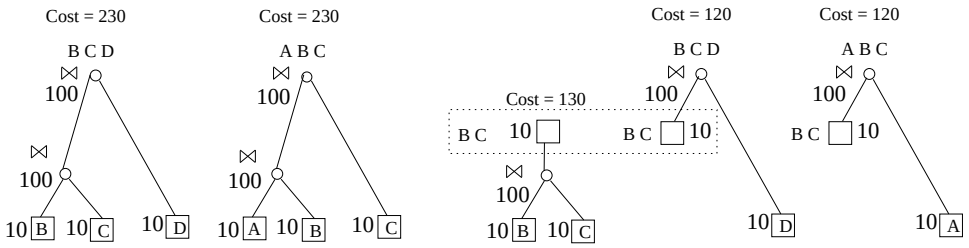
Figure 1.1: Example illustrating benefits of sharing computation

Example 1.1.1 Consider a batch of two queries $\{(A \bowtie B \bowtie C), (B \bowtie C \bowtie D)\}$. A traditional system will execute each of these queries independently using the individual best plans as suggested by the query optimizer; let these best plans be as shown in Figure 1.1(a). The base relations A, B, C and D each have a scan cost of 10 units.[1] Each of the joins have a cost of 100 units, giving a total execution cost of 460 units. On the other hand, in the plan shown in Figure 1.1(b), the intermediate result $(B \bowtie C)$ is first computed and materialized on the disk at a cost of 10. Then, it is scanned back twice – the first time to join with A in order to compute $(A \bowtie B \bowtie C)$, and the second time to join with D in order to compute $(B \bowtie C \bowtie D)$ – at a cost of 10 per scan. Each of these joins have a cost of 100 units. The total cost of this *consolidated* plan is thus 370 units, which is about 20% less than the cost of the traditional plan of Figure 1.1(a), demonstrating the benefit of sharing computation during query processing. □

The expression $(B \bowtie C)$ that is common between the two queries $(A \bowtie B \bowtie C)$ and $(B \bowtie C \bowtie D)$ in the above example is termed as a *common subexpression*. We address the problem of finding the cheapest execution plan for a batch of queries, exploiting transiently materialized common subexpressions; this is termed *multi-query optimization*. Section 1.2.1 provides further details of our work on multi-query optimization.

Multi-query optimization is an important practical problem. For instance, SQL-3 stored procedures may invoke several queries, which can be executed as a batch. Further, data analysis/reporting often requires a batch of queries to be executed. Recent work on using relational databases for storing XML data, has found that queries on XML data, written in a language

[1] The actual unit of measure is not relevant to this example



(a) Execution without query caching

(b) BC cached during execution of BCD reused during execution of ABC

Figure 1.2: Example illustrating the benefit of caching intermediate results

such as XML-QL and containing regular path expressions, are translated into a batch of relational queries; these queries have a large amount of overlap and can benefit significantly from multi-query optimization.

1.1.2 Dynamic Materialization

In online environments, where the queries are posed as a part of an ongoing stream instead of in a batch as above, individual query response times can be greatly improved by caching final/intermediate results of previous queries, and using them to answer later queries.

Given a sequence of queries arriving individually, each executed on arrival, dynamic materialization involves materializing, in a limited-size cache, results computed during the execution of individual queries. These results are used to compute queries appearing later in the sequence, and may be discarded at a later point of time when they lose their utility.

Example 1.1.2 Consider again the two queries $(B \bowtie C \bowtie D)$ and $(A \bowtie B \bowtie C)$ of Example 1.1.1, this time occurring one after another as a part of a workload sequence. As earlier, the queries are on the base relations A, B, C and D each having a scan cost of 10 each. The execution of the two queries, when caching is not supported, costs 230 for each query as shown in Figure 1.2(a), totaling to 460. Contrast this with the execution of the queries as shown in Figure 1.2(b). In this case, during the execution of $(B \bowtie C \bowtie D)$, the intermediate result $(B \bowtie C)$ is cached to the disk, at a cost of 10, and reused at a cost of 10 per query; the total execution cost

for the two queries is now 370. This illustrates the benefit of caching and reusing intermediate results. □

We use the term *query result caching* to mean caching of final and/or intermediate results of queries. Query result caching differs from multi-query optimization in that at the moment a given query is being executed, later queries in the workload sequence are not known. The main issue in query result caching is thus to dynamically determine the utility of a result, so as to figure out when to admit it into the cache and when to dispose it in favor of another result. Further details of our work on query result caching appear in Section 1.2.2.

1.1.3 Permanent Materialization

Permanent materialization involves precomputing results and keeping them materialized on the disk during the execution of the workload. However, unlike transient and dynamic materialization, these results are never discarded. Permanently meterialized results are also called *materialized views*.

Materialized views have dependencies on the underlying base relations – when these base relations get updated, the system needs to refresh these views in order to maintain consistency. The view can be refreshed by either recomputing it, or by first computing the incremental change to the view (tuples to be inserted or deleted as a consequence of the corresponding updates to the base relations) and then integrating the change into the view. This is termed *view maintenance*.

In current generation database systems, the system administrator can decide to permanently materialize a set of views, and the system must keep these views consistent by refreshing them. Efficient techniques for view maintenance are needed because whereas the amount of data entering a warehouse, the query loads, and the need to obtain up-to-date responses are all increasing, the time window available for making the warehouse up-to-date is shrinking.

We address the problem of minimizing the total cost of maintaining the given set of views. In order to do so, we show how to determine (a) for each materialized view, the best way to refresh it in face of updates to the base relation; and (b) an additional set of results to materialize, permanently or transiently, to speed up the refresh processing. These decisions are interdependent

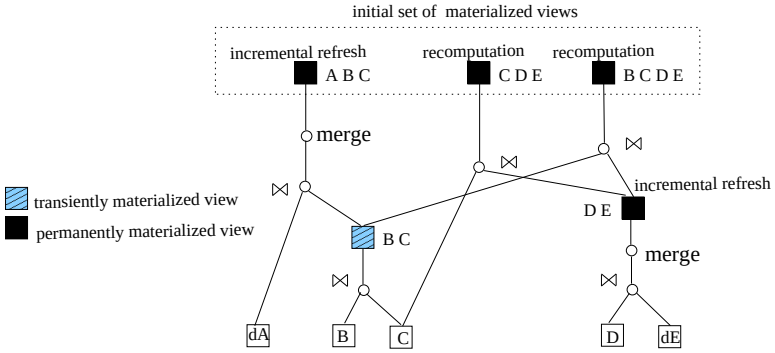


Figure 1.3: Example view maintenance plan. *Merge* refreshes a view given its "delta".

and need to be taken in an interleaved manner. This is termed *materialized view selection and maintenance*. Further details of our approach for materialized view selection and maintenance is presented in Section 1.2.3.

Example 1.1.3 Suppose we have three materialized views $V1 = (A \bowtie B \bowtie C)$, $V2 = (C \bowtie D \bowtie E)$ and $V3 = (B \bowtie C \bowtie D \bowtie E)$, and relations A and E are updated. In this example, we assume that the updates to A and E consist of inserts to the relations, and the other relations are not changed. This reflects reality in data warehouses, where only a few of the relations are updated. (However, our techniques do not have any restrictions on what is updated, or what is the form of the updates.)

If the maintenance plans of the three views are chosen independently, the best view maintenance plan (incremental or recomputation) for each would be chosen, without any sharing of computation. In contrast, as an illustration of the kind of plans our optimization methods are able to generate, Figure 1.3 shows a maintenance plan for the views that exploits sharing of computation. Here, $(A \bowtie B \bowtie C)$ is refreshed incrementally, while $(C \bowtie D \bowtie E)$ and $(B \bowtie C \bowtie D \bowtie E)$ are recomputed.

Two extra views, $(B \bowtie C)$ and $(D \bowtie E)$ have been chosen to be materialized. Of these, $(B \bowtie C)$ is materialized *transiently*, and is disposed as soon as the views are refreshed; this could happen because there are also updates on B and C which make it expensive to maintain $(B \bowtie C)$ as a materialized view.

The result $(D \bowtie E)$ has been chosen to be materialized *permanently*, and is itself refreshed

incrementally given the updates to the relation E. Its full result is then used to recompute $(C \bowtie D \bowtie E)$ as well as $(B \bowtie C \bowtie D \bowtie E)$. If an incremental maintenance plan had been chosen for $(B \bowtie C \bowtie D \bowtie E)$, with recomputation chosen for $(C \bowtie D \bowtie E)$, the differential result of $(D \bowtie E)$ would have been used in the incremental maintenance plan while the full result of $(D \bowtie E)$ would be used in the recomputation plan. □

1.2 Summary of Contributions

In this section we give a summary of the main contributions of the different chapters of the thesis. Section 1.2.1 describes our work on multi-query optimization which involves transient materialization, Section 1.2.2 describes our work on query result caching which involves dynamic materialization, and Section 1.2.3 describes our work on materialized view selection and maintenance which involves transient materialization as well as permanent materialization.

In addition to our technical contributions detailed below, another of our contributions lies in showing, for each of the above problems, how to engineer practical systems by adding just a few thousand lines of code to existing state-of-the-art query optimizers (in particular, the Volcano query optimizer engine [23], which forms the core of the Microsoft SQL-Server [22] and Tandem ServerWare SQL [6] optimizers).

1.2.1 Multi-Query Optimization

In Chapter 3, we address the problem of optimizing a set of queries exploiting the presence of common subexpressions among the queries; this problem is referred to as *multi-query optimization*. Common subexpressions are possible even *within* a single query; the techniques we develop deal with such intra-query common subexpressions as well. Traditional query optimizers are not appropriate for optimizing queries with common subexpressions, since they make locally optimal choices, and may miss globally optimal plans.

The job of multi-query optimizer, over and above that of ordinary query optimizer, is to (i) recognize the possibilities of shared computation, and (ii) modify the optimizer search strategy

to explicitly account for shared computation and find a globally optimal plan.

The contributions of this work are as follows:

1. The search space for multi-query optimization is doubly exponential in the size of the queries, and exhaustive strategies are therefore impractical; as a result, multi-query optimization was hitherto considered too expensive to be useful. We show how to make multi-query optimization *practical*, by developing novel heuristic algorithms.

   Further, our algorithms can be easily extended to perform multi-query optimization on nested queries as well as multiple invocations of parameterized queries (with different parameter values). Our algorithms also take into account sharing of computation based on "subsumption" – examples of such sharing include computing $\sigma_{A<5}(E)$ from the result of $\sigma_{A<10}(E)$.

   Our algorithms are independent of the data model and the cost model, and are extensible to new operators.

2. In addition to choosing what intermediate expression results to materialize and reuse, our optimization algorithms also choose physical properties, such as sort order, for the materialized results. By modelling presence of an index as a physical property, our algorithms also handle the choice of what (temporary) indices to create on materialized results/database relations.

We believe that in addition to our technical contributions, another of our contributions lies in showing how to engineer a practical multi-query optimization system — one which can smoothly integrate extensions, such as indexes and nested queries, allowing them to work together seamlessly.

1.2.2 Query Result Caching

In a traditional database engine, every query is processed independently. In decision support applications, queries often overlap in the data that they access and in the manner in which they utilize the data, i.e., there are common expressions between queries. A natural way to improve

performance is to allocate a limited-size area on the disk to be used as a cache for results computed by previous queries. The contents of the cache may be utilized to speed up the execution of subsequent queries. We use the term *query caching* to mean caching of final and/or intermediate results of queries.

Most existing decision support systems support static view selection: select a set of views apriori, and keep them permanently on disk. The selection is based on either (a) the intuition of the systems administrator, or (b) recommendation of "advisor wizards" as supported by Microsoft SQL-Server [1] based on a workload history. The advantage of query caching over static view selection is that it can cater to changing workloads — the data access patterns of the queries cannot be expected to be static, and to answer all types of queries efficiently, we need to dynamically change the cache contents.

In Chapter 4, we present the techniques needed (a) for intelligently and automatically managing the cache contents, given the cache size constraints, as queries arrive, and (b) for performing query optimization exploiting the cache contents, so as to minimize the overall response time for all the queries.

The contributions of this work are:

1. We show how to handle the caching of intermediate as well as final results of queries. Intermediate results, in particular, require careful handling since caching decisions are typically made based on usage rates, and usage rates of intermediate results are dependent on what *else* is in the cache. Techniques for caching intermediate results were proposed in [10], but they are based only on usage rates and would be biased against results that are currently not in the cache. Our caching algorithms use sophisticated techniques for deciding what to cache, taking into account what other results are cached. Moreover, we show how to consider caching indices constructed on the fly in the same way as we consider caching of intermediate results.

2. We show how to enable the optimizer to take into consideration the use of cached results and indices, piggybacked on the optimization step *with negligible overhead*. All prior cache-aware optimization algorithms have a separate cache result matching step.

3. Our algorithms are extensible to new operations, unlike much of the prior work on caching. Moreover, prior work has mainly concentrated on cube queries; while cube queries are important, general purpose decision support systems must support more general queries as well. Our algorithms can handle any SQL query including nested queries. To the best of our knowledge, no other caching technique is capable of handling caching of intermediate results for such a general class of queries.

4. We have implemented the proposed techniques and present a performance study that clearly demonstrates the benefits of our approach. Our study shows that intelligent, workload adaptive intermediate query result caching can be done fast enough to be practical, and leads to significant overall savings.

In this work, we confine our attention only to the issue of efficient query processing, ignoring updates. Data Warehouses are an example of an application where the cache replacement algorithm can ignore updates, since updates happen only periodically (once a day or even once a week).

1.2.3 Materialized View Selection and Maintenance

Materialization of views can help speed up query and update processing. Views are especially attractive in data warehousing environments because of the query intensive nature of data warehouses. However, when a warehouse is updated, the materialized views must also be updated. Typically, updates are accumulated and then applied to a data warehouse. Loading of updates and view maintenance in warehouses has traditionally been done at night. While the need to provide up-to-date responses to an increasing query load is growing and the amount of data that gets added to data warehouses has been increasing, the time window available for making the warehouse up-to-date has been shrinking. These trends call for efficient techniques for maintaining the materialized views as and when the warehouse is updated.

Chapter 5 addresses the problem of efficiently maintaining a *set* of materialized views. Although the focus of our work is to speed up view maintenance, our algorithms can also be used to choose extra transient and permanent views in order to speed up a workload containing queries and updates (that trigger view maintenance).

Our contributions are as folows:

1. We show how to exploit transient materialization of common subexpressions to reduce the cost of view maintenance plans.

   Sharing of subexpressions occurs when multiple views are being maintained, since related views may share subexpressions, and as a result the maintenance expressions may also be shared. Furthermore, sharing can occur even within the plan for maintaining a single view if the view has common subexpressions within itself.

   The shared expressions could include differential expressions, as well as full expressions which are being recomputed.

2. We show how to efficiently choose additional expressions for permanent materialization to speed up maintenance of the given views.

   Just as the presence of views allows queries to be evaluated more efficiently, the maintenance of the given permanently materialized views can be made more efficient by the presence of additional permanently materialized views [45, 44]. That is, given a set of materialized views to be maintained, we choose additional views to materialize in order to minimize the overall view maintenance costs. The expressions chosen for permanent materialization may be used in only one view maintenance plan, or may be shared between different views maintenance plans.

3. We show how to determine the optimal maintenance plan for each individual view, given the choice of results for transient/permanent materialization.

   Maintenance of a materialized view can either be done *incrementally* or *by recomputation*. Incremental view maintenance involves computing the differential ("delta"s) of a materialized view, given the "delta"s of the base relations that are used to define the views, and merging it with the old value of the view. However, incremental view maintenance may not always be the best way to maintain a materialized view; when the deltas are large the view may be best maintained by recomputing it from the updated base relations.

Our techniques determine the maintenance policy, incremental or recomputation, for each view in the given set such that the overall combination has the minimum cost.

4. We show how to make the above three choices in an integrated manner to minimize the overall cost.

   It is important to point out that the above three choices are highly interdependent, and must be taken in such a way that the overall costs of maintaining a set of views is minimized. Specifically:

   - Given a subexpression useful during materialization of multiple views, choosing whether it should be transiently or permanently materialized is an optimization problem, since each alternative has its cost and benefit. Transient views are materialized during the evaluation of the maintenance plan and discarded after maintenance of the given views; such transient views themselves need not be maintained. On the other hand, the permanent views are materialized a priori, so there is no (re)computation cost; however, there is a maintenance cost, and a storage cost (which is long term in that it persists beyond the view maintenance period) due to the permanently materialized views.
   - The choice of additional views must be done in conjunction with selecting the plans for maintaining the views, as discussed above. For instance, a plan that seems quite inefficient could become the best plan if some intermediate result of the plan is chosen to be materialized and maintained.

   We propose a framework that cleanly integrates the choice of additional views to be transiently or permanently materialized, the choice of whether each of the given set of (user-specified) views must be maintained incrementally or by recomputation, and the choice of view maintenance plans.

5. We have implemented all our algorithms, and present a performance study, using queries from the TPC-D benchmark, showing the practical benefits of our techniques.

1.3 Organization of the Thesis

This thesis is organized as follows. Chapter 2 gives a brief background overview of traditional query optimization. In particular, it describes our version of the Volcano optimization framework; the work presented in later chapters is based on this framework. Chapter 3 gives the details of our work on multi-query optimization. This is followed by Chapter 4 which addresses the query result caching problem. Chapter 5 describes how the multi-query optimization framework is extended to address the materialized view selection and maintenance problem. Finally, the conclusions of the thesis and directions for future work appear in Chapter 6.

Chapter 2

Traditional Query Optimization

This chapter sets the stage for the work covered in the rest of the thesis. Section 2.1 gives a brief overview of the important concerns and prior work in traditional query optimization. Section 2.2 describes the design and implementation of a query optimizer. Later chapters of this thesis build on the framework described in this section.

2.1 Background

In this section, we provide a broad overview of the main issues involved in traditional query optimization and mention some of the representative work in the area. This discussion will be kept very brief; for the details we point to the comprehensive, very readable survey by Chaudhuri [7].

Traditionally, the core applications of database systems have been online transaction processing (OLTP) environments like banking, sales, etc. The queries in such an environment are simple, involving a small number of relations, say three to five. For such simple queries, the investment in sophisticated optimization usually did not pay up in the performance gain. As such, only join-order optimization and that too in a constrained search space was effective enough. The seminal paper by Selinger et al. [51] presented a dynamic programming algorithm for searching optimal left-linear join ordered plans. The ideas presented in this paper formed the basis of most

optimization research and commercial development till a few years back.

However, with the growing importance of online analytical processing (OLAP) environments, which routinely involve expensive queries, more sophisticated query optimization techniques have become crucial. In order to be effective in such demanding environments, the optimizers need to look at less constrained search spaces without loosing much on efficiency. They need to adapt to new operators, new implementations of these operators and their cost models, changes in cost estimation techniques, etc. This calls for extensibility in the optimizer architecture. These requirements led to the current generation of query optimizers, of which two representative optimizers are Starburst [40] and Volcano [23]. While the IBM DB2 optimizer [20] is based on Startburst, the Microsoft SQL-Server optimizer [22] is based on Volcano. The main difference between the approaches taken by the two is the manner in which alternative plans are generated. Starburst generates the plans bottom-up – that is, best plans for all expressions on k relations are computed before expressions on more than k relations are considered. On the other hand, Volcano generates the plans top-down – that is, it computes the best plans for only those expressions on k relations which are included in some expression on greater than k relations being expanded.

The need for effective optimization of large, complex queries has brought focus to the intimately related problem of statistics and cost estimation. This is because the cost-based decisions of an optimizations can only be as reliable as its estimates of the cost of the generated plans.

A plan is composed of operators (e.g. select, join, sort). The cost of an operator is a function of the statistical summary of its input relations, which includes the size of the relation, and for each relevant attribute, the number of distinct values of the attribute, the distribution of these attribute values in terms of an histogram, etc. While the accuracy of these statistics is crucial – the plan cost estimate may be sensitive to these statistics – the maintenance of these statistics may be very time consuming. The problem of efficiently maintaining reasonably accurate statistics has received much attention in the literature; for the details, we refer to the paper by Poosala et al. [41].

Even if we have perfect information about the input relations, modeling the cost of the operators could still be very difficult. This is because a reasonable cost model must take into account

the affect of, for example, the buffering of the relations in the database cache, access patterns of the inputs, the memory available for the operator's execution, etc. Moreover, usually the plans execute in a pipeline – that is, multiple operators may execute simultaneously. Given the system's bounded resources like CPU and main memory, the execution of these operators may interfere, affecting the execution cost of the plan. There has been much research on cost modeling; an authoritative, very comprehensive survey by Graefe [21] provides the details of the prior work in this area.

2.2 Design of a Cost-based Query Optimizer

In this section, we describe the design of a cost-based transformational query optimizer, based on the Volcano optimizer [23].

There are two main advantages of using Volcano as the basis of our work. The first is that Volcano has gained widespread acceptance in the industry as a state-of-the-art optimizer; the optimizers of Microsoft SQL Server [22] and Tandem ServerWare SQL Product [6] are based on Volcano. Our work is easily integrable into such systems. Secondly, the Volcano optimization framework is not dependent on the data model or on the execution model. This makes Volcano extensible to new data models (e.g. use of Volcano optimization for object oriented systems was reported in [4]) and for new transformations, operators and implementations.

The implementation of this query optimizer worked out to around 17,000 lines of C++ code. Later chapters in this thesis, describing our work on multi-query optimization, query result caching and materialized view selection and maintenance, build on the framework described in this section. Each of these extensions could be implemented in about another 3,000 lines of C++ code.

2.2.1 Overview

Figure 2.1 gives an overview of the optimizer. Given the input query, the optimizer works in three distinct steps:

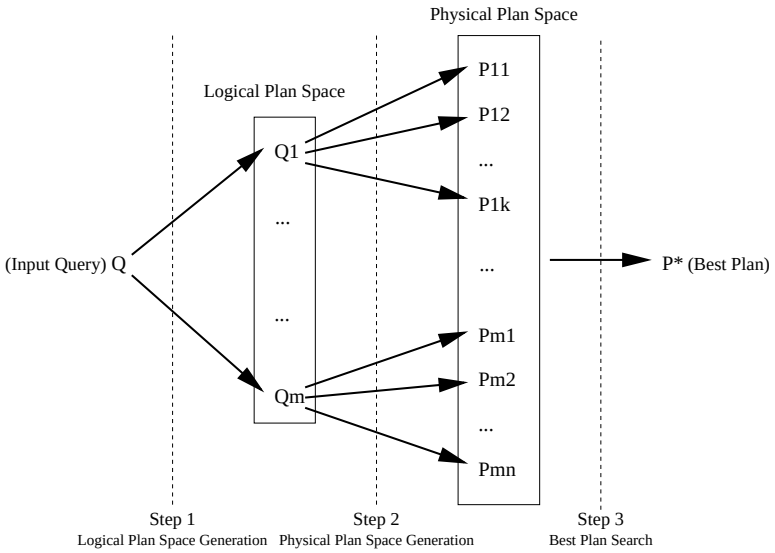


Figure 2.1: Overview of Cost-based Transformational Query Optimization

1. *Generate all the semantically equivalent rewritings of the input query.*

   In Figure 2.1, $Q_1, \ldots, Q_m$ are the various rewritings of the input query Q. These rewritings are created by applying "transformations" on different parts of the query; a transformation gives an alternative semantically equivalent way to compute the given part. For example, consider the query $(A \bowtie (B \bowtie C))$. The *join commutativity* transformation says that $(B \bowtie C)$ is semantically equivalent to $(C \bowtie B)$, giving $(A \bowtie (C \bowtie B))$ as a rewriting.

   An issue here is how to manage the application of the transformation so as to guarantee that all rewritings of the query possible using the given set of transformations are generated, in as efficient way as possible.

   For even moderately complex queries, the number of possible rewritings can be very large. So, another issue is how to efficiently generate and compactly represent the set of rewritings.

   This step is explained in Section 2.2.2.

2. *Generate the set of executable plans for each rewriting generated in the first step.*

   Each rewriting generated in the first step serves as a *template* that defines the order in

which the logical operations (selects, joins, aggregates) are to be performed – how these operations are to be executed is not fixed. This step generates the possible alternative execution plans for the rewriting. For example, the rewriting $(A \bowtie (C \bowtie B))$ specifies that A is to be joined with the result of joining C with B. Now, suppose the join implementations supported are nested-loops-join, merge-join and hash-join. Then, each of the two joins can be performed using any of these three implementations, giving nine possible executions of the given rewriting.

In Figure 2.1, $P_{11}, \ldots, P_{1k}$ are the k alternative execution plans for the rewriting Q_1, and $P_{m1}, \ldots, P_{mn}$ are the n alternative execution plans for Q_m.

The issue here, again, is how to efficiently generate the plans and also how to compactly store the enormous space of query plans.

This step is explained in Section 2.2.3

3. *Search the plan space generated in the second step for the "best plan".*

   Given the cost estimates for the different algorithms that implement the logical operations, the cost of each execution plans is estimated. The goal of this step is to find the plan with the minimum cost.

   Since the size of the search space is enormous for most queries, the core issue here is how to perform the search efficiently. The Volcano search algorithm is based on top-down dynamic programming ("memoization") coupled with branch-and-bound.

   Details of the search algorithm appear in Section 2.2.4.

For clarity of understanding, we take the approach of executing one step fully before moving to the next in the rest of this chapter. This is the approach that will be extended on in the later chapters. However, this may not be the case in practice; in particular, the original Volcano algorithm does not follow this execution order; Volcano's approach is discussed in Section 2.2.5.

In order to emphasize the "template-instance" relationship between the rewritings and the execution plans, we hereafter refer to them as *logical plans* and *physical plans* respectively.

2.2.2 Logical Plan Space

The logical plan space is the set of all semantically equivalent logical plans of the input query. We begin with a description of the *logical transformations* used to generate the logical plan space. The logical plan space is typically very large; a compact representation of the same, called the *Logical Query DAG* representation is described next. Further, the algorithm to generate all the logical plans possible given the set of transformations, compactly represented as a Logical Query DAG, is presented. Lastly, we give the rationale of choosing Volcano optimization as the basis of our work.

Logical Transformations

The logical transformations specify the semantic equivalence between two expressions to the optimizer. Examples of logical transformations are:

- *Join Commutativity:* $(A \bowtie B) \rightarrow (B \bowtie A)$
- *Join Associativity:* $((A \bowtie B) \bowtie C) \rightarrow (A \bowtie (B \bowtie C))$
- *Predicate Pushdown:* $(A \bowtie_{\theta \wedge \theta'} B) \rightarrow (A \bowtie_{\theta} \sigma_{\theta'}(B))$ if all attributes used in θ' are from B.

The complexity of the logical plan generation step, described below, depends on the given set of transformations; an unfortunate choice of transformations can lead to the generation of the same logical plan multiple times along different paths. Pellenkroft et al. [39] present a set of transformations that avoid this redundancy.

The complete list of logical transformations used in our optimizer is given in Appendix B.

Logical Query DAG Representation

A *Logical Query DAG* (LQDAG) is a directed acyclic graph whose nodes can be divided into *equivalence nodes* and *operation nodes*; the equivalence nodes have only operation nodes as children and operation nodes have only equivalence nodes as children.

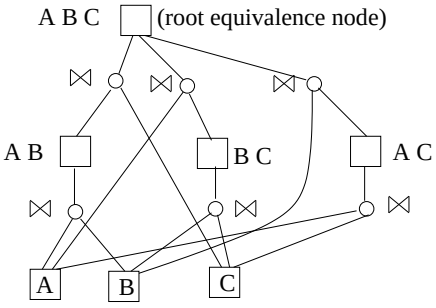


Figure 2.2: Logical Query DAG for A ⋈ B ⋈ C. Commutativity not shown; every join node has another join node with inputs exchanges, below the same equivalence node.

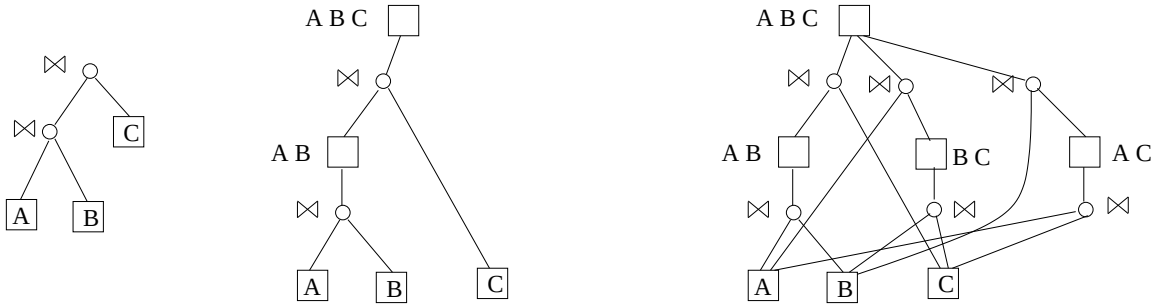
An operation node in the LQDAG corresponds to an algebraic operation, such as join (⋈), select (σ), etc. It represents the expression defined by the operation and its inputs. An equivalence node in the LQDAG represents the equivalence class of logical expressions (rewritings) that generate the same result set, each expression being defined by a child operation node of the equivalence node, and its inputs. An important property of the LQDAG is that there are no two equivalence nodes that correspond to the same result set. The algorithm for expansion of an input query into its LQDAG is presented later in this section.

Figure 2.2 shows a LQDAG for the query A ⋈ B ⋈ C. Note that the DAG has exactly one equivalence node for every subset of $\{A, B, C\}$; the node represents all ways of computing the joins of the relations in that subset. Though the LQDAG in this example represents only a single query A ⋈ B ⋈ C, in general a LQDAG can represent multiple queries in a consolidated manner.

Logical Plan Space Generation

The given query tree is initially represented directly in the LQDAG formulation. For example, the query tree of Figure 2.3(a) for the query $(A \bowtie B \bowtie C)$ is initially represented in the LQDAG formulation, as shown in Figure 2.3(b). The equivalence nodes are shown as boxes, while the operation nodes are shown as circles.

The initial LQDAG is then expanded by applying all possible transformations on every node of the initial LQDAG representing the given query. In the example, suppose the only transformations possible are join associativity and commutativity. Then the plans $(A \bowtie (B \bowtie C))$ and



(a) Initial Query (b) DAG representation of query (c) Expanded DAG after transformations

Figure 2.3: Logical Plan Space Generation for A ⋈ B ⋈ C.

$((A \bowtie C) \bowtie B)$, as well as several plans equivalent to these modulo commutativity can be obtained by transformations on the initial LQDAG of Figure 2.3(b). These are represented in the LQDAG shown in Figure 2.3(c).

Procedure EXPANDDAG, presented in Figure 2.4, expands the input query's LQDAG (as in Figure 2.3(b)) to include all possible logical plans for the query (as in Figure 2.3(c)) *in one pass* – that is, without revisiting any node. The procedure applies the transformations to the nodes in a bottom-up topological manner – that is, all the inputs of a node are fully expanded before the node is expanded.

In the process, new subexpressions are generated. Some of these subexpressions may be equivalent to expressions already in the LQDAG. Further, subexpressions of the query may be equivalent to each other, even if syntactically different. For example, suppose the query contains two subexpressions that are logically equivalent but syntactically different (e.g., $((R \bowtie S) \bowtie T)$, and $(R \bowtie (S \bowtie T))$). Before the second subexpression is expanded, the Query DAG would contain two different equivalence nodes representing the two subexpressions. Whenever it is found that by applying a transformation to an expression in one equivalence node leads to an expression in the other equivalence node (in the above example, after applying join associativity), the two equivalence nodes are deduced as representing the same result and *unified*, that is, replaced by a single equivalence node. The unification of the two equivalence nodes may cause the unification of their ancestors. For example, if the query had the subexpressions $(A \bowtie ((R \bowtie S) \bowtie T))$ and $(A \bowtie (R \bowtie (S \bowtie T)))$, then the unification of the equivalence nodes containing $((R \bowtie S) \bowtie T)$

```
Procedure EXPANDDAG
Input:   eq, the root equivalence node for the initial LQDAG
Output:  The expanded LQDAG
Begin
    for each unexpanded logical operation node op ∈ child(eq)
        for each inpEq ∈ input(op)
            EXPANDDAG(inpEq)
        apply all possible logical transformations to op
            /* may create new equivalence nodes */
        for each resulting logical expression E
            if E ∉ LQDAG
                add E's root operation node to child(eq)
            else if the previous instance E' ∈ eq' where eq' ≢ eq
                unify eq' with eq /* may trigger further unifications */
        mark op as expanded
End
```

Figure 2.4: Algorithm for Logical Query DAG Generation

and $(R \bowtie (S \bowtie T))$ will cause the equivalence nodes containing the above two subexpressions to be unified as well. Thus, the unification has a cascading effect up the LQDAG.

In order to efficiently check the presence of a logical expression in the LQDAG, a hash table is used. Recall that an expression is identified by a logical operator (called the *root operator*) and its input equivalence nodes; for example, the expression $(A \bowtie (B \bowtie C))$ is identified by the root operator $\bowtie$ and its two input equivalence nodes corresponding to A and $(B \bowtie C)$. As such, the has value of an expression is computed as a function of the type-id of the root operator and the id of its input equivalence nodes.

A logical space generation algorithm is called *complete* iff it acts on the initial LQDAG for a query Q and expands it into an LQDAG containing all possible logical plans possible using the given set of transformations. We end this description with a proof of completeness of EXPANDDAG.

Theorem 2.2.1 EXPANDDAG *is complete.*

Proof: Let D_0 denote the initial LQDAG for the query Q. EXPANDDAG acts on D_0 and, by applying the given set of transformations as shown in the pseudocode in Figure 2.4, generates

a final expanded LQDAG D^E. Now, consider any complete algorithm, called COMPLETE, that acts on D_0 and generates the LQDAG D^C. We show that all plans contained in D^C are contained in D^E, thus proving the theorem.

We trace the expansion of D_0 into D^C by COMPLETE as follows:

$$D_0 \xrightarrow{T_1} D_1 \xrightarrow{T_2} D_2 \xrightarrow{T_3} \cdots \xrightarrow{T_n} D_n \equiv D^C$$

where $\xrightarrow{T_i}$ denotes the application of the transformation T_i, transforming a subplan P_i^{old} below the equivalence node e_i in D_{i-1} to a new semantically equivalent plan P_i^{new} below e_i, resulting in D_i.

Let k be such that for all $i < k$, all plans in D_i are contained in D^E, but there exists a plan in D_k, say P, that is not contained in D^E. We show, by contradiction, that such a k cannot exist.

Clearly, the plan P_k^{new}, generated by the application of transformation T_k to the subplan P_k^{old} of e_k during the execution of COMPLETE, is a subplan of P. Let P' denote the plan obtained by replacing the subplan P_k^{new} of P by P_k^{old}; P' is contained in D_{k-1}. But then, by the choice of k above, P' is also contained in D^E. This implies that (a) the subplan P_k^{old} is present below e_k in D^E, and that (b) the subplan P_k^{new} is not present below e_k in D^E – otherwise, P would be present in D^E, which a contradiction due to the choice of k. Next, we use (b) to contradict (a).

When EXPANDDAG visits e_k, it applies all the available transformations, including T_k, to the plans below e_k till no further new plans are generated. Since P_k^{new} is not generated in this exercise, this implies that P_k^{old} is also not present below e_k after it has been expanded as above. Now, because EXPANDDAG visits nodes in a bottom-up topological manner, neither e_k nor any of its descendents are visited later during the expansion. This implies that P_k^{old} is never generated during the execution of EXPANDDAG and is therefore not present below e_k in D^E, leading to a contradiction. □

2.2.3 Physical Plan Space

The plans represented in the Logical Query DAG are only at an abstract, semantic level and, in a sense, provide "templates" that guarantee semantic correctness for the physical plans. For

instance, the logical plan $((A \bowtie B) \bowtie C)$ only specifies the order in which the relations are to be joined. It does not specify the actual execution in terms of the algorithms used for the different operators; for example, a $\bowtie$ can be either a nested-loops join, a merge-join, an indexed nested-loops or a hash-join. As such, the cost for these plans is undefined. Further, the logical plan does not consider the *physical properties* of the results, like sort order on some attribute, into account since results with different physical properties are logically equivalent.

However, the physical properties are important since (a) they affect the execution costs of the algorithms (e.g., the merge join does not need to sort its input if it is already sorted on the appropriate attribute), and (b) they need to be taken into account when specified in the query using the `ORDER BY` clause.

In this section, we give the details of how the physical plan space for a query is generated. Since the physical plan space is very large, a compact representation for the same is needed. We start with a description of the representation used in our implementation, called the *Physical Query DAG*. This representation is a refinement of the Logical Query DAG (LQDAG) representation for the logical plan space described in Section 2.2.2. This is followed by a description of the algorithm to generate the physical plan space in the Physical Query DAG representation given the LQDAG for the input query.

Physical Query DAG Representation

The Physical Query DAG (PQDAG) is a refinement of the LQDAG. Given an equivalence node e in the LQDAG, and a physical property p required on the result of e, there exists an equivalence node in the PQDAG representing the set of physical plans for computing the result of e with *exactly* the physical property p. A physical plan in this set is identified by a child operation node of the equivalence node (called the physical plan's root operation node), and its input equivalence nodes.

For contrast, we hereafter term the equivalence nodes in the LQDAG *logical equivalence nodes* and the equivalence nodes in the PQDAG *physical equivalence nodes*. Similarly, we hereafter term the operation nodes in the physical plans as *physical operation nodes* to disambiguate

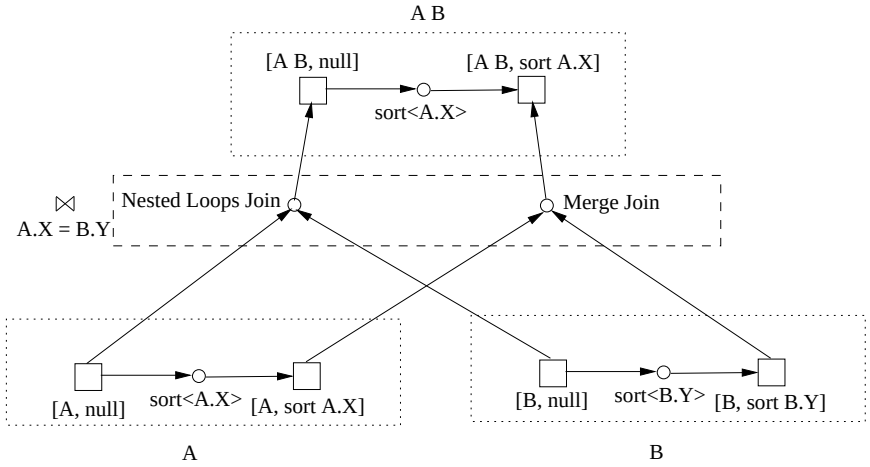


Figure 2.5: Physical Query DAG for $A \bowtie B$

from the logical operation nodes in the logical plans.

The physical operation nodes can either be (a) *algorithms* for computing the logical operations (e.g., the algorithm merge join for the logical operation $\bowtie$), or (b) *enforcers* that enforce the required physical property (e.g., the enforcer sort to enforce the physical property sort-order on an unsorted result).

Figure 2.5 illustrates the PQDAG for $(A \bowtie_{A.X=B.Y} B)$. The dotted boxes are the logical equivalence nodes, labelled alongside with the corresponding relational expressions. The solid boxes within are the corresponding physical equivalence nodes for the respective physical properties stated alongside. The circles denote the physical operators: those within the dotted boxes are the enforcers (sort operations), while those within the dashed box are the algorithms (nested loops join and merge join) corresponding to the logical join operator as shown.

Physical Property Subsumption. Figure 2.5 shows two physical equivalence nodes corresponding to the result $(A \bowtie B)$: one representing plans to compute $(A \bowtie B)$ with no sort order, and the other representing plans to compute $(A \bowtie B)$ with the result sorted on $A.X$. Clearly, any plan that computes $(A \bowtie B)$ sorted on $A.X$ can be used as a plan that computes $(A \bowtie B)$ with no sort order.

In general, we say that the physical equivalence node e *subsumes* the physical equivalence

node e' iff any plan that computes e can be used as a plan that computes e'; this defines a *partial order* on the set of physical equivalence nodes corresponding to a given logical equivalence node.

While finding the best plan for the physical equivalence node e (see Section 2.2.4), the procedure FINDBESTPLAN not only looks at the plans below e, but also at plans below physical quivalence nodes e' that subsume e, and returns the overall cheapest plan. To save on expensive physical property comparisons during the search, the physical equivalence nodes corresponding to the same logical equivalence node are explicitly structured into a DAG representing the partial order.

Furthering the terminology, we say that the physical equivalence node e *strictly subsumes* the physical equivalence node e' iff e subsumes e', but e and e' are distinct. Finally, we say that e *immediately subsumes* e' iff e strictly subsumes e' but there does not exist another distinct node e'' such that e strictly subsumes e'' and e'' strictly subsumes e'.

Physical Plan Space Generation

The PQDAG for the input query is generated from its LQDAG using Procedure PHYSDAGGEN listed in Figure 2.6.

Given a subgoal (e, p) where e is a logical equivalence node in the LQDAG, and p a physical property, PHYSDAGGEN creates a physical equivalence node corresponding to (e, p) if it does not exist already, and then populates it with the physical plans that compute e with the given physical property. Depending on the root operation node being an algorithm or an enforcer, the corresponding physical plan is called an *algorithm plan* or an *enforcer plan* respectively.

An algorithm plan is generated by taking a logical plan for e as a template and instantiating it as follows. The algorithm a that forms the root of the physical plan implements the logical operation o at the root of the logical plan, generating the result with the physical property p. The inputs of a are the physical equivalence nodes returned by recursive invocations of PHYSDAGGEN on the respective input equivalence nodes of o with physical properties as required by a.

For each enforcer f that enforces the physical property p, an enforcer plan is generated with f as the root operation node. The input of f is the physical equivalence node returned by a

```
Procedure PHYSDAGGEN
Input:   e, a equivalence node in the Logical Query DAG,
         p, the desired physical property
Output:  n_p, the equivalence node in the Physical Query DAG for e with physical property p,
         populated with the corresponding plans
Begin
    if an equivalence node n_p exists for e with property p
        return it
    create an equivalence node n_p
    for every operation node o below e
        for every algorithm a for o that guarantees property p
            create an algorithm node o_a under n_p.
            for each input i of e
                let o_i be the ith input
                let p_i the physical property required from input i by algorithm a
                set input i of o_a = PHYSDAGGEN(o_i, p_i)
    for every enforcer f that generates property p
        create an enforcer node o_f under n_p
        set the input of o_f = PHYSDAGGEN(e, null)
            /* null denotes "no physical property requirement" */
    return n_p
End
```

Figure 2.6: Algorithm for Physical Query DAG Generation

recursive invocation of PHYSDAGGEN on the same equivalence node with no required physical property.

In the PQDAG of Figure 2.5, the logical equivalence node $(A \bowtie_{A.X=B.Y} B)$ is refined into the two physical equivalence nodes – one for no physical property and the other for sort order on $A.X$. The logical join instantiated as nested loops join forms the root of the algorithm plan for the former. For the latter, the same logical join instantiated as merge-join forms the root of the algorithm plan while the sort operator forms the root of the enforcer plan. From the PQDAG shown, it is apparent that the nested loops join requires no physical property on its input relations A and B, while the merge join requires its input relations A and B sorted on $A.X$ and $B.X$ respectively.

The entire PQDAG is generated by invoking PHYSDAGGEN on the root of the input query's LQDAG, with the desired physical properties of the query.

2.2.4 The Search Algorithm

Each plan in the PQDAG has a cost computed recursively by adding the local cost of the physical operator at the root to the cost of the subplans of each of its inputs.[1] This section describes how Volcano determines the plan with the least cost from the space of plans represented in the Physical Query DAG generated as above. The search algorithm is based on dynamic programming – specifically, it uses the technique of *memoization* wherein the best plans for the nodes are saved after the first computation, and reused when needed later.

We assume that the set of enforcers being considered are such that in any best plan, no two enforcers can be cascaded together; hence the plans with enforcer cascades need not be considered while searching for the best plan. This may not be true always. For example, the index enforcer, that takes a sorted input and builds a clustered index on the same, requires that its input be sorted on the relevant attribute, and the best plan for the input may be an enforcer plan with the sort operator as the root. We handle this by introducing a *composite* enforcer for each possible cascade – in the above case, the sort-index cascade is handled by introducing a

[1]The formulae used to estimate the operator costs appear in Appendix C.

```
Procedure FINDBESTPLAN
Input:   e, a physical equivalence node in the PQDAG
Output:  The best plan for e
Begin
    bestEnfPlan = FINDBESTENFPLAN(e)
    bestAlgPlan = FINDBESTALGPLAN(e)
    return the cheaper of bestEnfPlan and bestAlgPlan
End

Procedure FINDBESTENFPLAN
Input:   e, a physical equivalence node in the PQDAG
Output:  The best enforcer plan for e
Begin
    if best enforcer plan for e is present /* memoized */
        return best enforcer plan for e
    bestEnfPlan = dummy plan with cost +∞
    for each enforcer child op of e
        planCost = cost of op
        for each input equivalence node e_i of op
            inpBestPlan = FINDBESTALGPLAN(e_i)
            planCost = planCost + cost of inpBestPlan
        if planCost < cost of bestEnfPlan
            bestEnfPlan = plan rooted at op
    memoize bestEnfPlan as best enforcer plan for e
    return bestEnfPlan
End

Procedure FINDBESTALGPLAN
Input:   e, a physical equivalence node in the PQDAG
Output:  The best algorithm plan for e
Begin
    if best algorithm plan for e is present /* memoized */
        return best algorithm plan for e
    bestAlgPlan = dummy plan with cost +∞
    for each algorithm child op of e
        planCost = cost of op
        for each input equivalence node e_i of op
            inpBestPlan = FINDBESTPLAN(e_i)
            planCost = planCost + cost of inpBestPlan
        if planCost < cost of bestAlgPlan
            bestAlgPlan = plan rooted at op
    for each equivalence node e' that immediately subsumes e
        subsBestAlgPlan = FINDBESTALGPLAN(e')
        if cost of subsBestAlgPlan < cost of bestAlgPlan
            bestAlgPlan = subsBestAlgPlan
    memoize bestAlgPlan as best algorithm plan for e
    return bestAlgPlan
End
```

Figure 2.7: The Search Algorithm

sort-cum-index enforcer. The space of enforcer plans generated using the resulting enforcer set contains the best enforcer plan.

Procedure FINDBESTPLAN, shown in Figure 2.7, finds the best plan for an equivalence node e in the PQDAG. FINDBESTPLAN calls the procedures FINDBESTENFPLAN and FINDBESTALGPLAN that respectively find the best enforcer plan and algorithm plan for e, and returns the cheaper of the two plans.

FINDBESTENFPLAN looks at each enforcer child of e, and constructs the best plan for that enforcer by taking the best algorithm plan for its input physical equivalence node. The cheapest of these plans is the best enforcer plan for e.

FINDBESTALGPLAN looks at each algorithm child of e, and builds the best plan for that algorithm by taking the best plan for each of its input physical equivalence nodes, determined by recursive invocations of FINDBESTPLAN. Further, it looks at the best plan for each immediately subsuming node (see Section 2.2.3), determined recursively. The cheapest of all these plans is the best algorithm plan for e.

Observe that subsuming physical equivalence nodes are considered only while searching for the best algorithm plan (in FINDBESTALGPLAN) and not while searching for the best enforcer plan (in FINDBESTENFPLAN). This is because an enforcer plan for the subsuming physical equivalence node has a cost at least as much as the best enforcer plan for the subsumed physical equivalence node.[2]

Branch-and-Bound Pruning. Branch-and-bound pruning is implemented by passing an extra parameter, the *cost limit*, which specifies an upper limit on the cost of the plans to be considered. The cost limit for the root equivalence node is initially infinity. When a plan for a physical equivalence node e with cost less than the current cost limit is found, its cost becomes the new cost limit for future search of the best plan for e.

The cost limit is propagated down the DAG during the search and helps prune the search space as follows. Consider the invocation of FINDBESTPLAN on the physical equivalence node e. In the call to FINDBESTENFPLAN, the cost limit for the input of the enforcer op is the cost

[2]This is assuming that, for example, cost of sorting A on $A.X$ is at most that of sorting it on $\langle A.X, A.Y \rangle$

limit for e minus the local cost of op. Similarly, in FINDBESTALGPLAN invoked on e, when invoking FINDBESTPLAN on the ith input of an algorithm node child op of e, the cost limit for the plan for the ith input is the cost limit for e minus the sum of the costs of best plans for earlier inputs to op as well as the local cost of computing op. The recursive plan generation occurs only till the cost limit is positive; when the cost limit becomes non-positive, the current plan is pruned. If all the plans for e are pruned for the given cost limit, then the cost limit is a lower bound on the best plan for e – this lower bound is used to prune later invocations on e with higher cost limits.

Branch-and-bound pruning is not shown in the pseudocode for FINDBESTPLAN in Figure 2.7, for sake of simplicity.

2.2.5 Differences from the Original Volcano Optimizer

In this section, we point out the major differences between our optimizer and the Volcano optimizer as described in [23].

Separation of Logical/Physical Plan Space Generation and Search

Our approach in this chapter has been to assume that the three steps of (1) LQDAG generation, (2) PQDAG generation, and (3) search for the best plan are executed one after another, independently. In other words, the optimization task goes "breadth-first" on the graph of Figure 2.1 – given the input query Q, first all its rewritings $Q_1, \ldots, Q_m$ are generated, then all its execution plans $P_{11}, \ldots, P_{mn}$ are generated, and finally the best execution plan P^* is identified and returned.

This may not be the case in reality, where these three steps may interleave. For example, on the other extreme, the optimizer may choose to go "depth-first" on the graph of Figure 2.1. First Q_1 is generated, then its corresponding execution plans $P_{11}, \ldots, P_{1k}$, are generated and the best plan so far identified. Then, the next rewriting Q_2 is generated, folowed by its corresponding execution plans and the best plan so far is updated, if a better plan is seen. This repeats for all the successive rewritings upto Q_m, and finally the overall best plan is returned. This is essentially the Volcano algorithm, as described in [23]. This approach may be advantageous when the complete

space of plans is too big to fit in memory, since here the rewritings and the plans that have already been found to be suboptimal can be discarded before the end of the algorithm.

Unification of Equivalent Subexpressions

The original Volcano algorithm does not generate the unified LQDAG as explained in Section 2.2.2. Instead, the generated LQDAG may have multiple logical equivalence nodes representing the same logical expressions.

For example, consider the query $((A \bowtie B \bowtie C) \cup (B \bowtie C \bowtie D))$. The Volcano optimizer does not consider the two occurences of B as refering to the same relation. Similarly for the two occurences of C. Instead each occurrence of B or C is considered a distinct relation; effectively, the query is interpreted as $((A \bowtie B \bowtie C) \cup (B' \bowtie C' \bowtie D))$ where B' and C' are clones of B and C respectively. This does not alter the search space, since during execution the two accesses of B (or C) are going to be independent, anyway. However, by doing so, it fails to recognise that the two subexpressions expressions $(B \bowtie C)$ and $(B' \bowtie C')$ are identical, and therefore optimizes them independently.

In our version of Volcano, since the equivalent subexpressions are unified (see Section 2.2.2), the subexpression is going to be optimized only once and the best plan reused for both of its occurrences. In general, the common subexpression may be rather complex, and unification may reduce the optimization effort significantly.

Separation of the Enforcer and Algorithm Plan Spaces

Our version of Volcano memoizes the best algorithm plan as well as the best enforcer plan for each physical equivalence node. On the other hand, Volcano stores only the overall best plan.

While searching for the best plan for, say, $(A \bowtie_{A.X=B.Y} B)$ sorted on $A.X$, Volcano explores the enforcer plan with the sort operation on $A.X$ as the root and the equivalence node for unsorted result as input. In order to determine the best plan for this input node, in the naive case, it visits the equivalence nodes that subsume the same. In particular, it explores the equivalence node for the sort order $A.X$ as well, landing back where it had started and thus gets into an infinite

recursion. Volcano tries to avoid this by passing down an extra parameter, the *excluding physical property*, to the search function. In the above example, the excluding physical property is "sort order on $A.X$" and helps the recursive call to determine the best plan for the unsorted result figure that the equivalence node with sort order on $A.X$ should not be explored while looking for the best plan.

However, this approach has its own problems. The best plan thus found for the equivalence node with no sort order is subject to the exclusion of the said physical property and may not be its overall best plan; in particular, the merge-join plan for the result that is present below the equivalence code for sort order $A.X$ may be the overall best plan for the unsorted result, but has not been considered above. Thus, at each equivalence node, the optimizer needs to memoize the best plan for each excluded physical property apart from the overall best plan — a significant amount of book-keeping.

Our version obviates the above problem, as discussed earlier in Section 2.2.4 by observing that one need only consider algorithm plans as input to an enforcer while looking for the best enforcer plan. While searching for the best plan for $(A \bowtie B)$ sorted on $A.X$, the enforcer plan considered only cosists of the sort operation over the best algorithm plan for the unsorted result. In general, it can be seen that in Figure 2.7, neither of FINDBESTPLAN, FINDBESTENFPLAN and FINDBESTALGPLAN are ever invoked more than once on the same equivalence node, thus proving that the recursion always terminates.

2.3 Summary

In this chapter, we first gave a brief overview of the issues in traditional query optimization, and pointed out the important research and development work in this area. We then gave a detailed description of the design of our version of the Volcano query optimizer, which provides the basic framework for the work presented in this thesis. Later chapters of this thesis modify this basic optimizer, enabling it to perform multi-query optimization, query result cache management and materialized view selection and materialization respectively.

For sake of simplicity, the later chapters restrict to the logical plan space. The Query DAG

refered hereafter will refer to the Logical Query DAG, unless explicitly stated otherwise. However, the descriptions therein can be easily extended in terms of the physical plan space.

Chapter 3

Multi-Query Optimization

This chapter[1] addresses the problem of optimizing a set of queries exploiting the presence of common sub-expressions among the queries; this problem is referred to as *multi-query optimization*. Common subexpressions are possible even *within* a single query; the techniques we develop deal with such intra-query common subexpressions as well.

Traditional query optimizers are not appropriate for optimizing queries with common sub expressions, since they make locally optimal choices, and may miss globally optimal plans as the following example demonstrates.

Example 3.0.1 Let Q_1 and Q_2 be two queries whose locally optimal plans (i.e., individual best plans) are $(R \bowtie S) \bowtie P$ and $(R \bowtie T) \bowtie S$ respectively. The best plans for Q_1 and Q_2 do not have any common sub-expressions, hence they cannot share However, if we choose the alternative plan $(R \bowtie S) \bowtie T$ (which may not be locally optimal) for Q_2, then, it is clear that $R \bowtie S$ is a common sub-expression and can be computed once and used in both queries. This alternative with sharing of $R \bowtie S$ may be the globally optimal choice.

On the other hand, blindly using a common sub-expression may not always lead to a globally optimal strategy. For example, there may be cases where the cost of joining the expression $R \bowtie S$ with T is very large compared to the cost of the plan $(R \bowtie T) \bowtie S$; in such cases it may make

[1] Joint work with S. Seshadri, S. Sudarshan and Siddhesh Bhobe. Parts of this chapter appeared in SIGMOD 2000 [47]

no sense to reuse $R \bowtie S$ even if it were available. □

Example 3.0.1 illustrates that the job of multi-query optimizer, over and above that of ordinary query optimizer, is to (i) recognize the possibilities of shared computation, and (ii) modify the optimizer search strategy to explicitly account for shared computation and find a globally optimal plan.

While there has been work on multi-query optimization in the past ([54, 56, 53, 13, 38]), prior work has concentrated primarily on exhaustive algorithms. Other work has concentrated on finding common subexpressions as a post-phase to query optimization [18, 59], but this gives limited scope for cost improvement. The search space for multi-query optimization is doubly exponential in the size of the queries, and exhaustive strategies are therefore impractical; as a result, multi-query optimization was hitherto considered too expensive to be useful. We show how to make multi-query optimization *practical*, by developing novel heuristic algorithms, and presenting a performance study that demonstrates their practical benefits.

We have decomposed our approach into two distinct tasks: (i) recognize possibilities of shared computation (thus essentially setting up the search space by identifying common subexpressions), and (ii) modify the optimizer search strategy to explicitly account for shared computation and find a globally optimal plan. Both of the above tasks are important and crucial for a multi-query optimizer but are *orthogonal*. In other words, the details of the search strategy do not depend on how aggressively we identify common sub-expressions (of course, the efficacy of the approach does).

The rest of this chapter is structured as follows: We describe how to set up the search space for multi-query optimization in Section 3.1. Next, we present three heuristics for finding the globally optimal plan. Two of the heuristics we present, Volcano-SH and Volcano-RU are lightweight modifications of the Volcano optimization algorithm, and are described in Section 3.2. The third heuristic is a greedy strategy which iteratively picks the subexpression that gives the maximum benefit (reduction in cost) if it is materialized and reused; this strategy is covered in Section 3.3. Our extensions to create indexes on intermediate relations and nested queries are discussed in Sections 3.5. We describe the results of our performance study in Section 3.6. Section 3.7

discusses related work. We summarize the chapter in Section 3.8.

3.1 Setting Up The Search Space

As we mentioned earlier, the job of a multi-query optimizer is to (i) recognize possibilities of shared computation (thus essentially setting up the search space by identifying common sub-expressions) and (ii) modify the optimizer search strategy to explicitly account for shared computation and find a globally optimal plan. Both of the above tasks are important and crucial for a multi-query optimizer but are *orthogonal*. In other words, the details of the search strategy do not depend on how aggressively we identify common sub-expressions (of course, the efficacy of the strategy does). We emphasize the search strategy component in this thesis.

To apply multi-query optimization to a batch of queries, the queries are represented together in a single Query DAG, sharing subexpressions (ref. Section 2.2.2). To make the Query DAG rooted, a pseudo operation node is created, which does nothing, but has the root equivalence nodes of all the queries as its inputs. We extend the Query DAG generation algorithm of Section 2.2.2 to aid multi-query optimization by introducing *subsumption derivations* which identify and add more CSEs into the Query DAG, thus increasing the potential of sharing within the plans.

For example, suppose two subexpressions $e1$: $\sigma_{A<5}(E)$ and $e2$: $\sigma_{A<10}(E)$ appear in the query. The result of $e1$ can be obtained from the result of $e2$ by an additional selection, i.e., $\sigma_{A<5}(E) \equiv \sigma_{A<5}(\sigma_{A<10}(E))$. To represent this possibility we add an extra operation node $\sigma_{A<5}$ in the Query DAG, between $e1$ and $e2$. Similarly, given $e3$: $\sigma_{A=5}(E)$ and $e4$: $\sigma_{A=10}(E)$, we can introduce a new equivalence node $e5$: $\sigma_{A=5 \vee A=10}(E)$ and add new derivations of $e3$ and $e4$ from $e5$. The new node represents the sharing of accesses between the two selection. In general, given a number of selections on an expression E, we create a single new node representing the disjunction of all the selection conditions.

Similar derivations also help with aggregations. For example, if we have $e6$: ${}_{dno}\mathcal{G}_{sum(Sal)}(E)$ and $e7$: ${}_{age}\mathcal{G}_{sum(Sal)}(E)$, we can introduce a new equivalence node $e8$: ${}_{dno,age}\mathcal{G}_{sum(Sal)}(E)$ and add derivations of $e6$ and $e7$ from equivalence node $e8$ by further groupbys on dno and age.

The idea of applying an operation (such as $\sigma_{A<5}$ on one subexpression to generate another

has been proposed earlier [45, 54, 59]. Integrating such options into the Query DAG, as we do, clearly separates the space of alternative plans (represented by the Query DAG) from the optimization algorithms. Thereby, it simplifies our optimization algorithms, allowing them to avoid dealing explicitly with such derivations.

Physical Properties. Our search algorithms can be easily understood on the Logical Query DAG representation (without physical properties), although they actually work on Physical Query DAGs (ref. Section 2.2.3). For brevity, therefore, we do not explicitly consider physical properties further.

3.2 Reuse Based Multi-Query Optimization Algorithms

In this section we study a class of multi-query optimization algorithms based on reusing results computed for other parts of the query. We present these as extensions of the Volcano optimization algorithm. Before we describe the extensions, in Section 3.2.1, we outline how to extend the basic Volcano optimization algorithm to find best plans given some nodes in the DAG are materialized. Sections 3.2.2 and 3.2.3 then present two of our heuristic algorithms, Volcano-SH and Volcano-RU.

3.2.1 Optimization in Presence of Materialized Views

We now consider how to extend Volcano to find best plans, given that (expressions corresponding to) some equivalence nodes in the DAG are materialized. Let $reusecost(m)$ denote the cost of reusing the materialized result of m, and let M denote the set of materialized nodes.

The only change from the algorithm presented in Chapter 2 is as follows. When computing the cost of a operation node o, if an input equivalence node e is materialized (i.e., in M), use the minimum of $reusecost(e)$ and $cost(e)$ when computing $cost(o)$. Thus, we use the following expression instead:

$$cost(o) = \text{cost of executing}(o) + \sum_{e_i \in children(o)} C(e_i)$$

where

$$C(e_i) = \begin{cases} cost(e_i) & \text{if } e_i \notin M \\ \min(cost(e_i), reusecost(e_i)) & \text{if } e_i \in M \end{cases}$$

3.2.2 The Volcano-SH Algorithm

In our first strategy, which we call Volcano-SH, the expanded DAG is first optimized using the basic Volcano optimization algorithm. The best plan computed for the virtual root is the combination of the Volcano best plans for each individual query. The best plans produced by the Volcano optimization algorithm may have common subexpressions. Thus the consolidated best plan for the root of the DAG may contain nodes with more than one parent, and is thus a DAG-structured plan.[2] The Volcano-SH algorithm works on the above consolidated best plan, and decides in a cost based manner which of the nodes to materialize and share.

Since materialization of a node involves storing the result to the disk, and we assume pipelined execution of operators, it may be possible for recomputation of a node to be cheaper than the cost of materializing and reusing the node. In fact, in our experiments in Section 3.6, there were quite a few occasions when it was cheaper to recompute an expression.

Let us consider first a naive (and incomplete) solution. Consider an equivalence node e. Let $cost(e)$ denote the computation cost of node e. Let $numuses(e)$ denote the number of times node e is used in course of execution of the plan. Let $matcost(e)$ denote the cost of materializing node e. As before, $reusecost(e)$ denote the cost of reusing the materialized result of e. Then, we decide to materialize e if

$$cost(e) + matcost(e) + reusecost(e) \times (numuses(e) - 1) < numuses(e) \times cost(e)$$

The left hand side of this inequality gives the cost of materializing the result when first computed, and using the materialized result thereafter; the right hand side gives the cost of the alternative wherein the result is not materialized but recomputed on every use. The above test can be simplified to

$$matcost(e)/(numuses(e) - 1) + reusecost(e) < cost(e) \tag{3.1}$$

[2]The ordering of queries does not affect the above plan.

The problem with the above solution is that $numuses(e)$ and $cost(e)$ both depend on what other nodes have been materialized, For instance, suppose node e_1 is used twice in computing node e_2, and node e_2 is used twice in computing node e_3. Now, if no node is materialized, e_1 is used four times in computing e_3. If e_2 is materialized, e_1 gets used twice in computing e_2, and e_2 gets computed only once. Thus, materializing e_2 can reduce both $numuses(e_1)$ and $cost(e_3)$.

In general, $numuses(e)$ depends on which ancestors of e in the Volcano best plan are materialized, and $cost(e)$ depends on which descendants have been materialized. Specifically, $numuses(e)$ can be computed recursively based on the number of uses of the parents of e: $numuses(root) = 1$, while for all other nodes, $numuses(e) = \sum_{p \in parents(e)} U(p)$, where $U(p) = numuses(p)$ if p is not materialized, and $= 1$ if p is materialized. Thus, computing $numuses$ requires us to know the materialization status of parents. On the other hand, as we have seen earlier, $cost(e)$ depends on what descendants have been materialized.

A naive exhaustive strategy to decide what nodes in the Volcano best plan to materialize is to consider each subset of the nodes in the best plan, and compute the cost of the best plan given that all nodes in this subset are materialized at their first computation; the subset giving the minimum cost is selected for actual materialization. Unfortunately, this strategy is exponential in the number of nodes in the Volcano best plan, and therefore is very expensive; we require cheaper heuristics.

To avoid enumerating all sets as above, the Volcano-SH algorithm, which is shown in Figure 3.1, traverses the tree bottom-up. As each equivalence node e is encountered in the traversal, Volcano-SH decides whether or not to materialize e. When making a materialization decision for a node, the materialization decisions for all descendants ia already known. When Volcano-SH is examining a node e, let M denote the set of descendants of e that have been chosen to be materialized. Based on this, we can compute $cost(e)$ for a node e, as described in Section 3.2.1.

To make a materialization decision for a node e, we also need to know $numuses(e)$. Unfortunately, $numuses(e)$ depends on the materialization status of its parents, which is not fixed yet. To solve this problem, the Volcano-SH algorithm uses an underestimate $numuses^{-}(e)$ of number of uses of e. Such an underestimate can be obtained by simply counting the number of ancestors of e in the Volcano best plan. We use this underestimate in our cost formulae, to make

Procedure VOLCANO-SH
Input: Consolidated Volcano best plan P for virtual root of DAG
Output: Set of nodes to materialize M, and the corresponding best plan P
Global variable: M, the set of nodes chosen to be materialized
Begin
    $M = \phi$
    Perform prepass on P to introduce subsumption derivations
    Let C_{root} = COMPUTEMATSET($root$)
    Set $C_{root} = C_{root} + \sum_{d \in M}(cost(d) + matcost(d))$
    Undo all subsumption derivations on P
        where the subsumption node is not chosen to be materialized.
    return (M,P)
End

Procedure COMPUTEMATSET
Input: e, equivalence node
Output: Cost of computing e
Global variable: M, the set of nodes chosen to be materialized
Begin
    If $cost(e)$ is already memoized, return $cost(e)$
    Let operator o_e be the child of e in P
    For each input equivalence node e_i of o_e
        Let C_i= COMPUTEMATSET(e_i) // returns computation cost of e_i
        If e_i is materialized, let $C_i = reusecost(e_i)$
    Compute $cost(e)$ = cost of operation o_e + $\sum_i C_i$
    If ($matcost(e)/(numuses^-(e) - 1) + reusecost(e) < cost(e)$)
        If e is *not* introduced by a subsumption derivation
            add e to M // Decide to materialize e
        else if $cost(e) + matcost(e) + reusecost(e) * (numuses^-(e) - 1)$ is less than
                savings to parents of e due to introducing materialized e
            add e to M // Decide to materialize e
    Memoize and return $cost(e)$
End

Figure 3.1: The Volcano-SH Algorithm

a conservative decision on materialization.

Based on the above, Volcano-SH makes the decision on materialization as follows: node e is materialized if

$$matcost(e)/(numuses^{-}(e) - 1) + reusecost(e) < cost(e) \tag{3.2}$$

Note that here we use the lower bound $numuses^{-}(e)$ in place of $numuses(e)$. Using the lower bound guarantees that if we decide to materialize a node, materialization will result in cost savings.

The final step of Volcano-SH is to factor in the cost of computing and materializing all nodes that were chosen to be materialized. Thus, to the cost of the pseudoroot computed as above, we add $\sum_{m \in M}(cost(m) + matcost(m))$, where M is the set of nodes chosen to be materialized.

Let us now return to the first step of Volcano-SH. Note that the basic Volcano optimization algorithm will not exploit subsumption derivations, such as deriving $\sigma_{A<5}(E)$ by using $\sigma_{A<5}(\sigma_{A<10}(E))$, since the cost of the latter will be more than the former, and thus will not be locally optimal.

To consider such plans, we perform a pre-pass, checking for subsumption amongst nodes in the plan produced by the basic Volcano optimization algorithm. If a subsumption derivation is applicable, we replace the original derivation by the subsumption derivation. At the end of Volcano-SH, if the shared subexpression is not chosen to be materialized, we replace the derivation by the original expressions. In the above example, in the prepass we replace $\sigma_{A<5}(E)$ by $\sigma_{A<5}(\sigma_{A<10}(E))$. If $\sigma_{A<10}(E)$ is not materialized, we replace $\sigma_{A<5}(\sigma_{A<10}(E))$ by $\sigma_{A<5}(E)$.

The algorithm of [59] also finds best plans and then chooses which shared subexpressions to materialize. Unlike Volcano-SH, it does not factor earlier materialization choices into the cost of computation.

3.2.3 The Volcano-RU Algorithm

The intuition behind Volcano-RU is as follows. Consider Q_1 and Q_2 from Example 3.0.1. With the best plans as shown in the example, namely $(R \bowtie S) \bowtie P$ and $(R \bowtie T) \bowtie S$, no sharing is

```
Procedure VOLCANO-RU
Input:    Expanded DAG on queries Q_1, ..., Q_k (including subsumption derivations)
Output:   Set of nodes to materialize M, and the corresponding best plan P
Begin
    N = φ // Set of potentially materialized nodes
    For each equivalence node e, Set count[e] = 0
    For i = 1 to k
        Compute P_i, the best plan for Q_i, using Volcano, assuming nodes in N are materialized
        For every equivalence node in P_i
            set count[e] = count[e] + 1
            If (cost(e) + matcost(e) + count[e] * reusecost(e) < (count[e] + 1) * cost(e))
                    // Worth materializing if used once more
                add e to set N
    Combine P_1, ..., P_k to get a single DAG-structured plan P
    (M,P) = VOLCANO-SH(P)      // Volcano-SH makes final materialization decision
End
```

Figure 3.2: The Volcano-RU Algorithm

possible with Volcano-SH. However, when optimizing Q_2, if we realize that $R \bowtie S$ is already used in in the best plan for Q_1 and can be shared, the choice of plan $(R \bowtie S) \bowtie T$ may be found to be the best for Q_2.

The intuition behind the Volcano-RU algorithm is therefore as follows. Given a batch of queries, Volcano-RU optimizes them in sequence, keeping track of what plans have already been chosen for earlier queries, and considering the possibility of reusing parts of the plans. The resultant plan depends on the ordering chosen for the queries; we return to this issue after discussing the Volcano-RU algorithm.

The pseudocode for the Volcano-RU algorithm is shown in Figure 3.2. Let $Q_1, \ldots, Q_n$ be the queries to be optimized together (and thus under the same pseudo-root of the DAG). The Volcano-RU algorithm optimizes them in the sequence $Q_1, \ldots, Q_n$. After optimizing Q_i, we note equivalence nodes in the DAG that are part of the best plan P_i for Q_i as candidates for potential reuse later. We maintain counts of number of uses of these nodes. We also check if each node is worth materializing, if it is used one more time. If so, we add the node to N, and when optimizing the next query, we will assume it to be available materialized.

Thus, in our example earlier in this section, after finding the best plan for the first query, we

check if $R \bowtie S$ is worth materializing if it is used once more. If so we add it to N, and assume it to be materialized when optimizing the second query.

After optimizing all the individual queries, the second phase of Volcano-RU executes Volcano-SH on the overall best plan found as above to further detect and exploit common subexpressions. This step is essential since the earlier phase of Volcano-RU does not consider the possibility of sharing common subexpressions within a single query – equivalence nodes are added to N only after optimizing an entire query. Adding a node to N in our algorithm does not imply it will get reused and therefore materialized. Instead Volcano-SH makes the final decision on what nodes to materialize. The difference from directly applying Volcano-SH to the result of Volcano optimization is that the plan P that is given to Volcano-SH has been chosen taking sharing of parts of earlier queries into account, unlike the Volcano plan.

A related implementation issue is in caching of best plans in the DAG. When optimizing Q_i we cache best plans in nodes of the DAG that are descendants of Q_i. When optimizing a later query Q_j, if we find a node that is not in P_i (the plan chosen for query Q_i) for some $i < j$, we must recompute the best plan for the node; for, the set of nodes M may have changed, leading to a different best plan. Therefore we note with each cached best plan which query was being optimized when the plan was computed; we recompute the plan as required above.

Note that the result of Volcano-RU depends on the order in which queries are considered. In our implementation we consider the queries in the order in which they are given, as well as in the reverse of that order, and pick the cheaper one of the two resultant plans. Note that the DAG is still constructed only once, so the extra cost of considering the two orders is relatively quite small. Considering further (possibly random) orderings is possible, but the optimization time would increase further.

3.3 The Greedy Algorithm

In this section, we present the greedy algorithm, which provides an alternative approach to the algorithms of the previous section. Our major contribution here lies in how to *efficiently implement* the greedy algorithm, and we shall concentrate on this aspect.

In this section, we present an algorithm with a different optimization philosophy. The algorithm picks a set of nodes S to be materialized and then finds the optimal plan given that nodes in S are materialized. This is then repeated on different sets of nodes S to find the best (or a good) set of nodes to be materialized.

Before coming to the greedy algorithm, we present some definitions, and an exhaustive algorithm. As before, we shall assume there is a virtual root node for the DAG; this node has as input a "no-op" logical operator whose inputs are the queries $Q_1 \ldots Q_k$. Let Q denote this virtual root node.

For a set of nodes S, let $bestcost(Q, S)$ denote the cost of the optimal plan for Q given that nodes in S are to be materialized (this cost includes the cost of computing and materializing nodes in S). As described in the Volcano-SH algorithm, the basic Volcano optimization algorithm with an appropriate definition of the cost for nodes in S can be used to find $bestcost(Q, S)$.

To motivate our greedy heuristic, we first describe a simple exhaustive algorithm. The exhaustive algorithm, iterates over each subset S of the set of nodes in the DAG, and chooses the subset S_{opt} with the minimum value for $bestcost(Q, S)$. Therefore, $bestcost(Q, S_{opt})$ is the cost of the globally optimal plan for Q.

It is easy to see that the exhaustive algorithm is doubly exponential in the size of the initial query DAG and is therefore impractical.

In Figure 3.3 we outline a greedy heuristic that attempts to approximate S_{opt} by constructing it one node at a time. The algorithm iteratively picks nodes to materialize. At each iteration, the node x that gives the maximum reduction in the cost if it is materialized is chosen to be added to X.

The greedy algorithm as described above can be very expensive due to the large number of nodes in the set Y and the large number of times the function $bestcost$ is called. We now present three important and novel optimizations to the greedy algorithm which make it efficient and practical.

1. The first optimization is based on the observation that the nodes materialized in the globally optimal plan are obviously a subset of the ones that are shared in some plan for the query.

```
Procedure GREEDY
Input:    Expanded DAG for the consolidated input query Q
Output:   Set of nodes to materialize and the corresponding best plan
Begin
      X = φ
      Y = set of equivalence nodes in the DAG
      while (Y ≠ φ)
L1:       Pick the node x ∈ Y with the smallest value for bestcost(Q, {x} ∪ X)
          if (bestcost(Q, {x} ∪ X) < bestcost(Q, X) )
                Y = Y - x;   X = X ∪ {x}
          else Y = φ /* benefit < 0, so break out of loop */
      return X
End
```

Figure 3.3: The Greedy Algorithm

Therefore, it is sufficient to initialize Y in Figure 3.3, with nodes that are shared in some plan for the query. We call such nodes *sharable nodes*. For instance, in the expanded DAG for Q_1 and Q_2 corresponding to Example 3.0.1, $R \bowtie S$ is sharable while $R \bowtie T$ is not. We present an efficient algorithm for finding sharable nodes in Section 3.3.1.

2. The second optimization is based on the observation that there are many calls to $bestcost$ at line L1 of Figure 3.3, with different parameters. A simple option is to process each call to $bestcost$ independent of other calls. However, observe that the symmetric difference[3] in the sets passed as parameters to successive calls to $bestcost$ is very small – sucessive calls take parameters of the form $bestcost(Q, \{x\} \cup X)$, where only x varies. It makes sense for a call to leverage the work done by a previous call. We describe a novel incremental cost update algorithm, in Section 3.3.2, that maintains the state of the optimization across calls to $bestcost$, and incrementally computes a new state from the old state.

3. The third optimization, which we call the monotonicity heuristic, avoids having to invoke $bestcost(Q, \{x\} \cup X)$, for every $x \in Y$, in line L1 of Figure 3.3. We describe this optimization in detail in Section 3.3.3.

[3]The symmetric difference of two sets S_1 and S_2 consists of elements that are in one of the two but not both; formally the symmetric difference of sets S_1 and S_2 is $(S_1 - S_2) \cup (S_2 - S_1)$, where $-$ denotes set difference.

3.3.1 Sharability

In this subsection, we outline how to detect whether an equivalence node can be shared in some plan. The plan tree of a plan is the tree obtained from the DAG structured plan, by replicating all shared nodes of the plan, to completely eliminate sharing. The degree of sharing of a logical equivalence node in an evaluation plan P is the number of times it occurs in the plan tree of P. The *degree of sharing* of a logical equivalence node in an expanded DAG is the maximum of the degree of sharing of the equivalence node amongst all evaluation plans represented by the DAG. A logical equivalence node is *sharable* if its degree of sharing in the expanded DAG is greater than one.

We now present a simple algorithm to compute the degree of sharing of each node and thereby detect whether a node is shared. A sub–DAG of a node x consists of the nodes below x along with the edges between these nodes that are in the original DAG. For each node x of the DAG, and every equivalence node z in the sub-DAG rooted at x, let $E[x][z]$ represent the degree of sharing of z in the sub–DAG rooted at x. Clearly for all equivalence nodes x, $E[x][x]$ is 1. For a given node x, all other $E[x][\_]$ values can be computed given the values $E[y][\_]$ for all children y of x as follows.

If x is an operation node

$$E[x][z] = Sum\{E[y][z] \mid y \in children(x)\}$$

and if x is an equivalence node,

$$E[x][z] = Max\{E[y][z] \mid y \in children(x)\}$$

The degree of sharing of an equivalence node z in the overall DAG is given by $E[r][z]$, where r is the root of the DAG.

Space is minimized in the above by computing $E[x][z]$ for one z at a time, discarding all but $E[r][z]$ at the end of computation for one z value.

In a reasonable implementation of the above algorithm, the time complexity of computing the row $E[x]$ is proportional to (a) the number of non-zero entries in $E[x]$ (say n_x), and (b) the number of children of x (say e_x). Thus, the overall complexity of the algorithm is proportional to $\sum_x n_x e_x$. Since n_x is (very conservatively) bounded above by the number of equivalence nodes

n_{eq}, and $\sum_x e_x$ equals the total number of edges e, the complexity is $O(n_{eq}e)$.

However, typically, E is fairly sparse since the DAG is typically "short and fat" – as the number of queries grows, the height of the DAG may not increase, but it becomes wider. Thus, $n_x \ll n_{eq}$ for most x, making this sharability computation algorithm fairly efficient in practice. In fact, for the queries we considered in our performance study (Section 3.6), the computation took at most a few tens of milliseconds.

3.3.2 Incremental Cost Update

The sets with which $bestcost$ is called successively at line L1 of Figure 3.3 are closely related, with their (symmetric) difference being very small. For, line L1 finds the node x with the maximum benefit, which is implemented by calling $bestcost(Q, \{x\} \cup X)$, for different values of x. Thus the second parameter to $bestcost$ changes by dropping one node x_i and adding another x_{i+1}. We now present an incremental cost update algorithm that exploits the results of earlier cost computations to incrementally compute the new plan.

Figure 3.4 outlines our incremental cost update algorithm. Let S be the set of nodes shared at a given point of time, i.e., the previous call to $bestcost$ was with S as the parameter. The incremental cost update algorithm maintains the cost of computing every equivalence node, given that all nodes in S are shared, and no other node is shared. Let S' be the new set of nodes that are shared, i.e., the next call to $bestcost$ has S' as the parameter. The incremental cost update algorithm starts from the nodes that have changed in going from S to S' (i.e., nodes in $S' - S$ and $S - S'$) and propagates the change in cost for the nodes upwards to all their parents; these in turn propagate any changes in cost to their parents if their cost changed, and so on, until there is no change in cost. Finally, to get the total cost we add the cost of computing and materializing all the nodes in S'.

If we perform this propagation in an arbitrary order then in the worst case we could propagate the change in cost through a node x multiple times (for example, once from a node y which is an ancestor of another node z and then from z). A simple mechanism for avoiding repeated propagation uses topological numbers for nodes of the DAG. During DAG generation the DAG

Procedure UPDATECOST
Input: S, previous set of shared nodes, corresponding best plan
S', new set of shared nodes
Output: Best plan corresponding to S'
Begin
// PropHeap is a priority heap (initially empty), containing
// equivalence nodes are ordered by their topological sort number
add $S - S' \cup S' - S$ to PropHeap
while (PropHeap is not empty)
N = equivalence node with minimum topological sort number in PropHeap
Remove N from PropHeap
oldCost = old value of cost(N)
cost(N) = Min $\{$ cost(p) — $p \in children(N)\}$ // $children(N)$ are operation nodes
if (cost(N) $\neq$ oldCost) or $N \in (S - S')$ or $N \in (S' - S)$
for every parent operation node p of N
cost(p) = cost of executing operation p + $\sum_{c \in children(p)}(C(c))$
where $C(c)$ = cost(c) if $c \notin S'$, and $= min(reusecost(c)$, cost(c)) if $c \in S'$
add p's parent equivalence node to PropHeap if not already present
TotalCost = $compcost(root) + \sum_{s \in S'}$(cost($s$) $+matcost(s)$)
End

Figure 3.4: Incremental Cost Update

is sorted topologically such that a descendant always comes before an ancestor in the sort order, and nodes are numbered in this order. As shown in Figure 3.4, cost propagation is performed in the topological number ordering using *PropHeap*, a heap built on the topological number. The heap is used to efficiently find the node with the minimum topological sort number at each step.

In our implementation, we additionally take care of physical property subsumption. Details of how to perform incremental cost update on Physical Query DAGs with physical property subsumption are given in the appendix of this chapter.

3.3.3 The Monotonicity Heuristic

In Figure 3.3, the function $bestcost$ will be called once for each node in Y, under normal circumstances. We now outline how to determine the node with the smallest value of $bestcost$ much more efficiently, using the monotonicity heuristic.

Let us define $benefit(x, X)$ as $bestcost(Q, X) - bestcost(Q, \{x\} \cup X)$. Notice that, minimizing $bestcost$ in line $L1$ corresponds to maximizing benefit as defined here. Suppose the benefit is *monotonic*. Intuitively, the benefit of a node is monotonic if it never increases as more nodes get materialized; more formally $benefit$ is monotonic if $\forall X \supseteq Y$, $benefit(x, X) \leq benefit(x, Y)$.

We associate an upper bound on the benefit of a node in Y and maintain a heap $\mathcal{C}$ of nodes ordered on these upper bounds.[4] The initial upper bound on the benefit of a node in Y uses the notion of the maximum degree of sharing of the node (which we described earlier). The initial upper bound is then just the cost of evaluating the node (without any materializations) times the maximum degree of sharing. The heap $\mathcal{C}$ is now used to efficiently find the node $x \in Y$ with the maximum $benefit(x, X)$ as follows: Iteratively, the node n at the top $\mathcal{C}$ is chosen, its current benefit is recomputed, and the heap $\mathcal{C}$ is reordered. If n remains at the top, it is deleted from the $\mathcal{C}$ heap and chosen to be materialized and added to X. Assuming the monotonicity property holds, the other values in the heap are upper bounds, and therefore, the node n added to X above, is indeed the node with the maximum real benefit.

If the monotonicity property does not hold, the node with maximum current benefit may not

[4]This cost heap is not to be confused with the heap on topological numbering used earlier.

be at the top of the heap $\mathcal{C}$, but we still use the above procedure as a heuristic for finding the node with the greatest benefit. Our experiments in Section 3.6 demonstrate that the above procedure greatly speeds up the greedy algorithm. Further, for all queries we experimented with, the results were exactly the same even if the monotonicity heuristic was not used.

3.4 Handling Physical Properties

The greedy algorithm described in Section 3.3 is in the context of the Logical Query DAG and selects logical equivalence nodes to materialize. However, in reality, the algorithm works over the Physical Query DAG instead, and selects the physical equivalence nodes to materialize. While the core algorithm and the sharability and monotonicity optimizations can be trivially restated to address the above change of context, the incremental recomputation optimization needs to be refined nontrivially to address the newer issues involving physical property subsumption and enforcer plans. In this section, we explain these issues and describe the change to the incremental recomputation algorithm.

Given the current best plan and an unmaterialized physical equivalence node, the incremental propagation algorithm is required to compute the new best plan when the given physical equivalence node is additionally materialized.

The additional materialization may affect the best plans for all the physical equivalence nodes for the same logical equivalence node. The propagation process starts by recomputing the best plans for these nodes. This may further affect, transitively, the best plans of all the physical equivalence nodes that belong to the logical equivalence nodes that are ancestors of this logical equivalence node. Thus, as in the algorithm described earlier, the propagation occurs across logical equivalence nodes – these nodes are visited bottom-up in a topological manner in order to prevent multiple visits of the same logical equivalence nodes.

Let L be a logical equivalence node being visited during the propagation, and let $E_1, E_2, \ldots, E_k$ be the physical equivalence nodes belonging to L. The crux of this section is to show how to compute the best plans for each E_i given (a) the best plans for all the physical equivalence nodes belonging to L's children logical equivalence nodes, and (b) for each pair E_i and E_j, the cost of

computing a E_j from E_i – if E_i is materialized then this cost includes the cost of reading E_i, and if E_j is materialized then this cost includes the cost of materializing E_j.

The first step is to compute the best algorithm plan for each E_i; this is straightforward since the costs of the inputs of all the algorithms below E_i is known – so we just need to recompute the cost of the corresponding algorithm plans and pick the cheapest one.

An example scenario is shown in Figure 3.5(a). E_1, E_2 and E_3 are physical equivalence nodes representing the same logical equivalence node with different physical properties; among these, E_1 and E_3 are specified as materialized, while E_2 is not. For each pair E_i and E_j, the cost of obtaining E_j from E_i is shown as the weight of directed edge $E_i \rightarrow E_j$. Further, the best algorithm plans for each E_1, E_2 and E_3 are also shown with the respective plan costs noted alongside.

The next step is to consider the enforcer plans for each E_i as well and choose the overall best plan. An obvious approach is to first compute the best enforcer plan for E_i by enumerating all the enforcer plans and select the cheapest one; comparing the best enforcer plan with the best algorithm plan for E_i determined earlier will then give the node's best plan. We illustrate this approach by an example.

Consider again the scenario of Figure 3.5(a). E_1's best algorithm plan has a cost of 3. E_1 also has two enforcer plans. The first computes E_2 using its algorithm plan at a cost of 1, and then derives E_1 from the result at an additional cost of 3 units – a total cost of 4; the second derives E_1 from E_3 at a cost of 1 – the cost of computing E_3 is not added since it is marked as materialized. Comparing the costs, the second enforcer plan is chosen as the best plan for computing E_1. Similarly, E_3's best algorithm plan has a cost of 3 and its two enforcer plans are as follows. The first computes E_2 using its algorithm plan at a cost of 1 and derives E_3 from the result at a cost of 1 – a total cost of 2. The second plan computes E_3 from materialized E_1 at a cost of 2. Breaking the tie among the two enforcer plans arbitrarily, the second enforcer plan is chosen as the best plan. Thus, the best plan for E_1 derives it from materialized E_3 while the best plan for E_3 derives it from materialized E_1 – this mutual derivation is clearly absurd.

The above example shows that while the approach described above works for unmaterialized nodes, it may not work for materialized nodes. We now give the details of our approach of

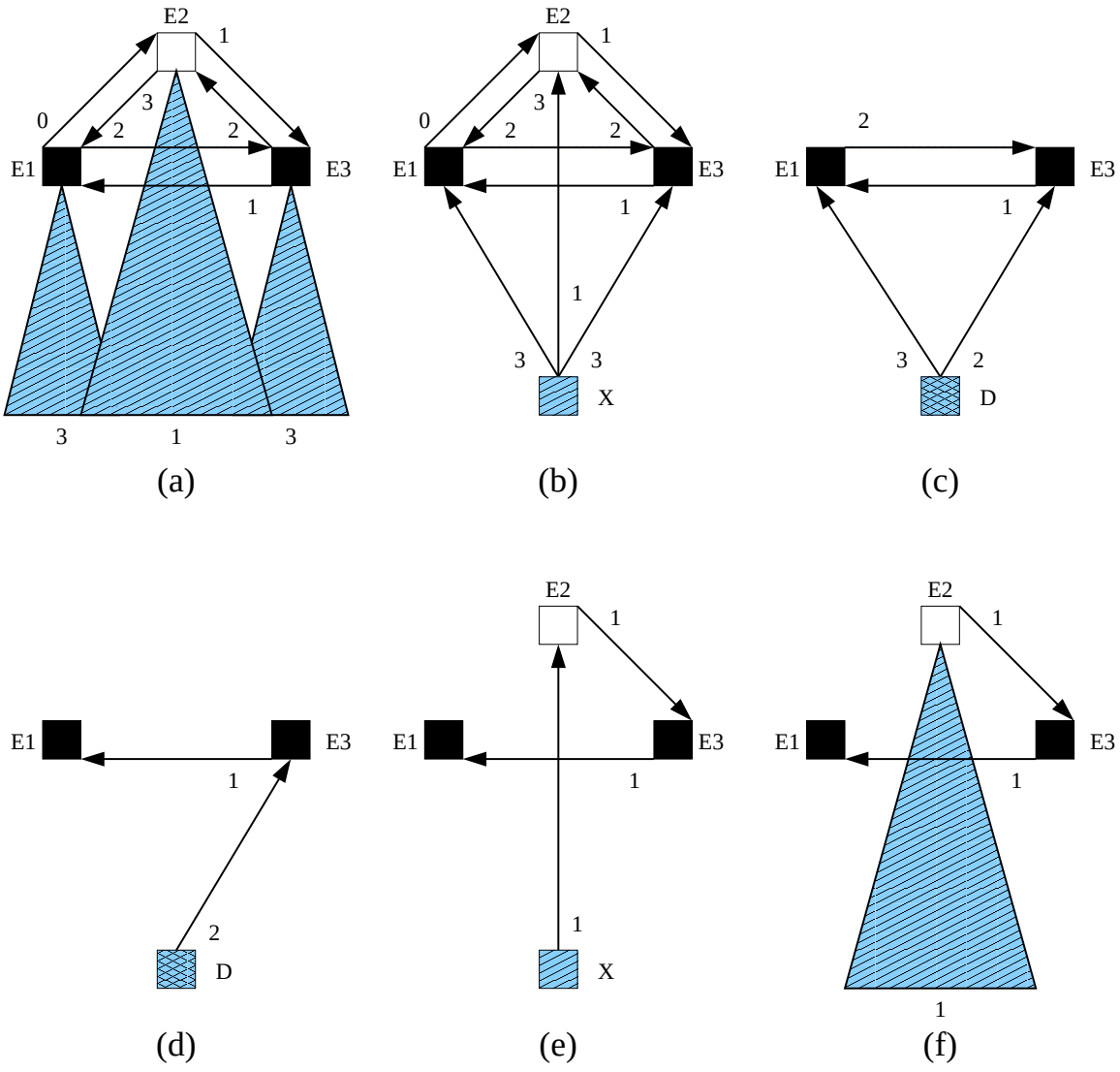


Figure 3.5: Example Showing Cost Propagation through Physical Equivalence Nodes

computing the best plans for the materialized nodes.

We introduce a dummy "external" node X and, for each E_i, replace the best algorithm plan for E_i by its cost summary in terms of an edge from X to E_i weighted by the cost of the algorithm plan. Figure 3.5(b) shows the result of the above transformation on our running example.

Next, for each E_i, we find the *shortest path* from E to E_i; this shortest path represents the best plan for computing E_i assuming none of $E_1, E_2, \ldots, E_k$ are materialized. To keep track of these shortest paths, we introduce another dummy node D and add an edge to each E_i from D representing the shortest path found as above. The edge is weighted by the sum of the edges in the shortest path.

Now, we consider the subgraph induced by D and the materialized nodes among $E_1, E_2, \ldots, E_k$. Each edge into the node E_i in this graph represents a way to compute E_j – if the edge is from D, then it corresponds to computing the result from X using the plan represented by the shortest path, and materializing it; otherwise if it is from some other materialized node E_j then it corresponds to reading the result, deriving E_i from it, and materializing it. We need to pick a set of edges, one into each E_i and without generating any cycles, such that the sum of the costs on the edges (the total cost) is minimized. This corresponds to a *minimum cost directed spanning tree* of the graph, which can be computed efficiently using Edmond's algorithm [17]. This spanning tree gives us – after expanding out any edges out of D included in this tree into the corresponding path – the best plan for each materialized node, taking into consideration other materialized nodes.

For our running example, Figure 3.5(c) shows the subgraph induced by the materialized nodes E_1 and E_2 and the dummy node D. Figure 3.5(d) shows the minimum cost directed spanning tree for the graph. The edge from D to E_3 is expanded out to the path $X \rightarrow E_2 \rightarrow E_3$ in Figure 3.5(e). The final best plan, obtained by replacing the edge $X \rightarrow E_2$ by the best algorithm plan for E_2, is shown in Figure 3.5(f). This plan corresponds to computing E_2 using its best algorithm plan, computing E_3 using the enforcer plan containing E_2's algorithm plan and materializing it, and computing E_1 from E_2, available as materialized.

Note that the solution is heuristic to the extent that some of the materialized nodes may not be needed in the overall best plan, and if eliminated, some other minimum spanning tree may have

resulted. However, we do not know the set of nodes that will get used. Hence, we conservatively assume that all of them may be used, and compute the spanning tree across all the materialized nodes.

3.5 Extensions

In this section, we briefly outline extensions to i) incorporate creation and use of temporary indices, ii) optimize nested queries to exploit common sub-expressions and iii) optimize multiple invocations of parameterized queries.

3.5.1 Selection of Temporary Indices

Costs may be substantially reduced by creating (temporary) indices on database relations or materialized intermediate results. To incorporate index selection, we model the presence of an index as a physical property, similar to sort order. Since our algorithms are actually executed on the physical DAG, they choose not only what results to materialize but also what physical properties they should have. Index selection then falls out as simply a special case of choosing physical properties, with absolutely no changes to our algorithms.

Note that our framework allows us to consider materialization of indices even if the corresponding relation is not materialized, which is useful for algorithms such as index-only joins.

3.5.2 Nested Queries

One approach to handling nested queries is to use decorrelation techniques (see, e.g. [55]). The use of such decorrelation techniques results in the query being transformed to a set of queries, with temporary relations being created. Now, the queries generated by decorrelation have several subexpressions in common, and are therefore excellent candidates for multi-query optimization. One of the queries in our performance evaluation brings out this point.

Correlated evaluation is used in other cases, either because it may be more efficient on the query, or because it may not be possible to get an efficient decorrelated query using standard

relational operations [43]. In correlated evaluation, the nested query is repeatedly invoked with different values for correlation variables. Consider the following query.

```
Query: select * from a, b, c
       where a.x = b.x and b.y = c.y and
             a.cost = (select min(a1.cost) from  a as a1, b as b1
                       where a1.x = b1.x and b1.y = c.y)
```

One option for optimizing correlated evaluation of this query is to materialize $a \bowtie b$, and share it with the outer level query and across nested query invocations. An index on $a \bowtie b$, on attribute $b.y$ is required for efficient access to it in the nested query, since there is a selection on $b.y$ from the correlation variable. If the best plan for the outer level query uses the join order $(a \bowtie b) \bowtie c$, materializing and sharing $a \bowtie b$ may provide the best plan.

In general, parts of the nested query that do not depend on the value of correlation variables can potentially be shared across invocations [43]. We now show how to extend our algorithms to consider such reuse across multiple invocations of a nested query. The key intuition is that when a nested query is invoked many times, benefits due to materialization must be multiplied by the number of times it is invoked; results that depend on correlation variables, however, must not be considered for materialization. The nested query invariant optimization techniques of [43] then fall out as a special case of ours.

The inner subquery forms part of a predicate of some select or join operation of an outer query. This predicate has a pointer to an equivalence node that forms the root of the Query DAG for the inner subquery. Common results between the Query DAGs of the inner subquery and outer query are unified. Thus, unlike optimizers that perform block at a time optimization, we can share optimization effort between the outer and the inner subquery.

In the Query DAG for the inner subquery, the predicate for a select or a join operation node can contain a reference to a correlation variable from the outer query. Let us call such a node a *referencer* node. Clearly, the result of an expression that contains a referencer node varies across different calls to the subquery (depending on the value of the correlation variable) and therefore can not be materialized and shared across calls with different parameter values. Hence, we tag

the equivalence node under which a referencer node occurs as well as all its ancestor nodes in the inner subquery's Query DAG as non-materializable. Such tagging can be performed efficiently while the inner subquery's Query DAG is being constructed.

The cost of the inner subquery is the product of (a) the cost of the best plan in the inner Query DAG, and (b) an estimate of the number of times the inner subquery is invoked.

After the above constructions, the rest of our optimization algorithms are used unchanged, except that they do not consider materializing nodes tagged as non-materializable. An important point to note here is that the above construction allows us to share computation not only across multiple invocations of the inner subquery, but also between the inner subquery and the outer query (see Example 3.0.1).

Extensions that allow memoization of results of the different invocation of the inner subquery (or even intermediate results of these invocations), along with the corresponding correlation variable values, are possible. These will reduce the number of times the inner subquery is evaluated [51]. Such optimizations are independent of the optimizations we present, and can be used in conjunction. Note that if inner subquery's results are memoed, the inner subquery is invoked as many times as there are distinct parameter values.

Parameterized Queries. Our algorithms can also be extended to optimize multiple invocations of parameterized queries. Parameterized queries are queries that take parameter values, which are used in selection predicates; stored procedures are a common example. Parts of the query may be invariant, just as in nested queries, and these can be exploited by multi-query optimization.

Although there has been much work on optimizing parameterized queries (e.g., [19]), to the best of our knowledge all the work in this area aims at finding the best way of executing an individual instance, not at multiquery optimization across multiple executions.

3.6 Performance Study

Our algorithms were implemented by extending and modifying a Volcano-based query optimizer we had developed earlier. All coding was done in C++, with the basic optimizer taking approx.

17,000 lines, common MQO code took 1000 lines, Volcano-SH and Volcano-RU took around 500 lines each, and Greedy took about 1,500 lines.

The optimizer transformation rule set is listed in Appendix B. Implementation algorithms included sort-based aggregation, merge join, nested loops join, indexed join, indexed select and relation scan. The cost estimation formulae for these operators appear in Appendix C. Our implementation incorporates all the techniques discussed in this chapter, including handling physical properties (sort order and presence of indices) on base and intermediate relations, unification and subsumption during DAG generation, and the sharability algorithm for the greedy heuristic.

The block size was taken as 4KB and our cost functions assume 6MB is available to each operator during execution (we also conducted experiments with larger memory sizes up to 128 MB, with similar results). Standard techniques were used for estimating costs, using statistics about relations. The cost estimates contain an I/O component and a CPU component, with seek time as 10 msec, transfer time of 2 msec/block for read and 4 msec/block for write, and CPU cost of 0.2 msec/block of data processed. We assume that intermediate results are pipelined to the next input, using an iterator model as in Volcano; they are saved to disk only if the result is to be materialized for sharing. The materialization cost is the cost of writing out the result sequentially.

The tests were performed on a single processor 233 Mhz Pentium-II machine with 64 MB memory, running Linux. Optimization times are measured as CPU time (user+system).

3.6.1 Basic Experiments

The goal of the basic experiments was to quantify the benefits and cost of the three heuristics for multi-query optimization, Volcano-SH, Volcano-RU and Greedy, with plain Volcano-style optimization as the base case. We used the version of Volcano-RU which considers the forward and reverse orderings of queries to find sharing possibilities, and chooses the minimum cost plan amongst the two.

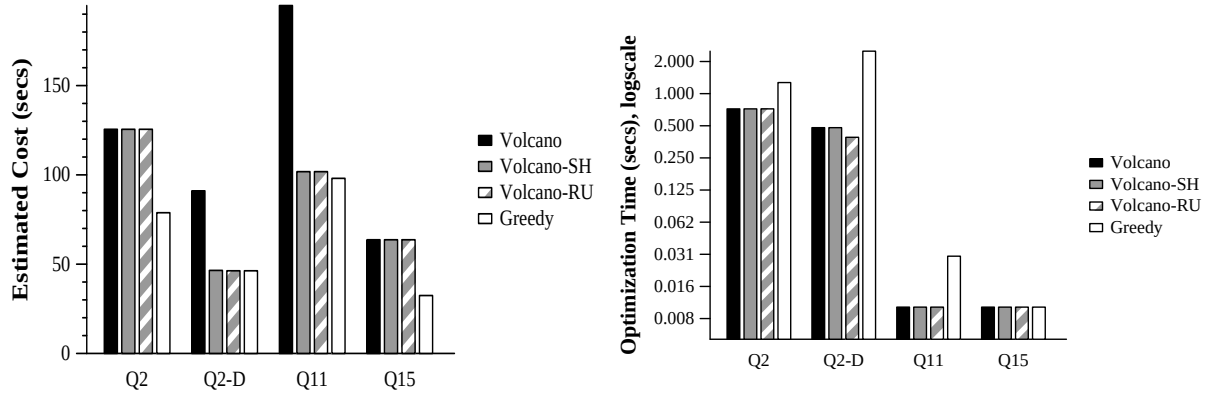


Figure 3.6: Optimization of Stand-alone TPCD Queries

Experiment 1 (Stand-Alone TPCD)

The workload for the first experiment consisted of four queries based on the TPCD benchmark [60]. The queries are listed in Appendix A. We used the TPCD database at scale of 1 (i.e., 1 GB total size), with a clustered index on the primary keys for all the base relations. The results are discussed below and plotted in Figure 3.6.

TPCD query Q2 has a large nested query, and repeated invocations of the nested query in a correlated evaluation could benefit from reusing some of the intermediate results. For this query, though Volcano-SH and Volcano-RU do not lead to any improvement over the plan of estimated cost 126 secs. returned by Volcano, Greedy results in a plan of with significantly reduced cost estimate of 79 secs. Decorrelation is an alternative to correlated evaluation, and Q2-D is a (manually) decorrelated version of Q2 (due to decorrelation, Q2-D is actually a batch of queries). Multi-query optimization also gives substantial gains on the decorrelated query Q2-D, resulting in a plan with estimated costs of 46 secs., since decorrelation results in common subexpressions. Clearly the best plan here is multi-query optimization coupled with decorrelation.

Observe also that the cost of Q2 (without decorrelation) with Greedy is much less than with Volcano, and is less than even the cost of Q2-D with plain Volcano — this results indicates that multi-query optimization can be very useful in other queries where decorrelation is not possible. To test this, we ran our optimizer on a variant of Q2 where the in clause is changed to not in clause, which prevents decorrelation from being introduced without introduc-

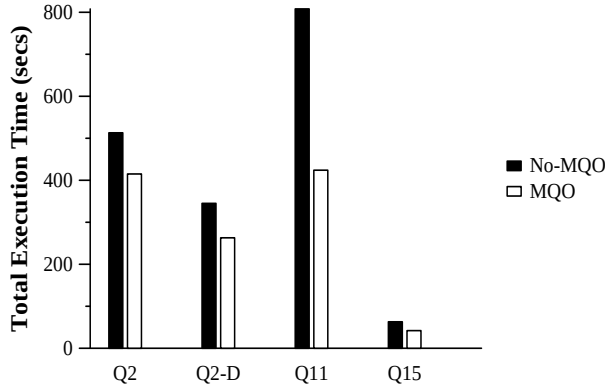


Figure 3.7: Execution of Stand-alone TPCD Queries on MS SQL Server

ing new internal operators such as anti-semijoin [43]. We also replaced the correlated predicate $PS\_PARTKEY = P\_PARTKEY$ by $PS\_PARTKEY \neq P\_PARTKEY$. For this modified query, Volcano gave a plan with estimated cost of 62927 secs., while Greedy was able to arrive at a plan with estimated cost 7331, an improvement by almost a factor of 9.

We next considered the TPCD queries Q11 and Q15, both of which have common subexpressions, and hence make a case for multi-query optimization.[5] For Q11, each of our three algorithms lead to a plan of approximately half the cost as that returned by Volcano. Greedy arrives at similar improvements for Q15 also, but Volcano-SH and Volcano-RU do not lead to any appreciable benefit for this query.

Overall, Volcano-SH and Volcano-RU take the same time and space as Volcano. Greedy takes more time than the others for all the queries. In terms of relative time taken, Greedy needed a maximum of about 5 times as much time as Volcano, but took a maximum of just over 2 seconds, which is very small compared to its benefits. The total space required by Greedy ranged from 1.5 to 2.5 times that of the other algorithms, and again the absolute values were quite small (up to just over 130KB).

Results on Microsoft SQL-Server 6.5:

To study the benefits of multi-query optimization on a real database, we tested its effect on

[5]As mentioned earlier, we use the term multi-query optimization to mean optimization that exploits common subexpressions, whether across queries or within a query.

the queries mentioned above, executed on Microsoft SQL Server 6.5, running on Windows NT, on a 333 Mhz Pentium-II machine with 64MB memory. We used the TPCD database at scale 1 for the tests. To do so, we encoded the plans generated by Greedy into SQL. We modeled sharing decisions by creating temporary relations, populating, using and deleting them. If so indicated by Greedy, we created indexes on these temporary relations. We could not encode the exact evaluation plan in SQL since SQL-Server does its own optimization. We measured the total elapsed time for executing all these steps.

The results are shown in Figure 3.7. For query Q2, the time taken reduced from 513 secs. to 415 secs. Here, SQL-Server performed decorrelation on the original Q2 as well as on the result of multi-query optimization. Thus, the numbers do not match our cost estimates, but clearly multi-query optimization was useful here. The reduction for the decorrelated version Q2-D was from 345 secs. to 262 secs; thus the best plan for Q2 overall, even on SQL-Server, was using multi-query optimization as per Greedy on a decorrelated query. The query Q11 speeded up by just under 50%, from 808 secs. to 424 secs. and Q15 from 63 secs. to 42 secs. using plans with sharing generated by Greedy.

The results indicate that multi-query optimization gives significant time improvements on a real system. It is important to note that the measured benefits are underestimates of potential benefits, for the following reasons. (a) Due to encoding of sharing in SQL, temporary relations had to be stored and re-read even for the first use. If sharing were incorporated within the evaluation engine, the first (non-index) use can be pipelined, reducing the cost further. (b) The operator set for SQL-Server 6.5 seems to be rather restricted, and does not seem to support sort-merge join; for all queries we submitted, it only used (index)nested-loops. Our optimizer at times indicated that it was worthwhile to materialize the relation in a sorted order so that it could be cheaply used by a merge-join or aggregation over it, which we could not encode in SQL/SQL-Server.

In other words, if multi-query optimization were properly integrated into the system, the benefits are likely to be significantly larger, and more consistent with benefits according to our cost estimates.

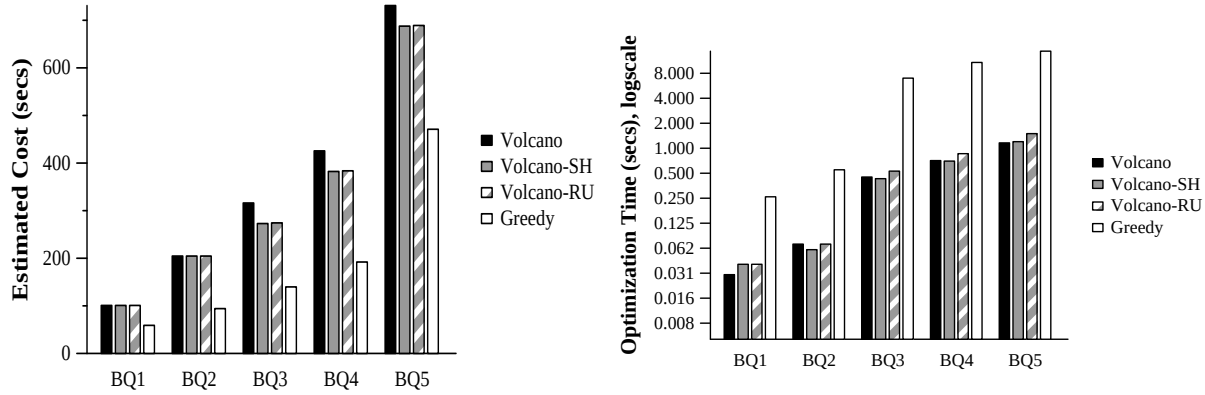


Figure 3.8: Optimization of Batched TPCD Queries

Experiment 2 (Batched TPCD Queries)

In the second experiment, the workload models a system where several TPCD queries are executed as a batch. The workload consists of subsequences of the queries Q3, Q5, Q7, Q9 and Q10 from TPCD; none of these queries has any common subexpressions within itself. These queries are listed in Appendix A. Each query was repeated twice with different selection constants. Composite query BQi consists of the first i of the above queries, and we used composite queries BQ1 to BQ5 in our experiments. Like in Experiment 1, we used the TPCD database at scale of 1 and assumed that there are clustered indices on the primary keys of the database relations.

Note that although a query is repeated with two different values for a selection constant, we found that the selection operation generally lands up at the bottom of the best Volcano plan tree, and the two best plan trees may not have common subexpressions.

The results on the above workload are shown in Figure 3.8. Across the workload, Volcano-SH and Volcano-RU achieve up to only about 14% improvement over Volcano with respect to the cost of the returned plan, while incurring negligible overheads. There was no difference between Volcano-SH and Volcano-RU on these queries, implying the choice of plans for earlier queries did not change the local best plans for later queries. Greedy performs better, achieving up to 56% improvement over Volcano, and is uniformly better than the other two algorithms.

As expected, Volcano-SH and Volcano-RU have essentially the same execution time and space requirements as Volcano. Greedy takes about 15 seconds on the largest query in the set,

BQ5, while Volcano takes slightly more than 1 second on the same. However, the estimated cost savings on BQ5 is 260 seconds, which is clearly much more than the extra optimization time cost of 14 secs. Thus the extra time spent on Greedy is well spent. Similarly, the space requirements for Greedy were more by about a factor of three to four over Volcano, but the absolute difference for BQ5 was only 60KB. The benefits of Greedy, therefore, clearly outweigh the cost.

3.6.2 Scaleup Analysis

To see how well our algorithms scale up with increasing numbers of queries, we defined a new set of 22 relations PSP_1 to PSP_{22} with an identical schema (P, SP, NUM) denoting part id, subpart id and number. Over these relations, we defined a sequence of 18 component queries SQ_1 to SQ_{18}: component query SQ_i was a pair of chain queries on five consecutive relations PSP_i to PSP_{i+4}, with the join condition being $PSP_j.SP = PSP_{j+1}.P$, for $j = i..i+3$. One of the queries in the pair SQ_i had a selection $PSP_i.NUM \geq a_i$ while the other had a selection $PSP_i.NUM \geq b_i$ where a_i and b_i are arbitrary values with $a_i \neq b_i$.

To measure scaleup, we use the composite queries CQ_1 to CQ_5, where CQ_i is consists of queries SQ_1 to SQ_{4i-2}. Thus, CQ_i uses $4i+2$ relations PSP_1 to PSP_{4i+2}, and has $32i-16$ join predicates and $8i-4$ selection predicates. Query CQ5, in particular, is on 22 relations and has 144 join predicates and 36 select predicates. The size of the 22 base relations $PSP_1, \ldots, PSP_{22}$ varied from 20000 to 40000 tuples (assigned randomly) with 25 tuples per block. No index was assumed on the base relations.

The cost of the plan and optimization time for the above workload is shown in Figure 3.9. The relative benefits of the algorithms remains similar to that in the earlier workloads, except that Volcano-RU now gives somewhat better plans than Volcano-SH. Greedy continues to be the best, although it is relatively more expensive. The optimization time for Volcano, Volcano-SH and Volcano-RU increases linearly. The increase in optimization time for Greedy is also practically linear, although it has a very small super-linear component. But even for the largest query, CQ5 (with 22 relations, 144 join predicates and 36 select predicates) the time taken was only 35 seconds. The size of the DAG increases linearly for this sequence of queries. From the

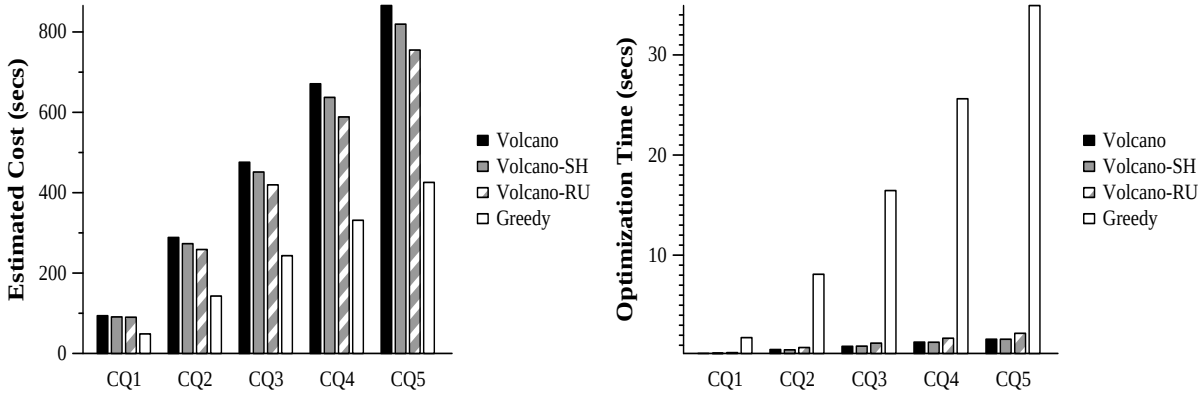


Figure 3.9: Optimization of Scaleup Queries

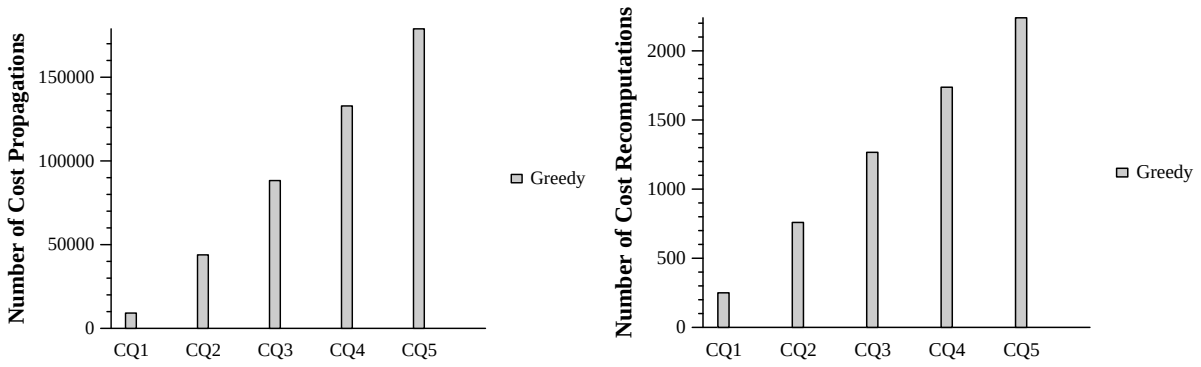


Figure 3.10: Complexity of the Greedy Heuristic

above, we can conclude that Greedy is scalable to quite large query batch sizes.

To better understand the complexity of the Greedy heuristic on the scaleup workload, in addition to the optimization time we measured the total number of times cost propagation occurs across equivalence nodes, and the total number of times cost recomputation is initiated. The result is plotted in Figure 3.10. Note that in addition to the size of the DAG, the number of sharable nodes also increases linearly across queries CQ1 to CQ5.

Greedy was considered expensive by [57] because of its worst case complexity: it can be as much as $O(k^2e)$, where k is the number of nodes in the DAG which are sharable, and e is the number of edges in the DAG. However, for multi-query optimization, the DAG tends to be wide rather than tall – as we add queries, the DAG gets wider, but its height does not increase, since the height is defined by individual queries.

The result shows that for the given workload, the number of times cost propagation occurs across equivalence nodes, and the number of times cost recomputation is initiated both increase almost linearly with number of queries. The observed complexity is thus much less than the worst case complexity.

The number of times costs are propagated across equivalence nodes is almost constant per cost recomputation. This is because the number of nodes of the DAG affected by a single materialization does not vary much with number of queries, which is exploited by incremental cost recomputation. The height of the DAG remains constant (since the number of relations per query is fixed, which is a reasonable assumption).

3.6.3 Effect of Optimizations

In this series of experiments, we focus on the effect of individual optimizations on the optimization of the scaleup queries. We first consider the effect of the monotonicity heuristic addition to Greedy. Without the monotonicity heuristic, before a node is materialized the benefits would be recomputed for all the sharable nodes not yet materialized. With the monotonicity heuristic addition, we found that on an average only about 45 benefits were recomputed each time, across the range of CQ1 to CQ5. In contrast, without the monotonicity heuristic, even at CQ2 there were about 1558 benefit recomputations each time, leading to an optimization time of 77 seconds for the query, as against 8 seconds with monotonicity. Scaleup is also much worse without monotonicity. Best of all, the plans produced with and without the monotonicity heuristic assumption had virtually the same cost on the queries we ran. Thus, the monotonicity heuristic provides very large time benefits, without affecting the quality of the plans generated.

To find the benefit of the sharability computation, we measured the cost of Greedy with the sharability computation turned off; every node is assumed to be potentially sharable. Across the range of scaleup queries, we found that the optimization time increased significantly. For CQ2, the optimization time increased from 35 secs. to 46 secs. Thus, sharability computation is also a very useful optimization.

In summary, our optimizations of the implementation of the greedy heuristic result in an

order of magnitude improvement in its performance, and are critical for it to be of practical use.

3.6.4 Discussion

To check the effect of memory size on our results, we ran all the above experiments increasing the memory available to the operators from 6MB to 32MB and further to 128MB. We found that the cost estimates for the plans decreased slightly, but the relative gains (i.e., cost ratio with respect to Volcano) essentially remained the same throughout for the different heuristics.

We stress that while the cost of optimization is independent of the database size, the execution cost of a query, and hence the benefit due to optimization, depends upon the size of the underlying data. Correspondingly, the benefit to cost ratio for our algorithms increase markedly with the size of the data. To illustrate this fact, we ran the batched TPCD query BQ5 (considered in Experiment 2) on TPCD database with scale of 100 (total size 100GB). Volcano returned a plan with estimated cost of 106897 seconds while Greedy obtains a plan with cost estimate 73143 seconds, an improvement of 33754 seconds. The extra time spent during optimization is 14 seconds, as before, which is negligible relative to the gain.

While the benefits of using MQO show up on query workloads with common subexpressions, a relevant issue is the performance on workloads with rare or nonexistent overlaps. If it is known apriori that the workload is not going to benefit from MQO, then we can set a flag in our optimizer that bypasses the MQO related algorithms described in this chapter, reducing to plain Volcano.

To study the overheads of our algorithms in a case with no sharing, we took TPCD queries Q3, Q5, Q7, Q9 and Q10, renamed the relations to remove all overlaps between queries, and created a batch consisting of the queries with relations renamed. The overheads of Volcano-SH and Volcano-RU are neglibible, as discussed earlier. Basic Volcano optimization took 650 msec, while the Greedy algorithm took 820 msec. Thus the overhead was around 25%, but note that the absolute numbers are very small. With no overlap, the sharability detection algorithm finds no node sharable, causing the Greedy algorithm to terminate immediately (returning the same plan as Volcano). Thus, the overhead in Greedy is due to (a) expansion of the entire DAG, and (b) the execution of the sharability detection algorithm. Of this overhead, cause (a) is predominant, and

the sharability computation was quite cheap on queries with no sharing.

In our experiments, Volcano-RU was better than Volcano-SH only in a few cases, but since their run times are similar, Volcano-RU is preferable. There exist cases where Volcano-RU finds out plans as good as Greedy in a much less time and using much less space; but on the other hand, in the above experiments we saw many cases where additional investment of time and space in Greedy pays off and we get substantial improvements in the plan.

To summarize, for very low cost queries, which take only a few seconds, one may want to use Volcano-RU, which does a "quick-and-dirty" job; especially so if the query is also syntactically complex. For more expensive queries, as well as "canned" queries that are optimized rarely but executed frequently over large databases, it clearly makes sense to use Greedy.

3.7 Related Work

The multi-query optimization problem has been addressed in [18, 54, 56, 53, 13, 38, 10, 64, 59]. The work in [54, 56, 53, 13, 38] describe exhaustive algorithms; they use an abstract representation of a query Q_i as a set of alternative plans $P_{i,j}$, each having a set of tasks $t_{i,j,k}$, where the tasks may be shared between plans for different queries. They do not exploit the hierarchical nature of query optimization problems, where tasks have subtasks. Finally, these solutions are not integrated with an optimizer.

The work in [59] considers sharing only amongst the best plans of each query – this is similar to Volcano-SH, and as we have seen, this often does not yield the best sharing.

The problem of materialized view/index selection [45, 44, 63, 9, 34, 26] is related to the multi-query optimization problem. The issue of materialized view/index selection for the special case of aggregates/data-cubes is considered in [29, 27] and implemented in Redbrick Vista [11]. The view selection problem can be viewed as finding the best set of sub-expressions to materialize, given a workload consisting of both queries and updates. The multi-query optimization problem differs from the above since it assumes absence of updates, but it must keep in mind the cost of computing the shared expressions, whereas the view selection problem concentrates on the cost of keeping shared expressions up-to-date. It is also interesting to note that multi-

query optimization is needed for finding the best way of propagating updates on base relations to materialized views [44].

Several of the algorithms presented for the view selection problem ([29, 27, 26]) are similar in spirit to our greedy algorithm, but none of them described how to efficiently implement the greedy heuristic. Our major contribution here lies in making the greedy heuristic practical through our optimizations of its implementation. We show how to integrate the heuristic with the optimizer, allowing incremental recomputation of benefits, which was not considered in any of the earlier work, and our sharability and monotonicity optimizations also result in great savings. The lack of an efficient implementation could be one reason for the authors in [57] to claim that the greedy algorithm can be quite inefficient for selecting views to materialize for cube queries. Another reason is that, for multi-query optimization of normal SQL queries (modeled by our TPC-D based benchmarks) the DAG is "short and fat", whereas DAGs for complicated cube queries tend to be taller. Our performance study (Section 3.6) indicates the greedy heuristic is quite efficient, thanks to our optimizations.

Another related area is that of caching of query results. Whereas multiquery optimization can optimize a batch of queries given together, caching takes a sequence of queries over time, deciding what to materialize and keep in the cache as each query is processed. Related work in caching includes [10, 64, 33]. The work in [64, 33] considers only queries that can be expressed as a single multi-dimensional expression. The work in [10] addresses the issue of management of a cache of previous results but considers only select-project-join (SPJ) queries. We consider a more general class of queries.

Our multi-query optimization algorithms implement query optimization in the presence of materialized/cached views, as a subroutine. By virtue of working on a general DAG structure, our techniques are extensible, unlike the solutions of [8] and [10]. The problem of detecting whether an expression can be used to compute another has also been studied in [35, 62, 52]; however, they do not address the problem of choosing what to materialize, or the problem of finding the best query plans in a cost-based fashion.

Recently, [43] considers the problem of detecting invariant parts of a nested subquery, and teaching the optimizer to choose a plan that keeps the invariant part as large as possible. Perform-

ing multi-query optimization on nested queries automatically solves the problem they address.

Our algorithms have been described in the context of a Volcano-like optimizer; at least two commercial database systems, from Microsoft and Tandem, use Volcano based optimizers. However, our algorithms can also be modified to be added on top of existing System-R style bottom-up optimizers; the main change would be in the way the DAG is represented and constructed.

3.8 Summary

We have described three novel heuristic search algorithms, Volcano-SH, Volcano-RU and Greedy, for multi-query optimization. We presented a a number of techniques to greatly speed up the greedy algorithm. Our algorithms are based on the AND/OR Query DAG representation of queries, and are thereby can be easily extended to handle new operators. Our algorithms also handle index selection and nested queries, in a very natural manner. We also developed extensions to the DAG generation algorithm to detect all common sub expressions and include subsumption derivations.

Our implementation demonstrated that the algorithms can be added to an existing optimizer with a reasonably small amount of effort. Our performance study, using queries based on the TPC-D benchmark, demonstrates that multi-query optimization is practical and gives significant benefits at a reasonable cost. The benefits of multi-query optimization were also demonstrated on a real database system. The greedy strategy uniformly gave the best plans, across all our benchmarks, and is best for most queries; Volcano-RU, which is cheaper, may be appropriate for inexpensive queries.

Our multi-query optimization algorithms were partially prototyped on Microsoft SQL Server in summer ’99, and are currently being evaluated by Microsoft for possible inclusion in SQL Server.

In conclusion, we believe we have laid the groundwork for practical use of multi-query optimization, and *multi-query optimization will form a critical part of all query optimizers in the future*.

Chapter 4

Query Result Caching

Data warehouses are becoming increasingly important parts of data analysis for decision support. The typical processing time of decision support queries range from minutes to hours. This is due to the nature of complex queries used for decision making. The aim of the work presented in this chapter[1] is to improve query response times by caching final as well as intermediate results produced during query processing.

In a traditional database engine, every query is processed independently. In decision support applications, queries often overlap in the data that they access and in the manner in which they utilize the data, i.e., there are common expressions between queries. A natural way to improve performance is to allocate a limited-size area on the disk to be used as a cache for results computed by previous queries. The contents of the cache may be utilized to speed up the execution of subsequent queries. We use the term *query caching* in this chapter to mean caching of final and/or intermediate results of queries.

Most exisiting decision support systems support *static view selection*: select a set of views apriori, and keep them permanently on disk. The selection is based on either (a) the intuition of the systems administrator, or (b) recommendation of "advisor wizards" as supported by Microsoft SQL-Server [9] based on a workload history. The advantage of query caching addressed in this work over static view selection is that it can cater to changing workloads — the data ac-

[1] Joint work with Krithi Ramamritham, S. Seshadri and S. Sudarshan.

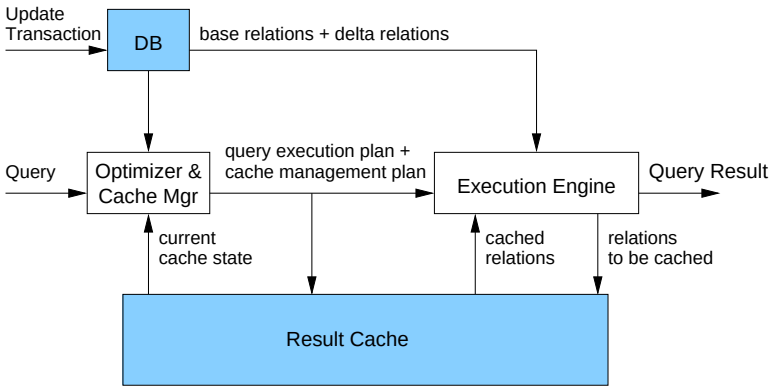


Figure 4.1: Architecture of the Exchequer System

cess patterns of the queries cannot be expected to be static, and to answer all types of queries efficiently, we need to dynamically change the cache contents.

The techniques needed for (a) for intelligently and automatically managing the cache contents, given the cache size contraints, as queries arrive, and (b) for performing query optimization exploiting the cache contents, so as to minimize the overall response time for all the queries, form the crux of this work. These techniques form a part of the Exchequer[2] query caching system. The architecture of the Exchequer system is portrayed in Figure 4.1.

Query results are cached on a fixed-size disk area, called the *result cache*. Thus the caching of a result incurs an overhead of writing the result to disk. If the cached result is to be indexed, the caching overhead includes the index creation overhead. A use of the cached result corresponds to index probes if it is indexed, a full scan otherwise. Our techniques also apply to -memory caching as well as to hybrid two-level (disk cum main-memory) caching. These variants are discussed in Section 4.6.

The cache manager and the optimizer are tightly integrated: (a) the optimizer optimizes an incoming query based on the current cache state, and (b) the cache manager decides which results to cache and which cached results to evict based on the workload (which depends on the sequence of queries in the past).

We assume that the workload presents queries in an ordered sequence, and only one query is

[2]Efficiently eXploiting caCHEd QUEry Results

processed at a time. Extending for concurrent optimization and execution, wherein new queries arrive and are to be optimized and executed while a previous query is being optimized and executed, is a topic of future study. In particular, we assume that the cache contents do not change between the optimization and execution of a query. The results are cached without any projections, to maximize the number of queries that can benefit from a cached result. Extensions to avoid caching very large attributes are possible.

In addition to the above functionality, a caching system should also support invalidation or refresh of cached results in the face of updates to the underlying database. In this chapter, however, we will confine our attention only to the issue of efficient query processing, ignoring updates. Data Warehouses are an example of an application where the cache replacement algorithm can ignore updates, since updates happen only periodically (once a day or even once a week).

The Rest of The Chapter: Section 4.1 describes how Exchequer performs cache-aware query optimization. In order to perform workload-adaptive caching, it is essential to dynamically maintain a characterization of the current workload; how Exchequer achieves this is discussed in Section 4.2. Next, Section 4.3 outlines Exchequer's cache management algorithm. Differences of this work from earlier related work are covered in detail in Section 4.4. Results of experimental evaluation of the proposed algorithms are discussed in Section 4.5. The chapter is summarized in in Section 4.7.

4.1 Cache-Aware Query Optimization

This section explains how cache-aware query optimization is carried out in Exchequer. Section 4.1.1 describes the *Consolidated DAG*, an auxiliary Query DAG (ref. Section 2.2.2) that is used to keep track of the queries in the workload as well as the cache contents. In Section 4.1.2, we outline how a Query DAG for the query is generated and melded with the Consolidated DAG; as we shall show, this takes care of cached result matching and expressing the query in terms of these cached results. Next, in Section 4.1.3, we describe Exchequer's variant of the Volcano query optimization algorithm that uses this Query DAG to find the best plan for the query in the

presence of the cached results.

4.1.1 Consolidated DAG

We now introduce *CDAG*, the Consolidated DAG. CDAG is an auxilliary Query DAG structure underlying Exchequer's algorithms. CDAG contains (a) all the queries in the workload (in the ideal case, when space is not at premium; a more practical alternative is discussed below), and (b) the set of results present in the cache.

CDAG is used (a) to perform cache-aware query optimization, as explained in Section 4.1.2; (b) to determine if a new query has occured earlier in the workload – this is needed in order to maintain query statistics used to characterize the workload, as explained in Section 4.2; and (c) to make dynamic caching decisions, as explained in Section 4.3.

Given the large number of queries involved, the space overhead of CDAG is a concern if all alternative plans of all the queries are to be stored. In practice, therefore, we (a) keep only the *best plan* of each query in the CDAG, and (b) specify a static space constraint and consider only a restricted set of queries to represent the workload, so that the resulting CDAG fits in the given space. Queries may be displaced if they are not expected to recur often in the current workload; how this can be determined is explained in Section 4.2. Note that most commercial database systems maintain a *procedure cache* [58] to cache the optimized plans of the queries in the workload; these procedure caches clearly have similar space overhead.

Due to the displacement of queries (because of the space constraints, as discussed above), as well as due to the evolution with time of the set of cached results, we need to delete and insert queries from CDAG. Since parts of CDAG may be shared by multiple queries and cached results, deletion of intermediate nodes of CDAG is done using a reference counting mechanism.

Equivalence nodes in CDAG that correspond to cached results are marked as such; this allows us to (a) keep track of the cached results for use in the cache-aware optimization algorithm as will be explained in Section 4.1.2, and (b) specify the needed reconfiguration of the cache by marking and unmarking the equivalence nodes as will be explained in Section 4.3.

Figure 4.2(a) shows a CDAG for the query set $\{A \bowtie C \bowtie D, A \bowtie C \bowtie E\}$, and the cached

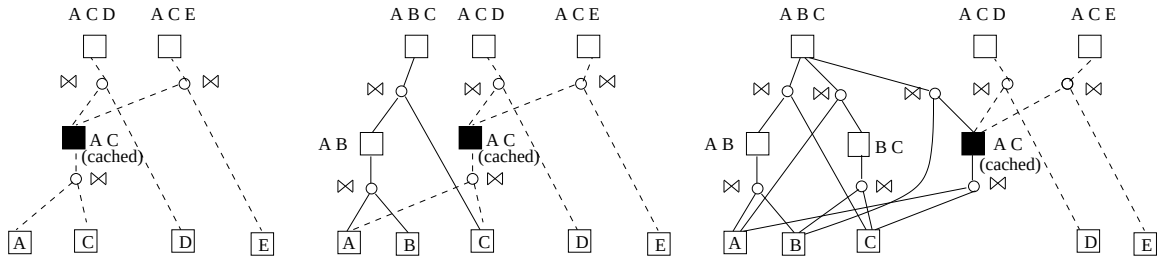


Figure 4.2: (a) CDAG for { A ⋈ C ⋈ D, A ⋈ C ⋈ E } (b) Unexpanded A ⋈ B ⋈ C inserted into CDAG (c) A ⋈ B ⋈ C expanded into CDAG

result set {A ⋈ C}.

4.1.2 Query DAG Generation and Query/Cached Result Matching

When a new query arrives, it is added to CDAG and expanded into its Query DAG. A fallout of the support for unification in our version of the Volcano optimizer (ref. Section 2.2.2) is that since the equivalence nodes in the Query DAG for a query may unify with a CDAG equivalence node that corresponds to a result present in the cache, we automatically get rewritings of the query in terms of the cached results. Moreover, unification allows us to determine if the new query has occured earlier in the workload, since in this case, the root equivalence node of the Query DAG will unify with the root equivalence node corresponding to the query in CDAG. This is needed in order to maintain the statistics needed to characterize the workload (Section 4.2).

As an example, consider again the CDAG of Figure 4.2(a), for the query set {A ⋈ C ⋈ D, A ⋈ C ⋈ E}, and the cached result set {A ⋈ C}. Now, when the query A ⋈ B ⋈ C arrives, its initial unexpanded representation is created and added to the CDAG as shown in Figure 4.2(b). The next step is the expansion of this query tree into the Query DAG for the query shown in Figure 4.2(c). This is achieved by applying all possible transformations on every equivalence node of the query tree. In our example, we assume that the only transformations applied are join associativity and commutativity. (To avoid clutter, the figure does not show the results of applying commutativity on the respective expressions.) In the process, when the expression (A ⋈ C) ⋈ B is generated, the new expression A ⋈ C is found to already exist in the CDAG. It turns

out that the equivalence node for A ⋈ C is marked as present in the cache (see Figure 4.2(c)); the expression (A ⋈ C) ⋈ B represents a rewriting of the query in terms of the cached result A ⋈ C.

Exchequer also detects and handles *subsumption* derivations. For example, suppose two subexpressions $e1$: $\sigma_{A<5}(E)$ and $e2$: $\sigma_{A<10}(E)$ appear in the query. The result of $e1$ can be obtained from the result of $e2$ by an additional selection, i.e., $\sigma_{A<5}(E) \equiv \sigma_{A<5}(\sigma_{A<10}(E))$. To represent this possibility, we add an extra operation node $\sigma_{A<5}$ between $e1$ and $e2$ in the Query DAG. Similarly, given $e3$: $\sigma_{A=5}(E)$ and $e4$: $\sigma_{A=10}(E)$, we introduce a new equivalence node $e5$: $\sigma_{A=5 \vee A=10}(E)$ and add new derivations of $e3$ and $e4$ from $e5$. In general, given a number of selections on an expression E, we create a single new equivalence node representing the disjunction of all the selection conditions. Similar derivations also help with aggregations. For example, if we have $e6$: ${}_{dno}\mathcal{G}_{sum(Sal)}(E)$ and $e7$: ${}_{age}\mathcal{G}_{sum(Sal)}(E)$, we introduce a new equivalence node $e8$: ${}_{dno,age}\mathcal{G}_{sum(Sal)}(E)$ and add derivations of $e6$ and $e7$ from equivalence node $e8$ by further groupbys on dno and age.

Subsumption derivations are important because (a) they allow reuse of cached results even though the cached result does not exactly match a subexpression of the query, but can be used to compute the same; and dually, (b) they make explicit the different ways in which a result may be used, which is important for determining the benefit of caching the result while making the dynamic caching decisions as explained in Section 4.3.

Volcano neither performs unification nor introduces subsumption derivations – these extensions were proposed as a part of our earlier work on multi-query optimization (Chapter 3). The novelty here is to show how this Query DAG framework can be used to perform matching of queries and cached results during optimization *with neglegible overhead on the optimizer*.

In the following section, we discuss how the Query DAG for the new query, generated as explained in this section, is used to generate the best plan for the query in a cache-aware manner.

4.1.3 Volcano Extensions for Cache-Aware Optimization

Exchequer makes use of the above Query DAG representation and uses a variant of the Volcano optimization algorithm (see Chapter 2) to optimize the queries.

The main extension to Volcano for Exchequer involves considering possible use of cached results while determining the minimum-cost plan for a query. To find the cost of a node given a set of equivalence nodes $\mathcal{S}$ whose results are present in the cache, we use the Volcano cost formulae stated above for the query, with the following change.

For the equivalence node e, whose result is present in the cache, let $reusecost(e)$ denote the cost of reusing the cached result. When computing the cost of an operation node o, if an input equivalence node $e' \in \mathcal{S}$, the minimum of $reusecost(e')$ and $cost(e')$ is used for $cost(o)$. Thus, we use the following expression instead:

$$\text{cost}(o) = \text{cost of executing}(o) + \Sigma_{e_i \in children(o)} C(e_i)$$

$$\text{where } C(e_i) = \begin{cases} cost(e_i) & \text{if } e_i \notin \mathcal{S} \\ \min(cost(e_i), reusecost(e_i)) & \text{if } e_i \in \mathcal{S} \end{cases}$$

Thus, the extended optimizer computes best plans for the query in the presence of cached results. The extra optimization overhead is quite small.

4.2 Dynamic Characterization of Current Workload

In this section, we outline how Exchequer characterizes the dynamically changing workload that are needed to make dynamic caching decisions.

Consider a point in time just before the arrival of the i^{th} query Q_i. We model the future workload at this point as a sequence of queries picked from some fixed set according to some fixed probability distribution. Thus, in this model, the set of queries and probability distribution together fully characterize the workload at this point; however, neither of these are known, and need to be predicted. These predictions need to be dynamic, and must be continuously updated to keep track of the changing workload as time progresses.

Our predictions for the future are entirely based on the past. As such, we predict the set of future queries as the set of queries present in CDAG at the given point in time. We denote this set by $\mathcal{U}_i$. Further, let the estimate of the probability distribution at this point be denoted by P_i. We assume the presence of (a) an arbitrary non-empty initial set of queries, $\mathcal{U}_1$, and (b) an arbitrary initial probability distribution, P_1, on $\mathcal{U}_1$. In the discussion below, we show how $\mathcal{U}_i$ and P_i are

updated to $\mathcal{U}_{i+1}$ and P_{i+1} respectively on the arrival of the query Q_i.

When Q_i arrives, it is optimized; the unification extension of Volcano algorithm, described in Section 4.1.2, enables us to determine whether or not $Q_i \in \mathcal{U}_i$. If $Q_i \in \mathcal{U}_i$, the CDAG remains unchanged; if not, Q_i is added to CDAG.[3] Thus, we have $\mathcal{U}_{i+1} = \mathcal{U}_i \cup \{Q_i\}$.

For a given $Q \in \mathcal{U}_{i+1}$, $P_{i+1}(Q)$ is computed using a simple exponential smoothing estimator on the series $\langle I_k(Q)\rangle_{k=1}^{i}$ where the indicator function $I_k(Q)$ is 1 if $Q \equiv Q_k$, and 0 otherwise. Formally:[4]

$$P_{i+1}(Q) = \begin{cases} (1-\alpha)P_i(Q) & \text{if } Q \not\equiv Q_i \\ \alpha & \text{if } Q \equiv Q_i \text{ and } Q_i \notin \mathcal{U}_i \\ (1-\alpha)P_i(Q) + \alpha & \text{if } Q \equiv Q_i \text{ and } Q_i \in \mathcal{U}_i \end{cases}$$

The smoothing factor $\alpha \in [0, 1]$ denotes the bias of the estimator in favour of the recent queries in the workload; we choose $\alpha = 0.05$ in our experiments. The exponential smooting estimator was chosen because of its simplicity and low overhead.

The probability estimates need to be maintained dynamically as the workload progresses. An option is to compute this estimate on the arrival of each successive query using the equations above for each query in the current CDAG. This is clearly not viable due to the large number of queries involved. In practice, therefore, these estimates are maintained lazily and computed only when accessed.

4.3 Cache Management in Exchequer

Consider an arbitrary query Q_i in the workload. The algorithm outlined in this section attempts to determine the intermediate results computed during the execution of Q_i that are worth caching; the goal being to minimize the *expected* execution cost of an arbitrary query in the future workload. This involves comparing the expected benefit of caching the results with (a) the cost involved in storing them on the disk, and (b) the loss due to the displacement of previously cached

[3] Possibly replacing some other queries due to space constraints. This case is not considered in the presented scheme for sake of simplicity; it is trivial extension to the same.

[4] It can be verified that P_{i+1} is a valid probability distribution if P_i is one.

results in order to accomodate these results, if necessary, due to cache space constraints.

As outlined in Section 4.2, the future workload at the point of execution of Q_i is characterized by (a) the set of queries $\mathcal{U}_{i+1}$ and (b) the probability distribution P_{i+1} on $\mathcal{U}_{i+1}$. Let $\mathcal{S}$ be the set of results present in the cache when a query Q' arrives, as a part of the predicted workload. Q' is then optimized using the results in $\mathcal{S}$ as explained in Section 4.1. The expected execution cost of the best plan for Q' chosen by the optimizer is given by $\sum_{Q \in \mathcal{U}_{i+1}}(cost(Q, \mathcal{S}) * P_{i+1}(Q))$, where $cost(Q, \mathcal{S})$ is the cost of computing the query Q given the set of cached results $\mathcal{S}$. However, since $\mathcal{U}_{i+1}$ contains a large number of queries, computation of the above sum is expensive. Thus, we identify a *representative set*, $\mathcal{R}$, a subset of $\mathcal{U}_{i+1}$ containing N queries that are most likely to occur as the next query (as suggested by the distribution P_{i+1} on $\mathcal{U}_{i+1}$) and compute the sum with respect to $\mathcal{R}$ — this is justified since the distribution P_{i+1} is most likely skewed due to locality of reference; therefore, restricting the sum with respect to the most probable queries should give a reasonable approximation of the actual expected cost. We thus compute an approximation $expcost(\mathcal{S})$ of the expected execution cost as:

$$expcost(\mathcal{S}) = \sum_{Q \in \mathcal{R}} (cost(Q, \mathcal{S}) * P_{i+1}(Q))$$

The algorithm described below, thus, chooses the set $\mathcal{S}$ that minimizes $expcost(\mathcal{S})$; Exchequer's execution engine reconfigures the cache accordingly during the execution of Q_i.

Given a set of results $\mathcal{S}$ already chosen for caching by the algorithm, and a result x, $benefit(x, \mathcal{S})$, the benefit of additionally caching node x, is defined as the decrease in $expcost(\mathcal{S})$ (the payoff), minus the cost of caching x, if it is not already present in the cache (the investment). Formally:

$$benefit(x, \mathcal{S}) = \begin{cases} expcost(\mathcal{S}) - expcost(\{x\} \cup \mathcal{S}) \\ \qquad \text{if } x \text{ is present in the cache} \\ expcost(\mathcal{S}) - (expcost(\{x\} \cup \mathcal{S}) + matcost(x)) \\ \qquad \text{if } x \text{ is not present in the cache} \end{cases}$$

where $matcost(x)$ is the cost of caching the new result x, which involves writing x to the disk.

The benefit measured as above is conservative since it does not amortize the $matcost(x)$ over multiple uses; computing a tighter measure of benefit is nontrivial since it is difficult to compute

```
Procedure GREEDY
Input:    C, the set of candidate results for caching
Output:   S, the set of results to be cached
Begin
    S = φ
    while (C ≠ φ)
        Among results y ∈ C
L1:         Pick the result x with the maximum benefit(x, S)/size(x)
            /* i.e., maximum benefit per unit space */
        if (benefit(x, S) ≤ 0 or size({x} ∪ S) > CacheSize)
            break; /* No further benefits to be had, stop */
        C = C − x;   S = S ∪ {x}
    return S
End
```

Figure 4.3: The Greedy Algorithm for Cache Management

apriori how many times the result x is going to be used between its admission into the cache and its replacement. However, in practice, we find that amortizing $matcost(x)$ does not have much effect; this is because for most x with high $benefit(x, \mathcal{S})$, $matcost(x)$ is relatively insignificant.

Figure 4.3 outlines an algorithm, hereafter called Greedy, that takes as input a *candidate set* of results, $\mathcal{C}$, and heuristically selects (for caching) the subset $\mathcal{S}$ of $\mathcal{C}$ with the maximum benefit overall under the cache space constraint of $CacheSize$. The purpose of Greedy is to weigh the benefits of caching the intermediate results that are computed during the execution of the best plan of Q_i against the benefit of retaining results that are already in the cache. As such, the candidate set $\mathcal{C}$ contains:

1. The final and intermediate results in the best plan of Q_i, and

2. The set of results that was selected as having the maximum benefit by the preceding invocation of the algorithm (this set is present in the cache).

Greedy works iteratively as follows. Starting with $\mathcal{S}$ empty, in each iteration, the algorithm greedily selects the node x among the results in $\mathcal{C}$ that, if cached, gives the maximum *benefit per unit space*, and moves it from $\mathcal{C}$ to $\mathcal{S}$. The algorithm terminates when $\mathcal{C}$ becomes empty, benefit becomes zero/negative, or the size of the nodes in $\mathcal{S}$ exceed the cache size, whichever is earlier.

The final value of $\mathcal{S}$ is the set of results to be placed in the cache, and is returned as the output of the algorithm.

After $\mathcal{S}$ has been computed by Greedy, the best plan of Q_i is executed. Two variants of the Exchequer algorithm are possible depending upon what is cached during the execution:

- **Exchequer/NoFullCache:** Only computed intermediate results that are included in $\mathcal{S}$ are added to the cache; no additional nodes are admitted even if there is space in the cache.
- **Exchequer/FullCache:** Apart from computed intermediate results that are included in $\mathcal{S}$, other computed results are also admitted to the cache if there is enough free space in the cache.

The idea behind Exchequer/FullCache is to keep the cache as occupied as possible at all times; however, the experimental results in Section 4.5 show that this does not provide any significant benefit.

In order to make the decisions regarding the eviction of results in the cache not in $\mathcal{S}$, we use *Largest Cache Space/Least Recently Used (LCS/LRU)*, wherein the largest results are preferentially evicted, and amongst all results of the same size, the least recently used one is evicted. We chose this policy because of its low overhead, since it does not need any statistical information. Moreover, this policy has been shown to work best among a host of alternatives considered by ADMS [10].

Optimizations of Greedy Algorithm: Two important optimizations to a greedy algorithm for multi-query optimization, originally proposed in the context of multi-query optimization (Chapter 3), can be adapted for the purpose of selecting the cachable nodes efficiently:

1. Since there are many calls to *benefit* (and thereby to $expcost$) at line L1 of Figure 4.3, with different parameters, a simple option is to process each call to $expcost$ independent of other calls. Our optimization is to *incrementally* update the costs, maintaining the state of the Query DAG (which includes previously computed best plans for the equivalence nodes) across calls to $expcost$. Details can be found in Chapter 3.

2. With the greedy algorithm as presented above, in each iteration the benefit of every candidate result that is not yet cached is recomputed since it may have changed. If we can assume that the benefit of a result cannot increase when another result is chosen to be cached (while this is not always true, it is often true in practise) there is no need to recompute the benefit of a result x if the new benefit of some result y is higher than the previously computed benefit of x. It is clearly preferable to cache y at this stage, rather than x — under the above assumption, the benefit of x could not have increased since it was last computed.

4.4 Differences from Prior Work

Much of the earlier work on caching has been for specialized applications (e.g. data cubes [16, 33, 50], or [10] which handles only select-project-join queries, or [15, 32, 31] which handle just selections). While specialized queries are important, general purpose decision support systems must support more general queries as well. Our algorithms can handle any SQL query, including nested queries. Moreover, our techniques are extensible in that new operators can be added easily, due to the use of the Query DAG framework.

Further, most of the earlier work does not take caching of intermediate results into account (e.g. WatchMan [49]), or has relatively simple cache replacement algorithms, which do not take into account the fact that the benefit of a cached result may depend on what else is in the cache (e.g. ADMS [10]). Dynamat [33] uses sophisticated cache replacement techniques, specifically computing benefits of cached results taking other cache contents into account. However, their techniques are restricted to the case where each result can be derived directly from exactly one parent (and indirectly from any ancestor). Our techniques do not have this restriction.

In earlier work, usage statistics are maintained for each cached result, which are used to compute a replacement metric for the same; the replacement metric is variously taken as the cached results last use, its frequency of use in a given window, its rate of use, etc. Our techniques do not maintain statistics at the granularity of the cached result – instead, the statistics maintained at the granularity of the queries are used to decide on admission and replacement of the intermediate

results.

Furthermore, in earlier work that considers general queries (e.g. WatchMan [49]), the cached results are matched *syntactically*. Our work carries out sematic matching of cached results during cache-aware query optimization.

It is important to contrast the caching problem with the materialized view/index selection problem, where the cache contents do not vary and the query workload is known fully apriori (e.g., see [44, 34, 26] for general views, [29, 27, 57] for data cubes, and [9] for index selection). Techniques for materialized view/index selection use sophisticated ways of deciding what to materialize, where the computation of the benefit of materializing a view takes into account what other views are materialized. The major disadvantage of static cache contents is that they cannot cater to changing workloads — the data access patterns of the queries cannot be expected to be static, and to answer all types of queries efficiently, we need to dynamically change the cache contents. Moreover, the cost of materializing the selected views is ignored.

Another related area is multi-query optimization (MQO), where (e.g., the work presented in Chapter 3) the optimizer takes the cost of temporarily materializing the selected views, but still makes a static decision on what to materialize based on a fixed set of queries. Still, as we saw in Section 4.3, dynamic cache management can benefit from some of the techniques developed for the efficient implementation of MQO. In particular, the Greedy algorithm presented in Section 4.3 is derived from the Greedy algorithm used in our earlier work on MQO (Chapter 3). However, that algorithm was concerned with minimizing the total one-time execution cost of the queries in a given batch, with no restriction on the storage space. The Greedy algorithm presented in Section 4.3, on the other hand, is concerned with minimizing the cost of an *infinite* workload, where each query can occur multiple times, under fixed constraints on the storage space for cached results. This leads to a very different notion of the "benefit" of sharing a result. Apart from this, a major design issue in this work is to make Greedy suitable for online operation, as is apparent from our discussion in Section 4.3.

Recently, there has been some interest in caching in context of LDAP queries [31]; these queries are simple in nature and involve only multi-attribute selects on a single table. The caching algorithm proposed in [31] performs complete reorganization of the cache contents (called *rev-*

olution) whenever the estimated benefit of the cached data drops below a dynamically estimated value. In between revolutions, the cache contents undergo incremental modifications (called *evolution*). Exchequer performs only evolution; our experiences with performing revolutions as well are presented in Section 4.6.

4.5 Experimental Evaluation of the Algorithms

In this section we describe our experimental setup and the results obtained. Our algorithms were implemented as extensions of the multi-query optimization code (Chapter 3) that we have integrated into our Volcano-based query optimizer. The basic optimizer took approx. 17,000 lines of C++ code, with caching code taking about 3,000 lines.

The block size was taken as 4KB and our cost functions assume 6MB is available to each operator during execution (we also conducted experiments with memory sizes up to 128 MB, with similar results). Standard techniques were used for estimating costs, using statistics about relations. The cost estimates contain an I/O component and a CPU component, with seek time as 10 msec, transfer time of 2 msec/block for read and 4 msec/block for write, and CPU cost of 0.2 msec/block of data processed. We assume that intermediate results are pipelined to the next input, using an iterator model as in Volcano. Caching a result has the cost of writing out the result sequentially to the disk.

The tests were performed on a Sun workstation with UltraSparc 10 333Mhz processor, 256MB RAM, running Solaris 2.7.

4.5.1 Test Query Sequences

We tested our algorithms with streams of 1000 randomly generated queries on a TPCD-based star schema similar to the one proposed by [50]. The schema has a central *Orders* fact table, and four dimension tables *Part, Supplier, Customer* and *Time*. The size of each of these tables is the same as that of the corresponding table in the 100 MB TPCD-0.1 database. This corresponds to base data size of approximately 40 MB (there are other tables in the TPCD-0.1 database which

account for the remaining 60MB). Each generated query was of the form:

```
SELECT SUM(QUANTITY)
FROM ORDERS, SUPPLIER, PART, CUSTOMER, TIME
WHERE join-list AND select-list
GROUP BY groupby-list;
```

The *join-list* enforces equality between attributes of the order fact table and primary keys of the dimension tables. We pick {*suppkey, partkey, custkey, month, year*} as the set of group-by attributes G. An additional attribute from each of PART, SUPPLIER and CUSTOMER was picked to form the list of select attributes A.

The *groupby-list* was generated by picking a subset of G at random. The *select-list*, i.e. the predicates for the selects, were generated by selecting attributes at random from A and G and creating equality or inequality predicates on these attributes using random values picked from the respective domains. The select predicates involving attribues in A define different cubes. Thus, in effect, the workload models simultaneous analysis of a large number of distinct cubes. A query is thus defined uniquely by the pair *(select-list, groupby-list)*. Even though our algorithms can handle a more general class of queries, the above class of cube queries was chosen so that we can have a fair comparison with DynaMat [33] and Watchman2 [50].

There are two independent criteria based on which the pair *(select-list, groupby-list)* was generated.

1. *The kind of predicates comprising the select-list.*

   Accordingly, we classify the workloads as:

   - *CubePoints:* Predicates are restricted to equalities, or
   - *CubeSlices:* Predicates are a random mix of equalities and inequalities.

   Figure 4.4 gives the distribution of the distinct intermediates results computed during the processing of the CubePoints and CubeSlices workloads. Since each predicate in Cube-Points is a highly selective equality, the size of most intermediate results is small, at most

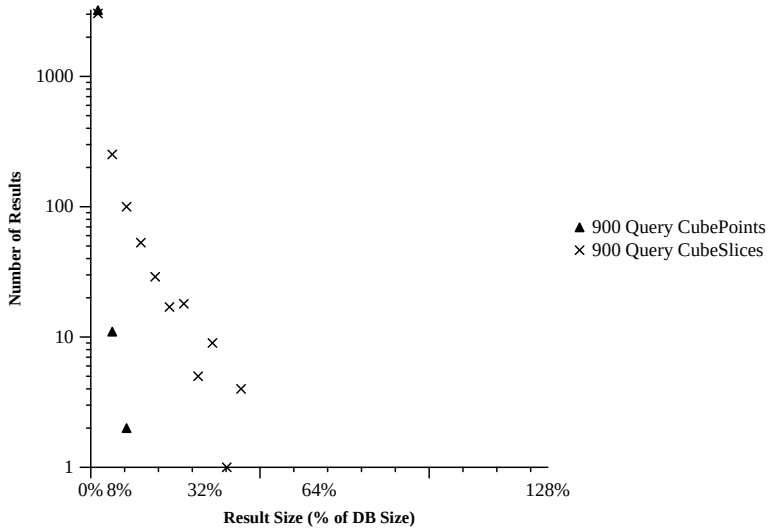


Figure 4.4: Distribution of distinct intermediate results generated during the processing of the CubePoints and CubeSlices workloads

10% of the database size. On the other hand, since CubeSlices contains inequalities as well, a number of larger intermediate results, with size upto 40% of the database size, are also present.

2. *The distribution from which the attributes and values are picked up in order to form the* groupby-list *and the predicates in the select-list.*

   We consider a moderately skewed and a highly skewed workload, based on the Zipfian distribution:[5]

   - *Zipf-0.5:* Uses Zipfian distribution with parameter 0.5. This workload is moderately skewed.
   - *Zipf-2.0:* Uses Zipfian distribution with parameter 2.0. This workload is highly skewed.

   The distribution additionally rotates after every interval of 128 queries, i.e. the most frequent subset of groupbys becomes the least frequent, and all the rest shift up one position.

[5]Zipfian distribution with parameter α on $\{1, \ldots, n\}$ specifies $p(k) \propto k^{-\alpha}$

Thus, within each block of 128 queries, some groupby combinations and selection constants are more likely to occur than others.

Based on the four combinations that result from the above criteria, the following four workloads are considered in the experiments:

- *CubePoints/Zipf-0.5:* a moderately skewed workload of CubePoints,
- *CubePoints/Zipf-2.0:* a highly skewed workload of CubePoints,
- *CubeSlices/Zipf-0.5:* a moderately skewed workload of CubeSlices, and
- *CubeSlices/Zipf-2.0:* a highly skewed workload of CubeSlices

4.5.2 Metric

The metric used to compare the goodness of caching algorithms is the *total response time of a set of queries*. We report the total response time for a sequence of 900 queries that enter the system after a sequence of 100 queries warm up the cache. This total response time is as estimated by the optimizer and hence denoted as *estimated cost* in the experimental results presented in Section 4.5.4. These estimates are the same as used in Section 3.6 and as demonstrated there, are a close approximation to the real execution costs on Microsoft SQL-Server 6.5.

4.5.3 List of algorithms compared

We consider the following three variants of Exchequer; the first two were described in Section 4.3:

- **Exchequer/FullCache:** Apart from computed intermediate results that are included in $\mathcal{S}$, other computed results are also admitted to the cache if there is enough free space in the cache. This is the variant actually used in the Exchequer system.
- **Exchequer/NoFullCache:** Only computed intermediate results that are included in the candidate set $\mathcal{S}$ are added to the cache; no additional nodes are admitted even if there is space in the cache.

- **Exchequer/FinalRes** Identical to Exchequer/FullCache, except that only the final results are cached. This variant is considered to illustrate the impact of caching intermediate results.

The size of the representative set is set to 10 for each of these variants. As a part of the experimental study in Section 4.5.4, we evaluate these variants against each other as well as against the following prior approaches.

- **LCS/LRU:** This approach uses the caching policy found to be the best in ADMS [10], namely replacing the result occupying the *largest cache space* (LCS), picking the *least recently used* (LRU) result in case of a tie. The incoming query is optimized taking the cache contents into account. The final as well as intermediate results in the best plan are considered for admission into the cache based on LCS.

- **DynaMat:** We simulate DynaMat [33] by considering only the top-level query results (in order to be fair to DynaMat, our benchmark queries were chosen to have either no selection or only single value selections). The original DynaMat performs matching of cube slices using R-trees on the dimension space. In our implementation, query matching is performed semantically, using our unification algorithm, rather than syntactically. We use our algorithms to optimize the query taking into account the current cache contents; this covers the subsumption dependency relationships explicitly maintained in [33]. The replacement metric is computed as:

  (number-of-accesses $*$ cost-of-computation)/(query-result-size)

  where the number of accesses are from the entire history (observed so far).

- **WatchMan:** Watchman [49] also considers caching only the top level query results. The original Watchman does syntactic matching of queries, with semantic matching left for future work. We improve on that by considering semantic matching. The difference between our implementation of DynaMat and WatchMan is in the replacement metric: instead of considering the number of accesses as in the Dynamat implementation, our WatchMan implementation considers the rate of use on a window of last five accesses for each query.

The replacement metric for Watchman is thus:

(rate-of-use $*$ cost-of-computation)/(query-result-size)

where the cost of computation is with respect to the current cache contents. The original algorithms did not consider subsumption dependencies between the queries; our implementation considers aggregation subsumption among the cube queries considered.

Given the enhancements mentioned above, our implementations of the above algorithms are slightly more sophisticated than the originally proposed versions.

It is important to investigate the promise dynamic materialized view selection hold over static materialized view selection. In order to do so, we consider our version of static view selection wizard as follows:

- **Static:** We use Exchequer/NoFullCache on the first 100 queries in the workload, with the representative set consisting of all queries so far. After the 100^{th} query, the cache contents are fixed and never changed in the duration of the remaining workload. The cost of computing the materialized views is not added in the execution cost of the workload.

In order to evaluate the absolute benefits and competitivity of the algorithms considered. we also consider the following baseline approaches:

- **NoCache:** Queries are run assuming that there is no cache. This gives an upper bound on the running time of any well-behaved caching algorithm.

- **InfCache:** The purpose of this simulation is to give a lower bound on the running time of any caching algorithm. We assume an infinite cache and *do not* include the materialization cost. Each new result is computed and cached the first time it occurs, and reused whenever it occurs later.

4.5.4 Experimental Results

The goal of this section is to study the following issues:

1. Merit of intermediate result caching over exclusively final result caching.

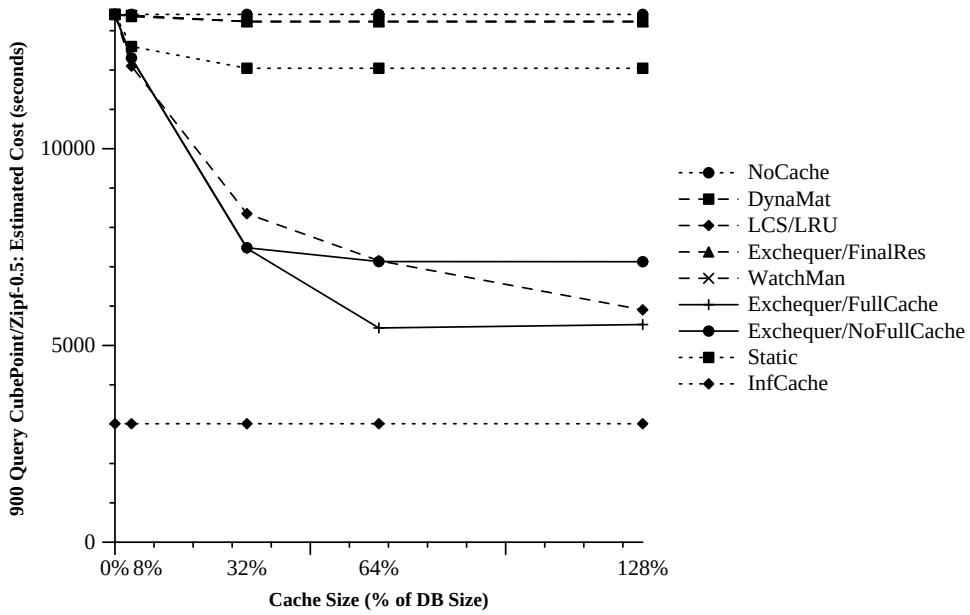


Figure 4.5: Performance on 900 Query CubePoints/Zipf-0.5 Workload

2. Merit of dynamic intermediate result caching over static result caching, for moderately and highly skewed workloads.

3. Merit of cost-benefit based approach over simpler policies like LCS/LRU.

4. Merit of keeping the cache full by caching additional results in case the results selected by greedy do not fill up the entire cache (as in Exchequer/FullCache) over caching only the results selected by greedy, as in Exchequer/NoFullCache).

5. Whether the overheads incurred by Exchequer/FullCache are acceptable.

We experiment with different cache sizes, corresponding to roughly 0%, 32% and 64% and 128% of the total database size of approximately 40 MB. For each of these cache sizes, the set of 9 algorithms mentioned in Section 4.5.3 (viz. NoCache, DynaMat, LCS/LRU, WatchMan, Exchequer/FinalRes, Exchequer/FullCache, Exchequer/NoFullCache, Static and InfCache) were executed on the four workloads listed in Section 4.5.1. The results for CubePoints/Zipf-0.5 and CubePoints/Zipf-2.0 workloads are shown in Figure 4.5 and Figure 4.6 respectively, while the results for CubeSlices/Zipf-0.5 and CubeSlices/Zipf-2.0 are shown in Figure 4.7 and Figure 4.8

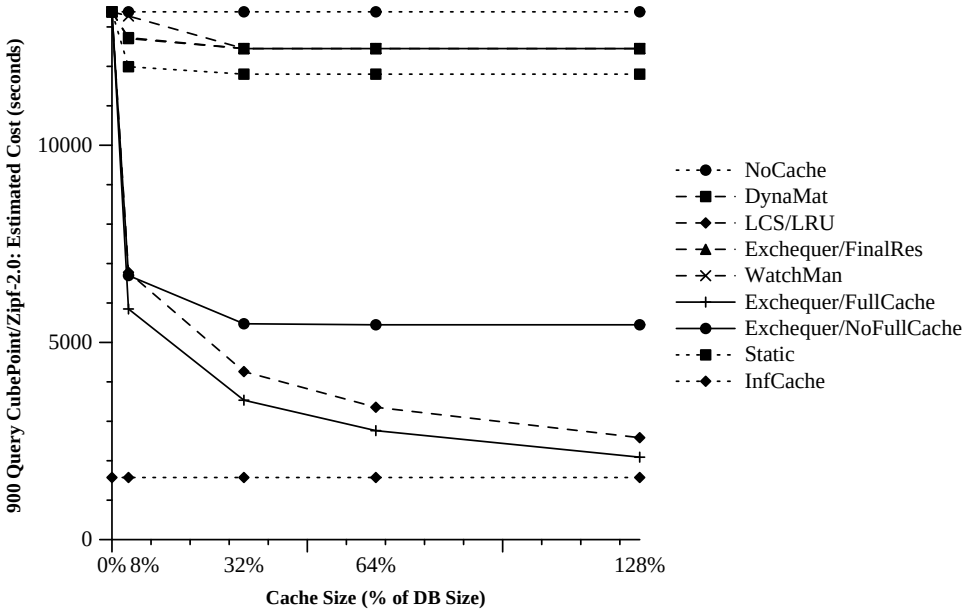


Figure 4.6: Performance on 900 Query CubePoints/Zipf-2.0 Workload

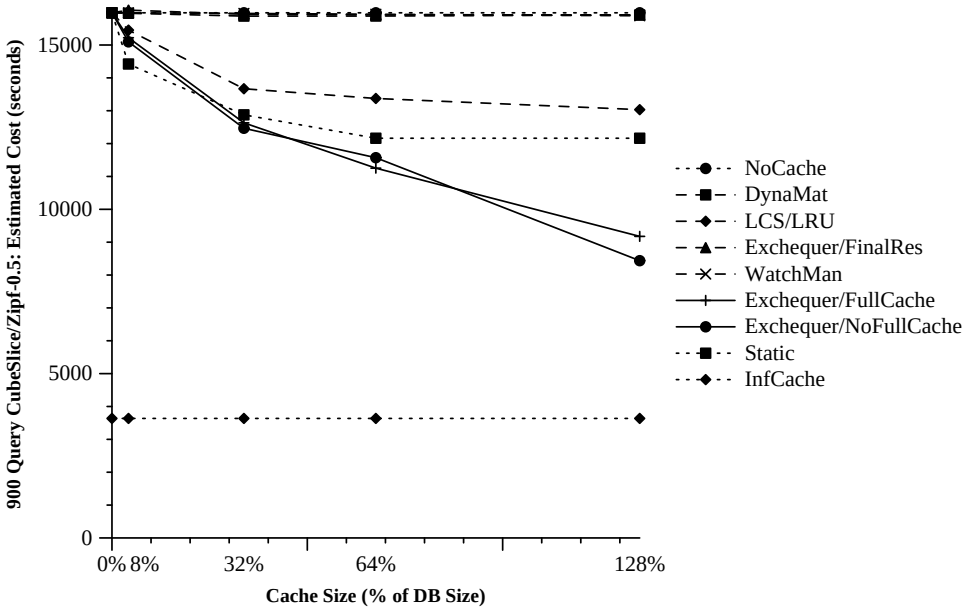


Figure 4.7: Performance on 900 Query CubeSlices/Zipf-0.5 Workload

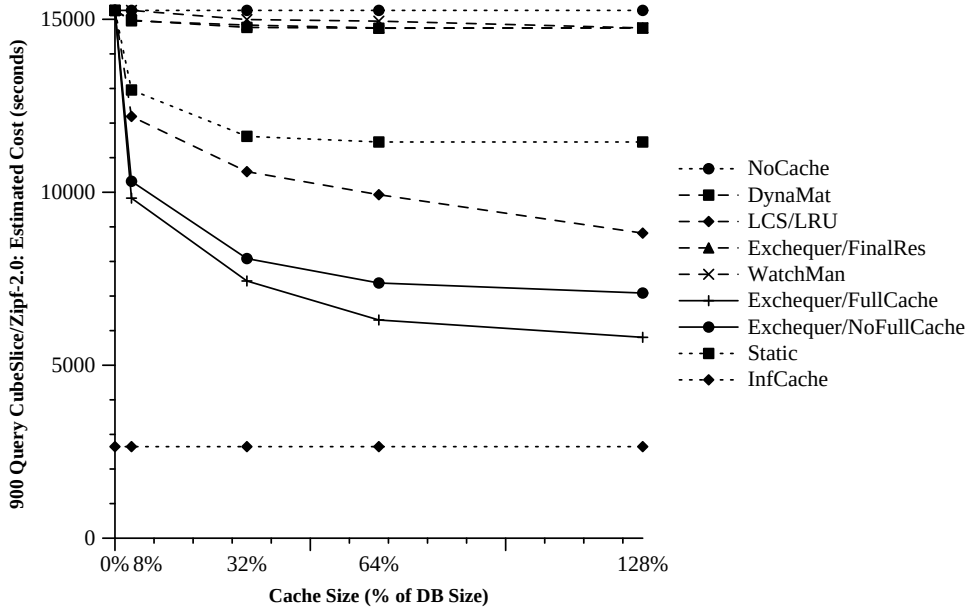


Figure 4.8: Performance on 900 Query CubeSlices/Zipf-2.0 Workload

respectively.

Effect of Intermediate Result Caching. For all the four workloads, DynaMat, WatchMan and Exchequer/FinalRes which cache only the full query results perform very poorly. This is because though there is a large amount of overlap among the queries in each workload, there is hardly any repetition of the same query. In fact, because of the select predicates involving the set A (ref. Section 4.5.1), the subsumption possibilities among the results (that can be exploited by these algorithms) are minimal.

The importance of intermediate result caching can be gauged by the fact that even Static, which maintains a *fixed* set of intermediate results, consistently performs far better than these algorithms. This is because the intermediate results cached by static, though fixed, can be used by a greater number of queries in the workload. This clearly demonstrates the heavy improvement in performance that can be achieved using intermediate result caching.

Effect of Dynamic Caching. We now compare the performance of Static with that of the algorithms which dynamically maintain the cached results, viz. LCS/LRU, Exchequer/NoFullCache

and Exchequer/FullCache.

Recall that Static builds up the cache contents using the query distribution of the first 100 queries, and keeps it fixed for the duration of the remaining 900 queries. However, each of the workloads changes the skew after every 128 queries, making the caching decisions by Static mostly ineffective. Naturally, therefore, we find that these dynamic intermediate result caching algorithms consistently perform much better than Static for all the workloads considered, with the sole exception of CubeSlices/Zipf-0.5.

In the case of CubeSlices/Zipf-0.5, Static performs better than LCS/LRU for the whole range of cache sizes considered. This is because CubeSlices/Zipf-0.5 workload contains large intermediate results with high benefit due to subsumption. While Static caches these results, LCS/LRU does not because of its bias against larger results. Surprisingly, for small cache sizes on the CubeSlices/Zipf-0.5 workload, even Exchequer/NoFullCache and Exchequer/FullCache perform better than Static. This is because for small cache sizes, these large cache results lead to significant overheads due to their repeated materialization and disposal in the dynamic algorithms, and the fixed caching approach of Static holds an advantage. However, for larger cache sizes, Exchequer/NoFullCache and Exchequer/FullCache are able to maintain these larger results in the cache longer, leading to the sharp gain in performance over Static.

Thus, overall, we conclude that dynamic intermediate can lead to large improvements over static caching. For consistent behaviour, however, it is important that the intermediate result caching policy be intelligent, taking into account the cost versus benefit of caching the results, unlike LCS/LRU. This is further discussed next.

Need for Cost-Benefit Based Algorithms. We now compare the sophisticated approach of Exchequer/FullCache, with the much simpler approach of LCS/LRU. We find that while Exchequer/FullCache performs very well for all the four workloads, the relative performance of LCS/LRU varies from very good to poor (even worse than Static), markedly depending upon the distribution of the intermediate results (ref. Figure 4.4) and the skew of the workload.

On the CubePoints workloads (both Zipf-0.5 and Zipf-2.0), LCS/LRU performs extremely well; in fact its performance is close to that of Exchequer/FullCache for this workload. This

is because the size of the intermediate results in these workloads is small; moreover, because of the predicates being exclusively equalities, subsumption plays little role and therefore larger results have small benefit given the space they occupy. Thus, on these workloads, the LCS/LRU strategy of preferably caching smaller results pays well, and the advantage due to occasional high benefit larger results cached by Exchequer/FullCache is not much. Thus, for the workloads having small intermediate results and low subsumption opportunities, the benefits offered by the more sophisticated Exchequer/FullCache over much simpler LCS/LRU are modest. On the CubeSlices workloads, however, Exchequer/FullCache performs much better than LCS/LRU. This is because, due to subsumption, the larger results have a higher benefit, but LCS/LRU preferentially maintains smaller results in cache.

LCS/LRU works on the assumption that smaller intermediate results have high benefit. In the cases when this assumption is satisfied, the performance of LCS/LRU is almost as well as Exchequer/FullCache. However, in case this assumption does not hold and larger intermediate results have greater benefit, LCS/LRU does not perform well. Exchequer/FullCache explicitly takes into account the costs and benefits of intermediate results while taking the caching decisions and, unlike LCS/LRU, does not rely on an ad-hoc rule. This makes it much less sensitive to the size of intermediate results, and it performs much better than other earlier algorithms on all the four workloads. Thus, at the cost of the extra sophistication, Exchequer/FullCache gives a performance that is not only better, but is much more stable than that given by the simpler LCS/LRU.

Effect of Caching Additional Results in Available Extra Space. The two variants of the basic Exchequer algorithm, Exchequer/NoFullCache and Exchequer/FullCache, differ in the decision about whether or not to make extra investments by caching additional results in the cache that may remain unfilled after all the results selected by Greedy are cached; this extra space is managed using LCS/LRU. Exchequer/FullCache makes this investment expecting to benefit in the future due to having more results in the cache. On the other hand, Exchequer/NoFullCache is more conservative and does not make this investment.

Our results show that Exchequer/FullCache benefits significantly in performance over Ex-

chequer/NoFullCache by making use of the extra cache space. There are instances when the investment does not pay off, as in the case of CubePoints/Zipf-0.5 for the cache size of 128%, and the performance actually deteriorates. But this occasional loss is neglegible as compared to the benefits obtained, as can be seen by comparing the graphs of Exchequer/FullCache and Exchequer/NoFullCache for all the four workloads.

It may be argued that since Exchequer/NoFullCache selects results for caching after carefully weighing their benefits against their costs, the extra benefit due to caching additional results should be minimal. However, the accuracy of these benefits depends on the how accurately the past workload estimates the future workload (ref. Section 4.2). In face of sudden changes in the workload skew (recall that each of our workloads changes skew after a block of 128 queries), the estimate may be inaccurate for a certain transient period. During this period, therefore, the benefit may not be accurate. Caching additional results reduces the impact of such occasional inaccuracies, and makes the caching policy more stable.

Space and Time Overheads. As an estimate of the memory overhead of the Exchequer algorithm, we determined the space taken by CDAG during the execution of the Exchequer algorithm; recall that the CDAG includes the best plans for the 10 queries in the representative set, the expanded DAG for the current query, and the best plans for the results currently in the cache. For the run of Exchequer/FullCache on the CubeSlices/Zipf-2.0 workload, the maximum size of CDAG was approximately 23M of memory, and was independent of the cache size.

The time taken by Exchequer/FullCache depends on the cache size since the Greedy algorithm (ref. Section 4.3) chooses results only till their size does not exceed the cache size. The table below shows the average optimization costs and optimization times per query for Exchequer/FullCache on the 900 query CubeSlices/Zipf-2.0 workload for different cache sizes; the corresponding numbers for other workloads are similar.

Metric	Cache Size (% of DB Size)				
	0%	8%	32%	64%	128%
Avg. Optimization Time/Query (secs)	0.16	1.01	1.18	1.22	1.05
Avg. Estimated Cost/Query (secs)	16.95	10.92	8.26	7.00	6.45

As we can see, the cost of optimization and cache management using Exchequer/FullCache is *an order of magnitude* less than the execution cost of the workload (the ratio can be expected to be even less on datasets larger than TPC-0.1), thus showing that the optimization of queries and cache management in Exchequer has negligible overhead.

4.6 Extensions

We have developed several extensions of our techniques, which we outline below.

We implemented a version of the Exchequer algorithm with periodic reorganization, which is similar to revolution [31]. This involved invoking Greedy with the candidate set containing all results in the best plan of each query in the Representative Set. However, for reasonably complex queries involving joins this leads to a large candidate set, and thus the reorganization step is very expensive. In many cases, this led to poor gains at a high cost. Therefore, we abandoned this strategy.

The Exhequer system described in this chapter supports only disk-caching. However, the techniques described can be extended for *main-memory* caching and *hybrid* (disk cum main-memory) caching. A main-memory caching system contains a fixed size area in memory allocated as the cache. The modification is restricted to the cost-model – there is no I/O overhead for caching results or for using them; the techniques as presented in this chapter remain unchanged. A hybrid caching system contains (a) a fixed size area in memory allocated as the main-memory cache, as well as (b) a fixed size area on disk allocated as the disk cache. We modify the Greedy algorithm to work in two phases: the first phase fills up the main-memory cache, while the second phase fills up the disk cache, choosing results from those that remain in the candidate set after the first phase is over. The two phases are identical in all respects, except that results in the first phase are chosen using the main-memory based cost model (no I/O overhead for caching or use of cached results), while the results in the second phase are chosen using the disk based cost model (same as considered in this chapter).

4.7 Summary

In this chapter we have presented new techniques for query result caching, which can help speed up query processing in data warehouses. The novel features incorporated in our Exchequer system include optimization aware cache maintenance and the use of a cache aware optimizer. In contrast, in existing work, the module that makes cost-benefit decisions is part of the cache manager and works independent of the optimizer which essentially reconsiders these decisions while finding the best plan for a query. Whereas existing approaches are either restricted to cube (slice/point) queries, or cache just the query results, our work presents a data-model independent framework and algorithm. Our experimental results attest to the efficacy of our cache management techniques.

Chapter 5

Materialized View Maintenance and Selection

Materialized views have been found to be very effective in speeding up query, as well as update processing, and are increasingly being supported by commercial database systems. Materialized views are especially attractive in data warehousing environments because of the query intensive nature of data warehouses. However, when a warehouse is updated, the materialized views must also be updated. Typically, updates are accumulated and then applied to a data warehouse. While the need to provide up-to-date responses to an increasing query load is growing and the amount of data that gets added to data warehouses has been increasing, the time window available for making the warehouse up-to-date has been shrinking. These trends call for efficient techniques for maintaining the materialized views as and when the warehouse is updated.

The view maintenance problem can be seen as computing the expressions corresponding to the "delta" of the views, given the "delta"s of the base relations that are used to define the views. It is not difficult to motivate that query optimization techniques are important for choosing an efficient plan for maintaining a view, as shown in [61]. For example, consider the materialized view $V = (A \bowtie B \bowtie C)$. We assume, as in SQL, that relations A, B and C are multisets (i.e., relations with duplicates). Given that the multiset of tuples δ_C^+ is inserted into C, the change to the materialized view V consists of a set of tuples $(A \bowtie B) \bowtie \delta_C^+$ to be inserted into V. This

expression can equivalently be computed as $(A \bowtie \delta_C^+) \bowtie B$ and by $(B \bowtie \delta_C^+) \bowtie A$, one of which may be substantially cheaper to compute. Further, in some cases the view may be best maintained by recomputing it, rather than by finding the differentials as above.

Our work addresses the problem of optimizing the maintenance of a *set* of materialized views. If there are multiple materialized views, as is common, significant opportunities exist for sharing computation between the maintenance of different views. Specifically, common subexpressions between the view maintenance expressions can reduce maintenance costs greatly.

Whether or not there are multiple materialized views, significant benefits can be had in many cases by materializing extra views or indices, whose presence can decrease maintenance costs significantly. The choice of what to materialize permanently depends on the choice of view maintenance plans, and vice versa. The choices of the two must therefore be closely coupled to get the best overall maintenance plans.

Contributions. The contributions of this work lie in optimization of the view maintenance plans. Specifically, the contributions are as follows.

1. *We show how to exploit transient materialization of common subexpressions to reduce the cost of view maintenance plans.*

   Sharing of subexpressions occurs when multiple views are being maintained, since related views may share subexpressions, and as a result the maintenance expressions may also be shared. Furthermore, sharing can occur even within the plan for maintaining a single view if the view has common subexpressions within itself.

   The shared expressions could include differential expressions, as well as full expressions which are being recomputed.

   Here, *transient materialization* means that these results are materialized during the evaluation of the maintenance plan and disposed on its completion.

2. *We show how to efficiently choose additional expressions for permanent materialization to speed up maintenance of the given views.*

Just as the presence of views allows queries to be evaluated more efficiently, the maintenance of the given permanently materialized views can be made more efficient by the presence of additional permanently materialized views [45, 44]. That is, given a set of materialized views to be maintained, we choose additional views to materialize in order to minimize the overall view maintenance costs.

The expressions chosen for permanent materialization may be used in only one view maintenance plan, or may be shared between different views maintenance plans. We outline differences between our work and prior work in this area, in Section 5.1.

3. *We show how to determine the optimal maintenance plan for each individual view, given the choice of results for transient/permanent materialization*.

   Maintenance of a materialized view can either be done *incrementally* or *by recomputation*. Incremental view maintenance involves computing the differential ("delta"s) of a materialized view, given the "delta"s of the base relations that are used to define the views, and merging it with the old value of the view. However, incremental view maintenance may not always be the best way to maintain a materialized view; when the deltas are large the view may be best maintained by recomputing it from the updated base relations.

   Our techniques determine the maintenance policy, incremental or recomputation, for each view in the given set such that the overall combination has the minimum cost.

4. *We show how to make the above three choices in an integrated manner to minimize the overall cost.*

   It is important to point out that the above three choices are highly interdependent, and must be taken in such a way that the overall costs of maintaining a set of views is minimized. Specifically:

   - Given a subexpression useful during materialization of multiple views, choosing whether it should be transiently or permanently materialized is an optimization problem, since each alternative has its cost and benefit. Transient views are materialized

during the evaluation of the maintenance plan and discarded after maintenance of the given views; such transient views themselves need not be maintained. On the other hand, the permanent views are materialized a priori, so there is no (re)computation cost; however, there is a maintenance cost, and a storage cost (which is long term in that it persists beyond the view maintenance period) due to the permanently materialized views.

- The choice of additional views must be done in conjunction with selecting the plans for maintaining the views, as discussed above. For instance, a plan that seems quite inefficient could become the best plan if some intermediate result of the plan is chosen to be materialized and maintained.

We propose a framework that cleanly integrates the choice of additional views to be transiently or permanently materialized, the choice of whether each of the given set of (user-specified) views must be maintained incrementally or by recomputation, and the choice of view maintenance plans.

5. We have implemented all our algorithms, and present a performance study, using queries from the TPC-D benchmark, showing the practical benefits of our techniques.

Our contributions go beyond the existing state of the art in several ways:

1. Earlier work on selecting views for materialization addresses either transient view selection (for multi-query optimization, but not for view maintenance) without considering permanent view selection, or permanent view selection, without considering transient view selection. Neither approach is integrated with the choice of view maintenance plans. To the best of our knowledge, ours is the first work that addresses the above aspects *simultaneously*, taking into account the intricate interdependence of the decisions. Making the decisions separately may lead to a non-optimal choice. See Section 5.1 for more details of related work.

   Moreover, as far as we know, the problem of automatically selecting the optimum maintenance policy for a materialized view in the presence of other materialized views has not

been addressed earlier. This is a major step beyond the current state-of-the-art in research or practice. For example, in Oracle 8i [5], a user has to specify a materialized view's maintenance policy during its definition in an ad-hoc manner.

2. Earlier work on transient materialization (done in the context of multiquery optimization) is not coupled with view maintenance. While those algorithms can be used directly on view maintenance expressions to decide on transient view materialization, using them naively would lead to very poor performance. We show how to integrate view maintenance choices into an optimizer in a way that leads to very good performance.

3. We have shown the practicality of our work by implementing all our algorithms and presenting a performance study illustrating the benefits to be had by using our techniques. Earlier work does not cover efficient techniques for the implementation of materialized view selection algorithms. Moreover, our implementation is built on top of an existing state-of-the-art query optimizer, showing the practicality of using our techniques on existing database systems.

Our performance study, detailed in Section 5.6 shows that significant benefits, often by factors of 2 or more, can be obtained using our techniques.

Although the focus of our work is to speed up view maintenance, and we assume an initial set of views have been chosen to be materialized, our algorithms can also be used to choose extra materialized views to speed up a workload containing queries and updates.

Paper Organization. Related work is outlined in Section 5.1. Section 5.2 gives an overview of the techniques presented in this chapter. Section 5.3 describes our system model, and how the search space of the maintenance plans is set up. Section 5.4 shows how to compute the optimal maintenance cost for a given set of permanently materialized views, and a given set of views to be transiently materialized during the maintenance. Section 5.5 describes a heuristic that uses this cost calculation to determine the set of views to be transiently or permanently materialized so as to minimize the overall maintenance cost. Section 5.6 outlines results of a performance study, and Section 5.7 presents a summary of the chapter.

5.1 Related Work

In the past decade, there has been a large volume of research on view maintenance, transiently materialized view selection (also known as multi-query optimization) and also on permanently materialized view selection. This work is summarized below. However, each of these problems have been addressed independently since the concerns are orthogonal; no prior work, to the best of our knowledge, has looked at addressing all of these problems in an integrated manner.

View Maintenance Amongst the early work on computing the differential results of operations and expressions was Blakeley et al. [3]. More recent work in this area includes [24, 12, 37, 36] and [48]. Gupta and Mumick [25] provide a survey of view maintenance techniques.

Vista [61] describes how to extend the Volcano query optimizer to compute the best maintenance plan, but does not consider the materialization of expressions, whether transient or permanent. [42] and [61] propose optimizations that exploit knowledge of foreign key dependencies to detect that certain join results involving differentials will be empty. Such optimizations are orthogonal and complementary to our work.

Transiently Materialized View Selection (Multi-Query Optimization) Blakeley et al. [3] and Ross et al. [44] noted that the computation of the expression differentials has the potential for benefiting from multi-query optimization. In the past, multi-query optimization was viewed as too expensive for practical use. As a result they did not go beyond stating that multi-query optimization could be useful for view maintenance.

Early work on multi-query optimization includes [54, 56, 53]. More recently [59] and [47] (Chapter 3 of this thesis) considered how to perform multi-query optimization by selecting subexpressions for transient materialization, and showed that multiquery optimization is practical and can give significant performance benefits at acceptable cost.

However, none of the work on multi-query optimization considers updates or view maintenance, which is the focus of this chapter. Using these techniques naively on differential maintenance expressions would be very expensive, since incremental maintenance expressions can

be very large. We utilize the optimizations proposed in Chapter 3 but significant extensions are required to to take update costs into account, and to efficiently optimize view maintenance expressions.

Permanently Materialized View Selection There has been much work on selection of views to be materialized. One notable early work in this area was by Roussopolous [45]. Ross et al. [44] considered the selection of extra materialized views to optimize maintenance of other materialized views/assertions, and mention some heuristics. Labio et al. [34] provide further heuristics. The problem of materialized view selection for data cubes has seen much work, such as [29], who propose a greedy heuristic for the problem. Gupta [26] and Gupta and Mumick [28] extend some of these ideas to a wider class of queries.

The major differences between our work and the above work on materialized view selection can be summarized as follows:

1. Earlier work in this area has not addressed optimization of view maintenance plans in the presence of other materialized views. Earlier work simply assumes that the cost of view maintenance for a given set of materialized views can be computed, without providing any details.

2. Earlier work does not consider how to exploit common subexpressions by temporarily materializing them because of their focus on permanent materialization. In particular, common subexpressions involving differential relations cannot be permanently materialized.

3. Earlier work does not cover efficient techniques for the implementation of materialized view selection algorithms, and their integration into state-of-the-art query optimizers. Showing how to do the above is amongst our important contributions.

5.2 Overview of Our Approach

We extend the Volcano query optimization framework [23] to generate optimal maintenance plans. This involves the following subproblems:

1. *Setting up the Search Space of Maintenance Plans*

   We extend the Query DAG representation (ref. Chapter 2), which represents just the space of recomputation plans, to include the space of incremental plans as well. This new extension uses *propagation-based differential generation*, which propagates the effect of one delta relation at a time in a predefined order. Our approach has a lower space cost of optimization as compared to using incremental view maintenance expressions, and is easier to implement.

   Propagation-based differential generation is explained in Section 5.3.2, and the extended Query DAG generation is explained in Section 5.3.3.

2. *Choosing the Policy for Maintenance and Computing the Cost of Maintenance*

   We show how to compute the minimum overall maintenance cost of the given set of permanently materialized views, given a fixed set of additional views to be transiently materialized. In addition to computing the cost, the proposed technique generates the best consolidated maintenance plan for the given set of permanently materialized views. The maintenance plan chosen for each materialized view can be incremental or recomputation, based on costs.

   Maintenance cost computation is explained in Section 5.4.

3. *Transient/Permanent Materialized View Selection*

   Finally, we address the problem of determining the respective sets of transient and permanently materialized views that minimize the overall cost. Our technique uses, as a subroutine, the previously mentioned technique for computing the best maintenance policy given fixed sets of permanently and temporarily materialized views. The costs of materialization of transiently materialized views and maintenance of permanently materialized views are taken into account by this step.

   We propose a greedy heuristic that iteratively picks up views in order of benefit – where benefit is defined as the decrease in the overall materialization cost if this view is tran-

siently or permanently materialized in addition to the views already chosen. Then, depending upon whether transient or permanent materialization of the view produces the greater benefit, the view is categorized as such.

The greedy heuristic is presented in Section 5.5.1, and several optimizations of this heuristic that result in an efficient implementation are described in Section 5.5.2.

5.3 Setting up the Maintenance Plan Space

In this section, we describe how the search space of maintenance plans is set up. We start by describing our system model. As mentioned earlier, our approach to incremental maintenance is based on the compact propagation-based differential generation technique; this is described in Section 5.3.2. The extensions to the Query DAG representation, introduced in Section 2.2.2, to compactly represent the search space of view maintenance plans as well, are described in Section 5.3.3.

5.3.1 System Model

We assume that we are given an initial set of permanently materialized views. We may add more views to this set. We do not consider space limitations on storing materialized views in the main part of the chapter, but address this issue in Section 5.5.3.

We assume that the updates (inserts/deletes) to relations are logged in corresponding *delta* relations, which are made available to the view refresh mechanism; for each relation R, there are two relations δ_R^+ and δ_R^- denoting, respectively, the (multiset of) tuples inserted into and deleted from the relation R. The maintenance expressions in our examples assume that the old value of the relation is available, but we can use maintenance expressions based on the new values of the relations in case the updates have already been performed on the base relations.

We assume that the given set of materialized views is refreshed at times chosen by users, which are typically regular intervals. For optimization purposes, we need estimates of the sizes of these delta relations. In production environments, the rates of changes are usually stable across

refresh periods, and these rates can be used to make decisions on what relations to materialize permanently. We will assume that the average insert and delete sizes for each relation are provided as percentages of the full relation size. The insert and delete percentages can be different for different relations. Other statistics, such as number of new distinct values for attributes (in each refresh interval), if available, can also be used to improve the cost estimates of the optimizer.

5.3.2 Propagation-Based Differential Generation for Incremental View Maintenance

We generate the differential of an expression by propagating differentials of the base relations up the *expression tree*, one relation at a time, and only one update type (insertions or deletions) at a time. The differential propagation technique we use is based on the techniques used in [45] and [44].

The differential of a node in the tree is computed using the differential (and if necessary, the old value) of its inputs. We start at the leaves of the tree (the base relations), and proceed upwards, computing the differential expressions corresponding to each node.

For instance, the differential of a join $(E_1 \bowtie E_2)$, given inserts on R is computed using the differentials of E_1 and E_2 and the old full results of E_1 and E_2. The differential result is empty if R is used in neither E_1 nor E_2. If R is used only in E_1, the differential is given by $(\delta_{E_1} \bowtie E_2)$; symmetrically if R is used only in E_2, the differential is given by $(E_1 \bowtie \delta_{E_2})$. If R is used in both, the differential consists of $(\delta_{E_1} \bowtie E_2) \cup (E_1 \bowtie \delta_{E_2}) \cup (\delta_{E_1} \bowtie \delta_{E_2})$.

The process of computing differentials starts at the bottom, and proceeds upwards, so when we compute the differential to $(E_1 \bowtie E_2)$, the differentials of the inputs have been computed already. The full results are computed when required, if they are not available already (materialized views and base relations are available already).

Extending the above technique to operations other than join is straightforward, using standard techniques for computing the differentials of operations, such as those in [3]; see [25] for a survey of view maintenance techniques.

It may appear that computing the change to $(E_1 \bowtie E_2)$, given a change to R, requires com-

putation of the entire result of E_2 if R is used in E_1. However, our search space will include differentials of all plans equivalent to $(E_1 \bowtie E_2)$. In the case of joins, in particular, the search space will include plans where every intermediate result includes the differential of R. To illustrate this point, consider the view $(A \bowtie B \bowtie C)$. If we wish to compute the differential of the view when tuples are inserted into A, then the plans $(B \bowtie (\delta_A^+ \bowtie C))$ and $(\delta_A^+ \bowtie (B \bowtie C))$ would both be among the plans considered, and the cheapest plan is selected. Similarly, if we wish to compute the differential of the view when tuples are inserted into B, then the plans $(A \bowtie (\delta_B^+ \bowtie C))$ and $(\delta_B^+ \bowtie (A \bowtie C))$ would be amongst the alternatives. Using the differentials of a single expression, such as $(A \bowtie (B \bowtie C))$ or $(B \bowtie (A \bowtie C))$, is not preferable for propagating all the base relation differentials.

Our optimizer's search space includes all of the alternatives for computing the differentials to $(A \bowtie B \bowtie C)$, including the above two, and the cheapest one is chosen for propagating the differential of each base relation.

Propagating differentials of only one type (inserts or deletes) to one relation at a time, simplifies choosing of a separate plan for each differential propagation. It is straightforward to extend the techniques to permit propagation of inserts and deletes to a single relation together, to reduce the number of different expressions computed.

We assume that the updates to the base relations are propagated one relation at a time. After each one is propagated, the base relation is itself updated, and the computed differentials are applied to all incrementally maintained materialized views.[1] We leave unspecified the order in which the base relations are considered. The order is not expected to have a significant effect when the deltas of all the relations are small percentages of the relation sizes: the relation statistics then do not change greatly due to the updates, and thus the costs of the plans should not be affected greatly by the order. For large deltas, our experimental results show that recomputation of the view is generally preferable to incremental maintenance, so the order of incremental propagation is not relevant.

[1]The differentials must be *logically* applied. The database system can give such a logical view, yet postpone physically applying the updates. By postponing physical application, multiple updates can be gathered and executed at once, reducing disk access costs.

An alternative approach for computing differentials is to generate the entire differential expression, and optimize it (see, e.g. [24]). However, the resultant expression can be very large – exponential in the size of the view expression. For instance, consider the view $(A \bowtie B \bowtie C)$, with inserts on all three relations. The differential in the result of the view can be computed as:

$$(\delta_A^+ \bowtie B \bowtie C) \cup (A \bowtie \delta_B^+ \bowtie C) \cup (A \bowtie B \bowtie \delta_c^+) \cup (A \bowtie \delta_B^+ \bowtie \delta_C^+) \cup$$
$$(\delta_A^+ \bowtie B \bowtie \delta_C^+) \cup (\delta_A^+ \bowtie \delta_B^+ \bowtie C) \cup (\delta_A^+ \bowtie \delta_B^+ \bowtie \delta_C^+)$$

There are many common subexpressions in the above expression, and the above expression could be simplified by factoring, to get:

$$(\delta_A^+ \bowtie B \bowtie C) \cup ((A \cup \delta_A^+) \bowtie \delta_B^+ \bowtie C) \cup ((A \cup \delta_A^+) \bowtie (B \cup \delta_B^+) \bowtie \delta_C^+)$$

This simplified expression is equivalent in effect to our technique for propagating differentials.

Creating differential expressions (whether in the unsimplified or in the simplified form) is difficult with more complex expressions containing operations other than join (see, e.g. [24]). Moreover, the size of the unsimplified expression is exponential in the number of relations. Optimizing such large expressions can be quite expensive, since query optimization is exponential in the size of the expression.

In contrast, the process of propagating differentials can be expressed purely in terms of how to compute the differentials for individual operations, given the differential of their inputs. As a result it is also easy to extend the technique to new operations.

5.3.3 Incorporating Incremental Plans in the Query DAG Representation

Consider a database consisting of n relations: $R_1, \ldots, R_n$. Then, for each equivalence node e in the Query DAG described in Section 2.2.2, we introduce n additional equivalence nodes $\delta_e^1, \ldots, \delta_e^{2n}$, where δ_e^{2i-1} and δ_e^{2i} (for $i = 1, \ldots, n$) correspond to the differentials of e with respect to $\delta_{R_i}^+$ and $\delta_{R_i}^-$ respectively. For example, the equivalence node $e : (R_1 \bowtie R_2)$ is refined into four additional equivalence nodes $\delta_e^1 : (\delta_{R_1}^+ \bowtie R_2)$, $\delta_e^2 : (\delta_{R_1}^- \bowtie R_2)$, $\delta_e^3 : (R_1 \bowtie \delta_{R_2}^+)$ and $\delta_e^4 : (R_1 \bowtie \delta_{R_2}^-)$.

We now describe the structure of δ_e^k, $k = 1..2n$. For each child operation node o of e, there exists a child operation node o^k of δ_e^k, representing the differential of o with respect to the corresponding base relation update. In the example above, consider equivalence node e has a child operation node o which is a join operation; the children of o are the equivalents nodes representing R_1 and R_2. The node δ_e^1 has as its child an operation node o^1 which is a join operation, and the children of o^1 are the equivalence nodes for $\delta_{R_1}^+$ and R_2. The other nodes δ_e^k are similar in structure.[2] As can be seen from the above example, the children of o^k can be full results as well as differentials. The rationale of this construction was given in Section 5.3.2. As also mentioned in that section, the approach is easily extended to other operations.

The equivalence node e represents the full result; but this result varies as successive differentials $\delta_e^1, \ldots, \delta_e^{2n}$ are merged with it. For cost computation purposes, the system keeps an array $L[0..2n]$ with e, where $L[0]$ is the list of logical properties (such as schema and estimated statistics) of the old result and $L[i]$, for $i = 1..2n$, is the list of logical properties of the result after the result has been merged with the differentials given by $\delta_e^1, \ldots, \delta_e^i$.

Space-Efficient Implementation. It might seem that by including all the differential expressions for each equivalence node, we have increased the size of the Query DAG by a factor of $2n$. However, our implementation reduces the cost by piggybacking the differential equivalence and operation nodes on the equivalence and operation nodes in the original Query DAG. These implementation details are explained next; however, for ease of explanation, in the rest of the chapter, we stick to the above logical description.

For space efficiency, the equivalence nodes for each differential are not created separately in our implementation. Instead, each equivalence node e stores an array $D[1..2n]$, where $D[k]$ logically represents the differential equivalence node δ_e^k, and contains: (a) logical properties of the differential result δ_e^k, and (b) the best plan for computing δ_e^k.

If e does not depend on a relation R_i, or if there is no corresponding update, then the logical properties and best plan ((a) and (b) above) for $D[2i-1]$ and $D[2i]$ are set as null. In addition,

[2]The structure is a little more complicated when a relation R is used in both children of a join node, requiring a union of several join operations. The details are straightforward and we omit them for simplicity.

as in the original representation, the equivalence node e stores the best plan for (and cost of) recomputing the entire result of the node after all updates have been made on the base relations.

5.4 Maintenance Cost Computation

In this section, we derive formulae for the total maintenance cost for a set $\mathcal{M}_p$ of views materialized permanently and a set $\mathcal{M}_t$ of views materialized temporarily. The optimizer basically traverses the Query DAG structure, applying these formulae, to find the overall cost.

The set $\mathcal{M}_t$ can have views corresponding to entire results (e.g. $A \bowtie B$), as well as views corresponding to differentials (e.g. $\delta_A^+ \bowtie B$). In contrast, the set $\mathcal{M}_p$ can only have views corresponding to entire results; this is because the differentials are only used during view maintenance.

The computation cost of the equivalence node e, denoted $cost(e|\mathcal{M}_p, \mathcal{M}_t)$, is computed as follows, where $\mathcal{C}(e)$ is the set of children operation nodes of e.

$$cost(e|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} \min_{o \in \mathcal{C}(e)} cost(o|\mathcal{M}_p, \mathcal{M}_t) & \text{if } \mathcal{C}(e) \neq \phi \\ 0 & \text{if } \mathcal{C}(e) = \phi \text{ (i.e. } e \text{ is a relation)} \end{cases}$$

In terms of forming the execution plan, the above equation represents the choice of the operation node with the minimum cost in order to compute the expression corresponding to the equivalence node e.

The computation cost of an operation node o, denoted $cost(o|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$cost(o|\mathcal{M}_p, \mathcal{M}_t) = localcost(o) + \sum_{e \in \mathcal{C}(o)} childcost(e|\mathcal{M}_p, \mathcal{M}_t)$$

where $localcost(o)$ is the "local" cost of the operation o, $\mathcal{C}(o)$ is the set of children equivalence nodes of o, and

$$childcost(e|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} reusecost(e) & \text{if } e \in \mathcal{M}_p \cup \mathcal{M}_t \\ cost(e|\mathcal{M}_p, \mathcal{M}_t) & \text{if } e \notin \mathcal{M}_p \cup \mathcal{M}_t \end{cases}$$

During transient materialization, the view is computed and materialized on the disk for the duration of the maintenance processing. Thus, the cost of transiently materializing a view $e \in$

$\mathcal{M}_t$, denoted by $transmatcost(e|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$transmatcost(e|\mathcal{M}_p, \mathcal{M}_t) = cost(e|\mathcal{M}_p, \mathcal{M}_t) + matcost(e)$$

where $matcost(e)$, is the cost of materializing the view (on disk, assuming materialized views do not fit in memory).

Further, for a given $e \in \mathcal{M}_p$, the cost of recomputing the result from the base relations is $cost(e|\mathcal{M}_p, \mathcal{M}_t)$; and the cost of computing the differential $\delta_e^k, k = 1..2n$ is $cost(\delta_e^k|\mathcal{M}_p, \mathcal{M}_t)$. Let $mergecost(\delta_e^k)$ denote the cost of merging the differential corresponding to δ_e^k with the view after the differentials corresponding to $\delta_e^1, \ldots, \delta_e^{k-1}$ have already been merged. Then, the cost of incrementally maintaining e, denoted $imaintcost(e|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$imaintcost(e|\mathcal{M}_p, \mathcal{M}_t) = \sum_{k=1}^{2n}(cost(\delta_e^k|\mathcal{M}_p, \mathcal{M}_t) + mergecost(\delta_e^k))$$

On the other hand, maintenance by recomputation involves computing the view and materializing it, replacing the old value. The recomputation maintenance cost, denoted by $rmaintcost(e|\mathcal{M}_p, \mathcal{M}_t)$, is:

$$rmaintcost(e|\mathcal{M}_p, \mathcal{M}_t) = cost(e|\mathcal{M}_p, \mathcal{M}_t) + matcost(e)$$

where $matcost(e)$, as before, is the cost of materializing the view.

Notice that $rmaintcost(e|\mathcal{M}_p, \mathcal{M}_t)$ is the same as $transmatcost(e|\mathcal{M}_p, \mathcal{M}_t)$, the cost of transiently materializing e derived above. As such, we do not consider materializing a view permanently and maintaining using recomputation, unless it was already specified as permanently materialized. For, if recomputation is the cheapest way of maintaining a view, we may as well materialize it transiently: keeping it permanently would not help the next round of view maintenance. Thus, the cost of maintaining the permanently materialized view $e \in \mathcal{M}_p$, denoted by $maintcost(e|\mathcal{M}_p, \mathcal{M}_t)$, is as follows, where $\mathcal{M}$ is the set of views given as already materialized in the system.

$$maintcost(e|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} \min(imaintcost(e|\mathcal{M}_p, \mathcal{M}_t), rmaintcost(e|\mathcal{M}_p, \mathcal{M}_t)) \\ \quad \text{if } e \in \mathcal{M} \\ imaintcost(e|\mathcal{M}_p, \mathcal{M}_t) \\ \quad \text{if } e \in \mathcal{M}_p - \mathcal{M} \end{cases}$$

For $e \in \mathcal{M}$, the choice corresponds to selecting the refresh mode – incremental refresh or recomputation – depending on whichever is cheaper.

Thus, the total cost incurred in maintaining the materialized views in $\mathcal{M}_p$ given that the views in $\mathcal{M}_t$ are transiently materialized, denoted $totalcost(\mathcal{M}_p, \mathcal{M}_t)$, is:

$$totalcost(\mathcal{M}_p, \mathcal{M}_t) = \sum_{e \in \mathcal{M}_p} maintcost(e|\mathcal{M}_p, \mathcal{M}_t) + \sum_{e \in \mathcal{M}_t} transmatcost(e|\mathcal{M}_p, \mathcal{M}_t) \quad (5.1)$$

Given the set $\mathcal{M}$ of views given as already materialized in the system, we need to determine the set $\mathcal{M}_p (\supseteq \mathcal{M})$ of views to be permanently materialized, as well as the set of views $\mathcal{M}_t$ to be transiently materialized, such that $totalcost(\mathcal{M}_p, \mathcal{M}_t)$ is minimized. In the next section, we propose a heuristic greedy algorithm to determine $\mathcal{M}_p$ and $\mathcal{M}_t$.

As mentioned earlier, the optimizer performs a depth-first traversal of the Query DAG structure, applying these formulae at each node, to find the overall cost.

5.5 Transient/Permanent Materialized View Selection

We now describe how to integrate the choice of extra materialized views with the choice of best plans for view maintenance. In Section 5.5.1, we present the basic algorithm for selecting the two sets of views for transient and permanent materialization respectivel, followed by a discussion of some optimizations and extensions in Section 5.5.2.

5.5.1 The Basic Greedy Algorithm

Given a set of results $\mathcal{M}_p$ and $\mathcal{M}_t$ already chosen to be respectively permanently and transiently materialized, and a equivalence node x, the benefit of additionally materializing x, $benefit(x|\mathcal{M}_p, \mathcal{M}_t)$, is defined as:

$$benefit(x|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} totalcost(\mathcal{M}_p, \mathcal{M}_t) - \\ \qquad \min(totalcost(\mathcal{M}_p \cup \{x\}, \mathcal{M}_t), totalcost(\mathcal{M}_p, \mathcal{M}_t \cup \{x\})) \\ \qquad \text{if } x \text{ is a full result} \\ totalcost(\mathcal{M}_p, \mathcal{M}_t) - totalcost(\mathcal{M}_p, \mathcal{M}_t \cup \{x\}) \\ \qquad \text{if } x \text{ is a differential} \end{cases}$$

Procedure GREEDY
Input: $\mathcal{M}$, the set of equivalence nodes for the initial materialized views
$\mathcal{C}$, the set of candidate equivalence nodes for materialization
Output: $\mathcal{M}_p$, set of equivalence nodes to be materialized permanently
$\mathcal{M}_t$, set of equivalence nodes to be materialized transiently
Begin
$\mathcal{M}_p = \mathcal{M}$; $\mathcal{M}_t = \phi$
while ($\mathcal{C} \neq \phi$)
L1: Pick the node $x \in \mathcal{C}$ with the highest $benefit(x|\mathcal{M}_p, \mathcal{M}_t)$
if ($benefit(x|\mathcal{M}_p, \mathcal{M}_t) < 0$)
break; /* No further benefits to be had, stop */
if (x is a full result and $maintcost(x|\mathcal{M}_p, \mathcal{M}_t) < transcost(x|\mathcal{M}_p, \mathcal{M}_t)$)
$\mathcal{M}_p = \mathcal{M}_p \cup \{x\}$
else $\mathcal{M}_t = \mathcal{M}_t \cup \{x\}$
$\mathcal{C} = \mathcal{C} - \{x\}$
return $(\mathcal{M}_p, \mathcal{M}_t)$
End

Figure 5.1: The Greedy Algorithm for Selecting Views for Transient/Permanent Materialization

Using Equation (5.1), and since (a) if x is a full result, then for all $e \in \mathcal{M}_p$, $maintcost(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\}) = maintcost(e|\mathcal{M}_p \cup \{x\}, \mathcal{M}_t)$, and (b) for all $e \in \mathcal{M}_t$, $transmatcost(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\}) = transmatcost(e|\mathcal{M}_p \cup \{x\}, \mathcal{M}_t)$, the above can be simplified to:

$$benefit(x|\mathcal{M}_p, \mathcal{M}_t) = gain(x|\mathcal{M}_p, \mathcal{M}_t) - investment(x|\mathcal{M}_p, \mathcal{M}_t)$$

where

$$\begin{aligned} gain(x|\mathcal{M}_p, \mathcal{M}_t) = \; & \textstyle\sum_{e \in \mathcal{M}_p}(maintcost(e|\mathcal{M}_p, \mathcal{M}_t) - maintcost(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\})) \\ & + \textstyle\sum_{e \in \mathcal{M}_t}(transcost(e|\mathcal{M}_p, \mathcal{M}_t) - transcost(e|\mathcal{M}_p, \mathcal{M}_t \cup \{x\})) \end{aligned}$$

and

$$investment(x|\mathcal{M}_p, \mathcal{M}_t) = \begin{cases} \min(maintcost(x|\mathcal{M}_p, \mathcal{M}_t), transcost(x|\mathcal{M}_p, \mathcal{M}_t)) \\ \quad \text{if } x \text{ is a full result} \\ transcost(x|\mathcal{M}_p, \mathcal{M}_t) \\ \quad \text{if } x \text{ is a differential} \end{cases}$$

Figure 5.1 outlines a greedy algorithm that iteratively picks nodes to be materialized. The procedure takes as input the set $\mathcal{C}$ of candidates (equivalence nodes, and their differentials) for

materialization, and returns the sets $\mathcal{M}_p$ and $\mathcal{M}_t$ of equivalence nodes to be materialized permanently and transiently, respectively. $\mathcal{M}_p$ is initialized to $\mathcal{M}$, the set of equivalence nodes for the initial materialized views, while $\mathcal{M}_t$ is initialized as empty. At each iteration, the equivalence node $x \in \mathcal{C}$ with the maximum benefit is selected for materialization. If x is a full result, then it is added to either $\mathcal{M}_p$ or $\mathcal{M}_t$ based on whether maintaining it or transiently materializing it would be cheaper; if x is a differential, then it is added to $\mathcal{M}_t$ since it cannot be permanently materialized.

Naively, the candidate set $\mathcal{C}$ can be the set of all equivalence nodes in the Query DAG (full results as well as differentials). In Section 5.5.2, we consider approaches to reduce the candidate set.

5.5.2 Optimizations

Three important optimizations to the greedy algorithm for multi-query optimization were presented in Chapter 3. While monotonicity optimization applies unchanges, the incremental cost update and sharability computation need to be extended to handle differentials, as follows.

1. The incremental cost update algorithm presented in Chapter 3 maintains the state of the Query DAG (which includes previously computed best plans for the equivalence nodes) across calls, and may even avoid visiting many of the ancestors of a node whose cost has been modified due to materialization or unmaterialization.

   We modify the incremental cost update algorithm to handle differentials as follows.

   (a) If the full result of a node is materialized, we update not only the cost of computing the full result of each ancestor node, but also the costs for the $2n$ differentials of each ancestor node since the full result may be used in any of the $2n$ differentials. Propagation up from an ancestor node can be stopped if there is no change in cost to computing the full result or any of the differentials.

   (b) If the differential of a node with respect to a given update is materialized, we update only the differentials of its ancestors with respect to the same update. Propagation can

stop on ancestors whose differentials with respect to the given update do not change in cost.

2. It is wasteful to transiently materialize nodes unless they are used multiple times during the refresh. An algorithm for computing sharability of nodes as proposed in Chapter 3, which detects equivalence nodes that can potentially be used multiple times in a single plan. We consider differential results for transient materialization only if the corresponding full result is detected to be sharable.

   The sharability optimization cannot be applied to full results in our context, since a full result may be worth materializing permanently even if it is used in only one query. Thus all full results are candidates for optimization.

We also observed that when it is worth transiently materializing the differential of an expression with respect to the update of a particular base relation, it is often worth transiently materializing the differentials with respect to updates of the other base relations as well. To reduce the cost of the greedy algorithm, we consider all differentials of an expression (with respect to different base relation updates) as a single unit of materialization. The number of candidates considered by the greedy algorithm reduces greatly as a result, reducing its execution time significantly.

5.5.3 Extensions

The algorithms we have outlined can be extended in several ways. One direction is to deal with limited space for storing materialized results. To deal with this problem, we can modify the greedy algorithm to prioritize results in order of benefit per unit space (got by dividing the benefit by the size of the result). If the space available for permanent and transient materialized results are separate, we can modify the algorithm to continue considering results for permanent (resp. transient) materialization even after the space of transient (resp. permanent) materialization is exhausted.

Another direction of extension would be to select materialized views in order to speed up a

workload of queries. The greedy algorithm can be modified for this task as follows: candidates would be final/intermediate results of queries, and benefits to queries would be included when computing benefits. In fact, many of the approaches proposed earlier for selecting materialized views use such a greedy approach, and our implementation techniques provide an efficient way to implement these algorithms. Longer term future work would include dealing with large sets of queries efficiently.

5.6 Performance Study

We implemented the algorithms described earlier for finding optimal plans for view maintenance. As mentioned earlier, the implementation performs index selection along with selection of results to materialize. The implementation was performed on top of an existing query optimizer.

5.6.1 Performance Model

We used a benchmark consisting of views representing the results of queries based on the TPC-D schema. In particular, we separately considered the following two workloads:

- *Set of Views Workload.* A set of 10 views, 5 with aggregates and 5 without, on a total of 8 distinct relations. There is some amount of overlap across these views, but most of the views have selections that are not present in other views, limiting the amount of overlap.

- *Single Views Workload.* The same views as above, but each optimized and executed separately, and we show the sum of the view maintenance times. Since the views are optimized separately, as if they were on separate copies of the database, sharing between views cannot be exploited.

The materialized views are shown in Appendix A.2. The purpose of choosing a simple workload in addition to the complex workload is to show that our methods are very effective not only for big sets of overlapping complex views, where one might argue that simple multi-query optimization may be as effective, but also for singleton views without common subexpressions, where a

technique based exclusively on multi-query optimization would be useless.

The performance measure is *estimated maintenance cost*. The cost model used takes into account number of seeks, amount of data read, amount of data written, and CPU time for in-memory processing. While we would have liked to give actual run times on a real database, we do not currently have a query execution engine which we can extend to perform differential view maintenance. We are working on translation of the plans into SQL queries that can be run on any SQL database. However, the results would not be as good as if we had fine grain control, since the translation will split queries into small pieces whose results are stored in disk and then used, resulting in decreased pipelining benefits. Our cost model is fairly sophisticated, and we have verified its accuracy by comparing its estimates with numbers obtained by running queries on commercial database systems. We found close agreement (within around 10 percent) on most queries, which indicates that the numbers obtained in our performance study are fairly accurate.

We provide performance numbers for different percentages of updates to the database relations; we assume that all relations are updated by the same percentage. In our notation, a 10% update to a relation consists of inserting 10% as many tuples are currently in the relation.

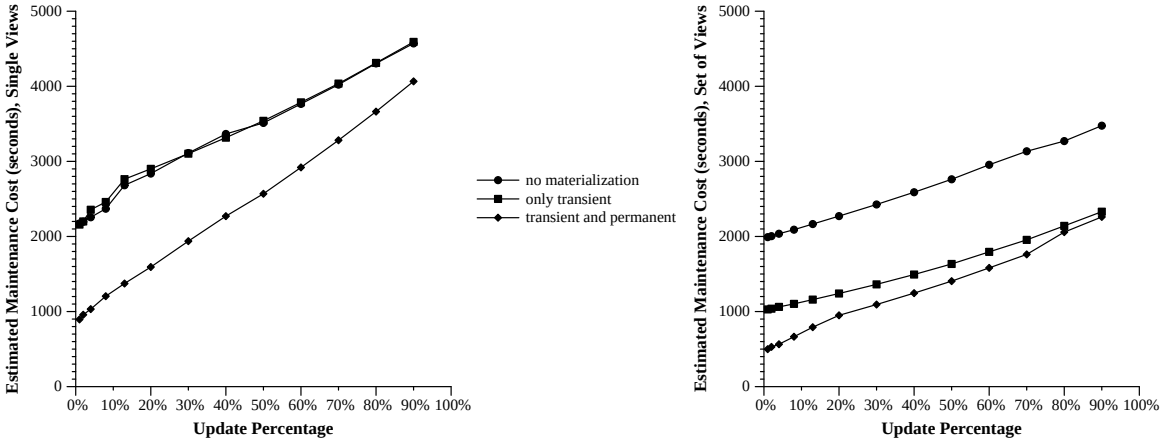
We assume a TPC-D database at scale factor of 0.1, that is the relations occupy a total of 100 MB. The buffer size is set at 8000 blocks, each of size 4KB, for a total of 32 MB, although we also ran some tests at a much smaller buffer size of 1000 blocks. However, the numbers are not greatly affected by the buffer size, and in fact smaller buffer sizes can be expected to benefit more from sharing of common subexpressions. The tests were run on an Ultrasparc 10, with 256 MB of memory.

5.6.2 Performance Results

The purpose of the experiments reported in this section is to:

1. Verify the efficacy of transient and permanent materialization of additional views (Section 5.6.2),

2. Verify the efficacy of adaptive determination of maintenance policy for each permanently materialized view (Section 5.6.2), and



Single Views Set of Views

Figure 5.2: Effect of Transient and Permanent Materialization

3. Establish that our methods are indeed practical by showing that the overheads of our optimization-based techniques are reasonable, and that our methods scale with respect to increasing number of views (Section 5.6.2).

Effect of Transient and Permanent Materialization

We executed the following variations of our algorithm:

- *No Materialization.* Neither transient nor permanent materialization of additional views is allowed. That is, only the given set of initial views is permanently materialized and maintained without any sharing. This corresponds to the current state of the art.

- *Only Transient.* Transient materialization is allowed, but permanent materialization of additional views is disallowed. This corresponds to using multi-query optimization in view maintenance.

- *Transient and Permanent.* Both transient and permanent materialization of additional results is allowed. This corresponds to the techniques proposed in this chapter.

In all the cases, the maintenance policy of each of the views is decided based on whether recomputation and incremental computation is cheaper, given the constraints in each case as above.

The results for the single view workload and the set of views workload are reported in Figure 5.2.

For the single-view workload, transient materialization is not useful if the view maintenance plan used is recomputation, but when incremental computation is used, full results can potentially be shared between differentials for updates to different base relations. Indeed, we found several such instances at low update percentages. At higher update percentages we found fewer such occurrences, and using only transient materialization did not offer much benefit. However, permanent materialization of intermediate results reduces the overall materialization cost by up to 50% for smaller update percentages (the smallest update percentage we considered was 1%). These results clearly illustrate the efficacy of the methods proposed in this chapter over and above multi-query optimization (Chapter 3).

The set of views workload has a significant amount of overlap among the constituent views. Thus, the substantial reduction, as high as 48%, in the overall maintenance cost due to only transient materialization is as expected. Permanent materialization has a significant impact in this case also, and further reduces the maintenance cost by up to another 17%, resulting in a total reduction of up to 65%.

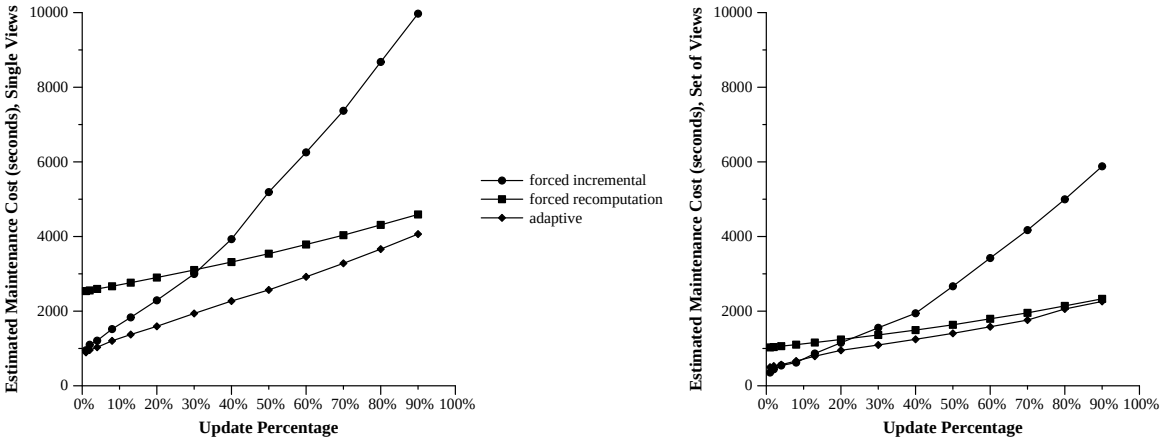
Recall from our discussion in Section 5.4 that all additional permanently materialized nodes are always maintained incrementally, since if recomputation-based maintenance of these views is cheaper than incremental maintenance, then they would be chosen for transient materialization instead of permanent materialization. Now, the cost of incremental maintenance increases with the size of the updates; for larger updates, recomputation of a permanently materialized view is a better alternative than incremental maintenance, so a smaller fraction of views are permanently materialized. These two facts together account for the slightly decreasing advantage of transient cum permanent materialization over only transient materialization as update percentages increase, as is clear from the convergence of the respective plots in Figure 5.2 for either workload.

Comparing across the two workloads reveals an interesting result: the cost of maintenance without selecting additional materialized view is less for the set of views than for the single view workload, even though they have the same set of queries. The reason is that in the case of set of views, the maintenance of a view can exploit the presence of *existing* materialized views, even without selecting additional materialized views. Our optimizer indeed takes such plans into consideration even when it does not select additional materialized views.

We also executed tests on an *Only Permanent* variant of our algorithms, where permanent materialization is allowed, but transient materialization of additional views is disallowed. This corresponds to using only permanent materialized view selection for optimization of view maintenance. However, since views for which the recomputation is cheaper than incremental maintenance can still be permanently materialized, the only difference from the case of transient and permanent is that differential results cannot be shared.

For the single view benchmark there is no possibility of sharing differential results, since each query can have only one occurrence of any expression involving a particular differential. For the set of views benchmark, we found that the benefits of materializing differentials was relatively low. Full results are more expensive to compute, and since they can be used with differentials for all relations not used in their definition, they are also shared to a greater degree. As a result full results are preferentially chosen for materialization, and differential results were rarely chosen, and even when chosen gave only small benefits. Thus, in this case too the plots for only permanent were almost identical to the plots for transient and permanent. To avoid clutter, we omitted the plots for only permanent from our graphs.

To summarize this section, to the best of our knowledge ours is the first study that demonstrates quantitatively the benefits of materializing extra views (transiently or permanently) to speed up view maintenance in a general setting. Earlier work on selection of materialized views, as far as we are aware, has not presented any performance results except in the limited context of data cubes or star schemas [11].



Single Views Set of Views

Figure 5.3: Effect of Adaptive Maintenance Policy Selection

Effect of Adaptive Maintenance Policy Selection

In the current database systems, the user needs to specify the maintenance policy (incremental or recomputation) for a materialized view during its definition. In this section, we show that an apriori fixed specification as above may not be the a good idea, and make a case for adaptively choosing the maintenance policy for a view in an adaptive manner.

We explored the following variants of our algorithm:

- *Forced Incremental.* All the permanent materialized views, including the views given initially as well as the views picked additionally by greedy, are forced to be maintained incrementally.

- *Forced Recomputation.* Incremental maintenance is disallowed and all the permanent materialized views are forced to be recomputed.

- *Adaptive.* The maintenance policy, incremental or recomputation, for each permanently materialized view is chosen based on the goal of minimizing the overall maintenance cost; one or the other may be chosen for a given view at different update percentages. This corresponds to the techniques proposed in the chapter.

In all the cases, additional transient and materialized views were chosen by executing greedy as described earlier in the chapter. The results of executing the above variants on each of our workloads are plotted in Figure 5.3.

The graphs show that incremental maintenance may be much more expensive than recomputation; the incremental maintenance cost increases sharply for medium to large update percentages – in our case, beyond 30% for the single view workload, and beyond 20% for the multi-view workload. In both the workloads, the adaptive technique performs better than both forced incremental and forced recomputation; this extra improvement, up to 34% for the single-view workload, is due to its ability to adaptively choose incremental maintenance for some of the initial as well as additionally materialized views, and recomputation for the others and always maintain a mix that leads to the lowest overall maintenance cost. However, the difference between adaptive and forced recomputation for either workload decreases slightly with increasing update percentage. This is because for large update percentages, incremental maintenance is expensive, and hence every view is recomputed.

These observations clearly show that blindly favoring incremental maintenance over recomputation may not be a good idea (this conclusion is similar to the findings of Vista [61]); and make a case for adaptively choosing the maintenance policy for each view, as done by our algorithms. It is also important to note that the ability to mix different maintenance policies for different subparts of the maintenance plan, even for a single view, is novel to our techniques, and not supported by [61].

Overheads and Scalability Analysis

To see how well our algorithms scale up with increasing numbers of views, we used the following benchmark. The benchmark uses 22 relations PSP_1 to PSP_{22} with an identical schema (P, SP, NUM) denoting part id, subpart id and number. Over these relations, we defined a sequence of 10 views V_1 to V_{10}: the view V_i was a star query on four relations PSP_1, PSP_{2i}, PSP_{2i+1} and PSP_{2i+2}, with $PSP_1.SP$ joined with $PSP_{2i}.P$, $PSP_{2i+1}.P$, and $PSP_{2i+2}.P$. We then grouped these views into 10 sets, where the k^{th} set SV_k consisted of the k views $V_1, \ldots, V_k$.

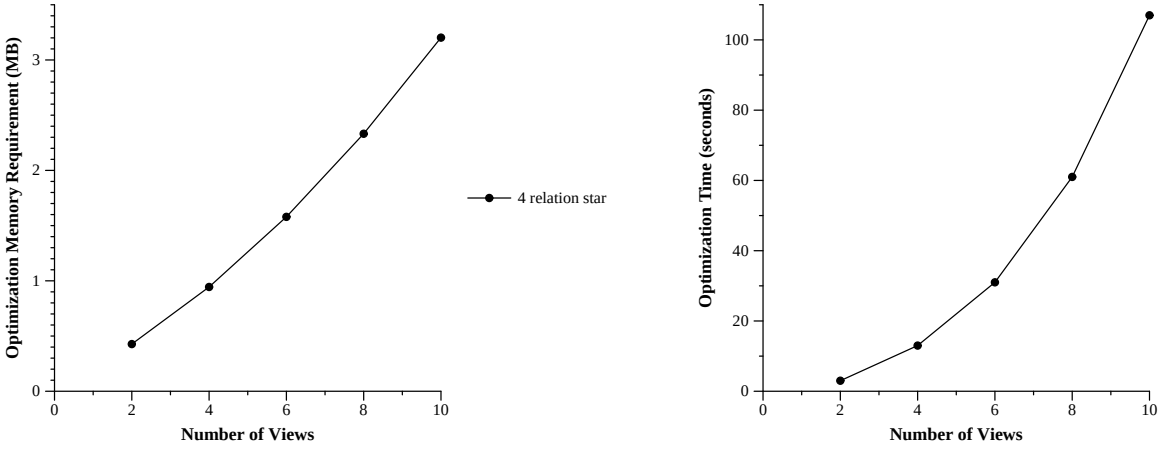


Figure 5.4: Scalability analysis on increasing number of views

For each SV_k we measured (a) the memory requirements of our algorithm and (b) the time taken by our algorithm, and report the same in Figure 5.4.

The figure shows that the memory consumption of our algorithm increases practically linearly with the number of views in the set. The reason for this is that the memory usage is basically in maintaining the Query DAG, and for our view set, the increase in the size of the Query DAG is constant per additional view added to the DAG (with a fixed number of base relations). The memory requirement for the view set SV_{10}, containing 10 views on a total of 22 relations, is only about 3.2 MB.

Further, addition of a new view from our view set to the Query DAG increases the breadth of the DAG, not its height (we think this is the expected case in reality – most views are expected to be of similar size and with only partial mutual overlap). Since the height remains constant, the time taken per incremental cost update (ref. Section 5.5.2) remains constant. However, the number of these incremental cost updates increases quadratically with the size of the Query DAG, as observed by in Chapter 3. This accounts for the quadratic increase in the time spent by our algorithm with increasing number of views, as shown in Figure 5.4. However, despite the quadratic growth, the time spent on the 22-relation 10-view set SV_{10} was less than a couple of minutes. This is very reasonable for an algorithm that needs to be executed only occasionally, and which provides savings of the order of 1000's of seconds on each view refresh.

Thus, we conclude that the memory requirements of our algorithm are reasonable and scale well with increasing number of views. The time taken shows quadratic growth, but this growth is slow enough to make the algorithm practical for large enough view sets; especially since the tremendous cumulative reduction in the maintenance cost across multiple maintenance passes far outweighs the time spent only once while executing the algorithm to make the reduction possible.

Finally, we tested the effect of our optimization of treating all the deltas of an expression as a single unit of materialization instead of considering them separately. We found that this reduced the time taken for greedy optimization by about 30 percent, yet made no difference to the plans generated. However, neither alternative found any significant benefits for materializing delta results, whether as a single unit or separately, for reasons that we outlined earlier when discussing the effect of "only permanent". Optimization time can therefore be saved by not considering any deltas as candidates for materialization; we found this reduces optimization times by a further factor of 2 from those reported in our experiments.

5.7 Summary

The problem of finding the best way to maintain a given set of materialized views is an important practical problem, especially in data warehouses/data marts, where the maintenance windows are shrinking. We have presented solutions that exploit commonality between different tasks in view maintenance, to minimize the cost of maintenance. Our techniques have been implemented on an existing optimizer, and we have conducted a performance study of their benefits. As shown by the results in section 5.6, our techniques can generate significant speedup in view maintenance cost, and the increase in cost of optimization is acceptable. *We therefore believe that our techniques provide a timely and effective solution to a very important real problem.*

Chapter 6

Conclusions and Future Work

In this thesis, we looked at ways to exploit shared computation in order to speed up query processing. Review of transformational cost-based query optimization in terms of our version of the Volcano algorithm [23] was provided in Chapter 2. The framework explained in that chapter is extended in the later chapters to incorporate multi-query optimization, query result caching and materialized view selection and maintenence.

In Chapter 3, we looked at multi-query optimization and introduced three novel heuristic search algorithms, Volcano-SH, Volcano-RU and Greedy, for the same. Among these, the Greedy algorithm proved to be the most promising, and flexible enough to be applied to the problems of query result caching and materialized view selection and maintenance. One of the major contributions of this work are a number of techniques to greatly speed up the greedy algorithm, making use of the structure of the Query DAG on which our implementation is based.

In Chapter 4, we presented new techniques for query result caching, based on the core framework developed in Chapter 3, which can help speed up query processing in data warehouses. The novel features incorporated in our system, Exchequer, include optimization aware cache maintenance and the use of a cache aware optimizer. In contrast, in existing work, the module that makes cost-benefit decisions is part of the cache manager and works independent of the optimizer which essentially reconsiders these decisions while finding the best plan for a query.

In Chapter 5, we presented techniques that exploit commonality between different tasks to

speed up view maintenance, and also select additional views for materialization to minimize the overall cost of maintenance. These techniques, which are extensions of the core techniques developed in context of multi-query optimization in Chapter 3, can generate significant speedup in view maintenance cost, and the increase in cost of optimization is acceptable.

Our algorithms are based on the AND/OR Query DAG representation of queries, making them easily extensible to handle new transformations, operators and implementations. Our algorithms also handle index selection and nested queries, in a very natural manner. We also developed extensions to the Query DAG generation algorithm as proposed for Volcano [23] to detect all common sub expressions and include subsumption derivations. Further, our algorithms are easy to implement on a Volcano-type query optimizer (e.g. the Cascades optimizer of Microsoft SQL-Server [22] and the optimizer of the Tandem ServerWare SQL Product [6]), requiring addition of only a few thousand lines of code.

Future Work

Our current work on multi-query optimization (Chapter 3) does not take space constraints into account. While changing our techniques given a constraint on the total size of all materialized results is straightforward (use benefit-per-unit size instead of benefit in the Greedy algorithm, as in the case of Query Result Caching), it would be too pessimistic. This is because it is seldom the case that the materialized results are to be used all at the same time. As such, it should be possible to schedule the execution such that first use of a materialized result e, the point when e gets materialized, follows the last use of another result e', the point when e' can be disposed; thus, the same disk space can be used for both e and e'. Determining such plans requires an interleaving of query optimization and scheduling, and promises to be an interesting problem to explore.

Moreover, during query execution, pipelining can be generalized to incorporate multiple consumers (multiple parts of the query that share an intermediate result) without materialization e.g., the Redbrick data warehouse product allows a scan of a base relation to be shared by multiple consumers. In this thesis, we have assumed that sharing always results in materialization; Dalvi

et al. [14] have extended this work to incorporate shared pipelines. Another followup work by Hulgeri et al. [30] incorporates into our work the issues of allocation of memory to individual operators executing in a pipeline. Furthermore, the materialization cost can be eliminated or reduced in some cases by piggybacking the materialization with the actions of an operator that uses the expression. For instance, if an expression is the input to a sort, it can be materialized by simply saving runs generated during sorting, at no extra cost.

In query result caching, we can compactly represent large workloads by making use of the fact that many queries (or parts of queries) in a large workload are likely to be the same except for values of selection constants. We can unify such selections and replace them by a parameterized selection, thereby collapsing many selections into a single parameterized selection that is invoked as many times as the number of selections we replaced.

Also, when we run short of cache space, instead of discarding a stored result in its entirety, it should be possible to (a) replace it by a summarization, or (b) discard only parts of the result. We can implement the latter by partitioning selection nodes into smaller selects and replacing the original select by a union of the other selects. Two issues in introducing these partitioned nodes are: (a) What partition should we choose? and (b) If the top level is not a select, we can still choose an attribute to partition on, but which should this be?

An important direction of future work is to take updates into account in Query Result Caching, thus integrating the techniques developed in Chapter 4 and Chapter 5. We need to develop techniques for: (a) taking update frequencies into account when deciding whether to cache a particular result, and (b) decide when and whether to discard or refresh cached results. We could refresh cached results eagerly as updates happen, or update them lazily, when they are accessed. Another aspect of the integration could be to take into account the query workloads apart from the materialized views in order to determine what additional views to materialize.

Finally, Query DAG generation can be extended to include *query splitting* [15] as well. For example, given $e1$: $\sigma_{A<5}(E)$ and $e2$: $\sigma_{A<10}(E)$, an alternative plan for $e2$ can be obtained by introducing the *remainder* expression $e3$:$\sigma_{5\leq A<10}(E)$ in the Query DAG, and taking its union with $e1$, i.e., $\sigma_{A<10}(E) \equiv \sigma_{A<5}(E) \cup \sigma_{5\leq A<10}(E)$. However, this plan, along with the plan $\sigma_{A<5}(E) \equiv \sigma_{A<5}(\sigma_{A<10}(E))$ introduced by the subsumption derivations, leads to a cycle involv-

ing $e1$ and $e2$, countering our assumptions about the Query DAG. We are currently working on approaches to address the above problem.

Appendix A

TPCD-Based Benchmark Queries

A.1 List of Queries Used in Section 3.6

Q2

```
SELECT P_PARTKEY
FROM PART, PARTSUPP, SUPPLIER, NATION, REGION
WHERE P_PARTKEY = PS_PARTKEY
AND S_SUPPKEY = PS_SUPPKEY
AND P_SIZE = 10
AND S_NATIONKEY = N_NATIONKEY
AND N_REGIONKEY = R_REGIONKEY
AND R_NAME = 1
AND PS_SUPPLYCOST IN (
    SELECT MIN(PS_SUPPLYCOST)
    FROM PARTSUPP, SUPPLIER, NATION, REGION
    WHERE P_PARTKEY = PS_PARTKEY
    AND S_SUPPKEY = PS_SUPPKEY
    AND S_NATIONKEY = N_NATIONKEY
    AND N_REGIONKEY = R_REGIONKEY
    AND R_NAME = 1
    GROUP BY PS_CONST
);
```

Q3

```
SELECT O_SELKEY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_SELKEY = 1
```

```
AND C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND O_SELKEY < 13
AND L_SELKEY > 12;
```

Q5

```
SELECT MAX(O_SELKEY)
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY
AND O_ORDERKEY = L_ORDERKEY
AND L_SUPPKEY = S_SUPPKEY
AND C_NATIONKEY = S_NATIONKEY
AND S_NATIONKEY = N_NATIONKEY
AND N_REGIONKEY = R_REGIONKEY
AND R_REGIONKEY = 1
AND O_SELKEY < 5
GROUP BY N_NATIONKEY;
```

Q7

```
SELECT S_SUPPKEY
FROM SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION, NATION1
WHERE S_SUPPKEY = L_SUPPKEY
AND O_ORDERKEY = L_ORDERKEY
AND C_CUSTKEY = O_CUSTKEY
AND S_NATIONKEY = NATION.N_NATIONKEY
AND C_NATIONKEY = NATION1.N1_NATIONKEY
AND ((NATION.N_NATIONKEY = 1 AND NATION1.N1_NATIONKEY = 2)
 OR (NATION.N_NATIONKEY = 2 AND NATION1.N1_NATIONKEY = 1))
AND L_SELKEY > 16;
```

Q8

```
SELECT P_PARTKEY
FROM PART, SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION, NATION1, REGION
WHERE  P_PARTKEY = L_PARTKEY
AND S_SUPPKEY = L_SUPPKEY
AND L_ORDERKEY = O_ORDERKEY
AND O_CUSTKEY = C_CUSTKEY
AND C_NATIONKEY = NATION.N_NATIONKEY
AND NATION.N_NATIONKEY = R_REGIONKEY
AND R_REGIONKEY = 2
AND S_NATIONKEY = NATION1.N1_NATIONKEY
AND O_SELKEY > 16
AND P_SELKEY < 3;
```

Q9

```
SELECT P_SELKEY
FROM PART, SUPPLIER, LINEITEM, PARTSUPP, ORDERS, NATION
WHERE S_SUPPKEY = L_SUPPKEY
AND PS_SUPPKEY = L_SUPPKEY
AND PS_PARTKEY = L_PARTKEY
AND P_PARTKEY = L_PARTKEY
AND O_ORDERKEY = L_ORDERKEY
AND S_NATIONKEY = N_NATIONKEY
AND P_SELKEY > 251;
```

Q10

```
SELECT MIN(CUSTOMER.C_CUSTKEY)
FROM CUSTOMER, ORDERS, LINEITEM, NATION
WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
AND ORDERS.O_SELKEY = 1
AND LINEITEM.L_SELKEY < 7
AND CUSTOMER.C_NATIONKEY = NATION.N_NATIONKEY
GROUP BY CUSTOMER.C_CUSTKEY, NATION.N_NATIONKEY;
```

Q11

```
SELECT MIN(PARTSUPP.PS_SUPPKEY)
FROM PARTSUPP, SUPPLIER, NATION
WHERE PARTSUPP.PS_SUPPKEY = SUPPLIER.S_SUPPKEY
AND SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY
AND NATION.N_NATIONKEY = 7
GROUP BY PARTSUPP.PS_PARTKEY;

SELECT PARTSUPP.PS_SUPPKEY
FROM PARTSUPP, SUPPLIER, NATION
WHERE PARTSUPP.PS_SUPPKEY = SUPPLIER.S_SUPPKEY
AND SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY
AND NATION.N_NATIONKEY = 7;
```

Q14

```
SELECT LINEITEM.L_PARTKEY
FROM LINEITEM, PART
WHERE LINEITEM.L_PARTKEY = PART.P_PARTKEY
AND LINEITEM.L_SELKEY =  20;
```

A.2 List of View Definitions Used in Section 5.6

```
SELECT MIN(CUSTOMER.C_SELKEY)
FROM CUSTOMER, ORDERS, LINEITEM, NATION
WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
AND CUSTOMER.C_NATIONKEY = NATION.N_NATIONKEY
GROUP BY CUSTOMER.C_CUSTKEY, NATION.N_NATIONKEY;

SELECT MIN(CUSTOMER.C_SELKEY)
FROM CUSTOMER, ORDERS, LINEITEM
WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
GROUP BY CUSTOMER.C_CUSTKEY
HAVING CUSTOMER.C_CUSTKEY > 2;

SELECT MIN(PARTSUPP.PS_SUPPKEY)
FROM PARTSUPP, SUPPLIER, NATION
WHERE PARTSUPP.PS_SUPPKEY = SUPPLIER.S_SUPPKEY
AND SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY
AND NATION.N_NATIONKEY = 7;

SELECT COUNT(SUPPLIER.S_SUPPKEY)
FROM SUPPLIER, LINEITEM, ORDERS
WHERE SUPPLIER.S_SUPPKEY  = LINEITEM.L_SUPPKEY
AND ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
AND LINEITEM.L_SELKEY > 16
GROUP BY SUPPLIER.S_NATIONKEY, LINEITEM.L_ORDERKEY;

SELECT MIN(PARTSUPP.PS_SUPPLYCOST)
FROM PARTSUPP , PART , LINEITEM , ORDERS
WHERE PARTSUPP.PS_PARTKEY > 10
AND PART.P_PARTKEY = PARTSUPP.PS_PARTKEY
AND LINEITEM.L_PARTKEY = PARTSUPP.PS_PARTKEY
AND ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
GROUP BY PART.P_PARTKEY;

SELECT PARTSUPP.PS_SUPPLYCOST
FROM PARTSUPP , LINEITEM , ORDERS
WHERE PARTSUPP.PS_PARTKEY > 10
AND LINEITEM.L_PARTKEY = PARTSUPP.PS_PARTKEY
AND ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY;
```

```
SELECT PARTSUPP.PS_SUPPLYCOST
FROM PART , SUPPLIER , PARTSUPP , NATION , REGION
WHERE PART.P_PARTKEY = PARTSUPP.PS_PARTKEY
AND SUPPLIER.S_SUPPKEY = PARTSUPP.PS_SUPPKEY
AND SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY
AND NATION.N_REGIONKEY = REGION.R_REGIONKEY;

SELECT PARTSUPP.PS_SUPPLYCOST
FROM PART , PARTSUPP , LINEITEM , SUPPLIER
WHERE PART.P_PARTKEY = PARTSUPP.PS_PARTKEY
AND SUPPLIER.S_SUPPKEY = PARTSUPP.PS_SUPPKEY
AND LINEITEM.L_PARTKEY = PARTSUPP.PS_PARTKEY;

SELECT PARTSUPP.PS_SUPPLYCOST
FROM PARTSUPP , PART , LINEITEM , ORDERS
WHERE PARTSUPP.PS_PARTKEY > 10
AND PART.P_PARTKEY = PARTSUPP.PS_PARTKEY
AND LINEITEM.L_PARTKEY = PARTSUPP.PS_PARTKEY
AND ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY;

SELECT PARTSUPP.PS_SUPPLYCOST
FROM PART , SUPPLIER , PARTSUPP
WHERE PART.P_PARTKEY = PARTSUPP.PS_PARTKEY
AND SUPPLIER.S_SUPPKEY = PARTSUPP.PS_SUPPKEY;
```

Appendix B

List of Logical Transformations

In this section, we list the main logical transformations used to generate the logical plan space. These transformations are augmented by the subsumption transformations mentioned in Chapter 3. Note that, for space efficiency, we do not represent $A \bowtie B$ and $B \bowtie A$ separately; the transformations stated below take care of this fact.

Select Predicate Pushdown

$$\begin{aligned} \sigma_\theta(A \bowtie_{theta'} B) &\rightarrow A \bowtie_{\theta \wedge \theta'} B \\ \sigma_\theta(A \times B) &\rightarrow A \bowtie_\theta B \end{aligned}$$

Join Predicate Pushdown

$$A \bowtie_\theta B \rightarrow \sigma_{\theta_A}(A) \bowtie_{\theta'} \sigma_{\theta_B}(B)$$

where[1] $\theta_A \wedge \theta' \wedge \theta_B \equiv \theta$, $attr(\theta_A) \subseteq attr(A)$, $attr(\theta_B) \subseteq attr(B)$, $attr(\theta') \not\subseteq attr(A)$, and $attr(\theta') \not\subseteq attr(B)$.

Further, A is used instead of σ_{θ_A} is $\theta_A \equiv true$. Similarly for σ_{θ_B} and σ_{θ_C}. Also, if $\theta' \equiv true$ then $\times$ is used instead of $\bowtie_{\theta'}$.

[1] $attr(X)$ is the set of attributes of relation X; $attr(\theta)$ is the set of attributes referenced in predicate θ.

Join Left Associativity

$$\begin{aligned} A \bowtie (B \bowtie C) &\rightarrow (A \bowtie B) \bowtie C \\ &\rightarrow (A \bowtie C) \bowtie B \end{aligned}$$

Join Right Associativity

$$\begin{aligned} (A \bowtie B) \bowtie C &\rightarrow A \bowtie (B \bowtie C) \\ &\rightarrow B \bowtie (A \bowtie C) \end{aligned}$$

Join Exchange

$$\begin{aligned} (A \bowtie B) \bowtie (C \bowtie D) &\rightarrow (A \bowtie C) \bowtie (B \bowtie D) \\ &\rightarrow (B \bowtie C) \bowtie (A \bowtie D) \\ &\rightarrow (A \bowtie D) \bowtie (B \bowtie C) \\ &\rightarrow (B \bowtie D) \bowtie (A \bowtie C) \end{aligned}$$

Appendix C

Operator Cost Estimates

In this appendix, we present formulae giving the cost estimates for the various physical operators considered by our optimizer. Our performance studies in earlier chapters (ref. Section 3.6.1 and Section 4.5.4) attest to the accuracy of these cost estimates. Figure C.1 gives the values of the constants involved in the formulae along with their values, and Figure C.2 summarizes the parameters used in the cost formulae.

In the discussion below, the inputs are assumed to be available in a stream; the operator does not pay any cost for reading in the inputs. Similarly, the output is streamed out and the operator does not pay any cost for writing the output. The cost is in terms of the response time measured in milliseconds.

Assuming that, on the average, the operators execute I instructions per byte of data processed, then with a block size of B KB/block and CPU speed of P MIPS, we get the computation cost as $I * B/P$ ms/block. Thus, in terms of C_{IO} and N, the total cost C (in ms) is computed as:

$$C = C_{IO} + I * B * N/P$$

The sections below give, for each operator, the formulae for C_{IO}, the I/O cost (in milliseconds) and N, the blocks of memory processed by the CPU.

Relation Scan. Each block of the relation read and processed once.

$$C_{IO} = R * S_o$$

R	readtime (ms)	2 ms
W	writetime (ms)	4 ms
K	seektime (ms)	8 ms
F	index fanout	20
B	size of a block in kilobytes	4 KB
P	CPU speed in MIPS	100 MIPS
M	available main memory (number of blocks)	8000 blocks
I	average number of instructions executed per byte of data	5

Figure C.1: Constants

S_{ik}	size of the k^{th} input (number of blocks)
T_{ik}	size of the k^{th} input (number of tuples)
S_o	size of the output (number of blocks)
T_o	size of the output (number of tuples)
D_{ik}	number of distinct values in the k^{th} input

Figure C.2: Cost Formulae Parameters

$$N = S_o$$

Result Materialization. Each block of the relation processed and written once.

$$C_{IO} = W * S_o$$

$$N = S_o$$

Sort. In memory sort if the relation fits in the main memory. Otherwise, merge sort with fanin $M - 1$.

$$C_{IO} = \begin{cases} 0 & \text{if } S_o \leq M \\ (R + W) * S_o * \lceil \log_{M-1}(S_o/M) \rceil & \text{otherwise} \end{cases}$$

$$N = \log_2(T_o) * S_o + 1$$

Clustered Index Creation on Sorted Relation. The input is already sorted on the relevant attribute. The index B-Tree is created bottom-up.

Size of clustered index (in number of blocks) $= S_o * (1 + 1/F + 1/F^2 + \ldots) = S_o * F/(F-1)$.

$$C_{IO} = W * S_o * F/(F - 1)$$

$$N = 0$$

Clustered Index Creation on Unsorted Relation. The input is first sorted. Then the index is created bottom-up. The overall cost is the total of the sorting cost and the index creation cost.

$$C_{IO} = (R + W) * S_o * \lceil log_{M-1}(S_o/M) \rceil + W * S_o * F/(F - 1)$$

$$N = log_2(T_o) * S_o + 1$$

Selection. The input streaming in is filtered using the predicate and the result streamed out. No I/O occurs.

$$C_{IO} = 0$$

$$N = S_{i0} + S_o$$

Index based Select. We assume at least first level of the clustered is in memory. If $S_o < 0.25 * M$, assume lower levels are also partially cached.

$$C_I = \begin{cases} 0 & \text{if } S_{i0} < 0.25 * M \\ \max(0, \lceil log_F(S_{i0}) \rceil) & \text{if } 0.25 * M \leq S_{i0} < 0.75 * M \\ \lceil log_F(S_{i0}) \rceil & \text{otherwise} \end{cases}$$

$$C_{IO} = S_o(K + R) * C_I + R * S_o$$

$$N = S_o$$

Merge Join. Both the inputs are streaming in already sorted. We introduce an arbitrary factor of 2 to account for merge processing costs per block of output.

$$C_{IO} = 0$$

$$N = 2 * S_o$$

Nested Loops Join. Since the inputs are streaming in, we do not pay the read cost for the outer relation. If both inputs are smaller that $O.5 * M$, the join occurs in-memory without any need of I/O.

$$C_{IO} = \begin{cases} 0 & \text{if } S_{i0} < 0.5 * M \text{ or } S_{i1} < 0.5 * M \\ R * (S_{i0} * S_{i1})/(M - 1) & \text{otherwise} \end{cases}$$

$$N = \begin{cases} S_{i0} * T_{i1} + S_o & \text{if } S_{i0} < 0.5 * M \\ T_{i0} * S_{i1} + S_o & \text{if } S_{i0} \geq 0.5 * M \text{ and } S_{i1} < 0.5 * M \\ T_{i0} * T_{i1} + S_o & \text{otherwise} \end{cases}$$

Indexed Nested Loops Join. Input 0 is the probe and input 1 is indexed on the join attribute. The total number of block accesses B_I, assuming nothing is available in the cache is given by:

$$B_I = T_{i0} * \max(0, \lceil log_{F-1}(S_{i1}) \rceil) + S_{i1}/D_{i1}$$

B_I^* is the effective number block accesses taking into account the buffering.

$$B_I^* = \begin{cases} S_{i1}^2/M & \text{if } S_{i1} < 0.5 * M \text{ and } B_I > S_{i1}^2/M \\ B_I & \text{otherwise} \end{cases}$$

$$C_{IO} = (K + R) * B_I^*$$

$$N = T_{i0} * 0.05 + S_o$$

Hashing based Aggregation. We assume hybrid hashing, with half of the available M buffers are used in the hybrid portion.

$$C_{IO} = \begin{cases} (R + W) * (S_{i0} - 0.5 * M) & \text{if } S_o \geq 0.5 * M \\ 0 & \text{otherwise} \end{cases}$$

$$N = S_{i0} * 0.01 + S_o$$

Sort based Aggregation. The input is streaming in sorted, so no I/O is involved.

$$C_{IO} = 0$$

$$N = S_{i0} + S_o$$

References

[1] AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. Automated Selection of Materialized Views and Indexes in Microsoft SQL Server. In *Intl. Conf. Very Large Databases* (2000).

[2] ASHWIN, S., ROY, P., SESHADRI, S., AND SUDARSHAN, S. Garbage collection in object oriented databases using transactional cyclic reference counting. In *Intl. Conf. Very Large Databases* (1997).

[3] BLAKELEY, J., COBURN, N., AND LARSON, P. A. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *Intl. Conf. Very Large Databases* (1986).

[4] BLAKELEY, J. A., MCKENNA, W. J., AND GRAEFE, G. Experiences Building the Open OODB Query Optimizer. In *ACM SIGMOD Intl. Conf. on Management of Data* (Washington, DC., 1993), pp. 287–295.

[5] BOBROWSKI, S. Using materialized views to speed up queries. *Oracle Magazine* (Sept. 1999). http://www.oracle.com/oramag/oracle/99-Sep/59bob.html.

[6] CELIS, P. The Query Optimizer in Tandem's new ServerWare SQL Product. In *Intl. Conf. Very Large Databases* (1996).

[7] CHAUDHURI, S. An overview of query optimization in relational systems. In *ACM SIGACT-SIGART-SIGMOD Symposium on Priciples of Database Systems* (1998).

[8] CHAUDHURI, S., KRISHNAMURTHY, R., POTAMIANOS, S., AND SHIM, K. Optimizing queries with materialized views. In *Intl. Conf. on Data Engineering* (Taipei, Taiwan, 1995).

[9] CHAUDHURI, S., AND NARASAYYA, V. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Intl. Conf. Very Large Databases* (1997).

[10] CHEN, C. M., AND ROUSSOPOLOUS, N. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Intl. Conf. on Extending Database Technology (EDBT)* (1994).

[11] COLBY, L., COLE, R. L., HASLAM, E., JAZAYERI, N., JOHNSON, G., MCKENNA, W. J., SCHUMACHER, L., AND WILHITE, D. Redbrick Vista: Aggregate computation and management. In *Intl. Conf. on Data Engineering* (1998).

[12] COLBY, L., GRIFFIN, T., LIBKIN, L., MUMICK, I. S., AND TRICKEY, H. Algorithms for deferred view maintenance. In *ACM SIGMOD Intl. Conf. on Management of Data* (1996).

[13] COSAR, A., LIM, E.-P., AND SRIVASTAVA, J. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In *Intl. Conf. on Information and Knowledge Management (CIKM)* (1993).

[14] DALVI, N., SANGHAI, S., ROY, P., AND SUDARSHAN, S. Pipelining in multi-query optimization. Tech. rep., Indian Institute of Technology, Bombay, 2000. Submitted for publication.

[15] DAR, S., FRANKLIN, M. J., JONSSON, B. T., SRIVASTAVA, D., AND TAN, M. Semantic data caching and replacement. In *Intl. Conf. Very Large Databases* (1996).

[16] DESHPANDE, P. M., RAMASAMY, K., SHUKLA, A., AND NAUGHTON, J. F. Caching multidimensional queries using chunks. In *ACM SIGMOD Intl. Conf. on Management of Data* (1998).

[17] EDMONDS, J. Optimum branchings. *J. Research of the National Bureau of Standards 71B* (1967).

[18] FINKELSTEIN, S. Common expression analysis in database applications. In *ACM SIGMOD Intl. Conf. on Management of Data* (Orlando,FL, 1982), pp. 235–245.

[19] GANGULY, S. Design and analysis of parametric query optimization algorithms. In *Intl. Conf. Very Large Databases* (New York City, New York, August 1998).

[20] GASSNER, P., LOHMAN, G. M., SCHIEFER, K. B., AND WANG, Y. Query optimization in the ibm db2 family. *Data Engineering Bulletin 16*, 4 (1993).

[21] GRAEFE, G. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys 25*, 2 (1993).

[22] GRAEFE, G. The Cascades Framework for Query Optimization. *Data Engineering Bulletin 18*, 3 (1995).

[23] GRAEFE, G., AND MCKENNA, W. J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Intl. Conf. on Data Engineering* (1993).

[24] GRIFFIN, T., AND LIBKIN, L. Incremental maintenance of views with duplicates. In *ACM SIGMOD Intl. Conf. on Management of Data* (1995).

[25] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views : Problems, techniques, and applications. *IEEE Data Engineering Bulletin (Special issue on Materialized Views and Data Warehousing) 18(2) 18*, 2 (June 1995).

[26] GUPTA, H. Selection of views to materialize in a data warehouse. In *Intl. Conf. on Database Theory* (1997).

[27] GUPTA, H., HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. Index selection for olap. In *Intl. Conf. on Data Engineering* (Binghampton, UK, April 1997).

[28] GUPTA, H., AND MUMICK, I. S. Selection of views to materialize under a maintenance cost constraint. In *Intl. Conf. on Database Theory* (1999), pp. 453–470.

[29] HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. Implementing data cubes efficiently. In *ACM SIGMOD Intl. Conf. on Management of Data* (Montreal, Canada, June 1996).

[30] HULGERI, A., SESHADRI, S., AND SUDARSHAN, S. Memory cognizant query optimization. In *International Conference on Management of Data (COMAD)* (2000). (to appear).

[31] KAPITSKAIA, O., NG, R. T., AND SRIVASTAVA, D. Evolution and Revolutions in LDAP Directory Caches. In *Intl. Conf. on Extending Database Technology (EDBT)* (2000).

[32] KELLER, A. M., AND BASU, J. A predicate-based caching scheme for client-server database architectures. *VLDB Journal 5*, 1 (1996).

[33] KOTIDIS, Y., AND ROUSSOPOULOS, N. DynaMat: A dynamic view management system for data warehouses. In *ACM SIGMOD Intl. Conf. on Management of Data* (1999).

[34] LABIO, W., QUASS, D., AND ADELBERG, B. Physical database design for data warehouses. In *Intl. Conf. on Data Engineering* (1997).

[35] LARSON, P. A., AND YANG, H. Z. Computing queries from derived relations. In *Intl. Conf. Very Large Databases* (Stockholm, 1985), pp. 259–269.

[36] LEHNER, W., SIDLE, R., PIRAHESH, H., AND COCHRANE, R. Maintenance of Automatic Summary Tables in IBM DB2/UDB. In *ACM SIGMOD Intl. Conf. on Management of Data* (2000).

[37] MUMICK, I. S., QUASS, D., AND MUMICK, B. S. Maintenance of data cubes and summary tables in a warehouse. In *ACM SIGMOD Intl. Conf. on Management of Data* (1997), pp. 100–111.

[38] PARK, J., AND SEGEV, A. Using common sub-expressions to optimize multiple queries. In *Intl. Conf. on Data Engineering* (Feb. 1988).

[39] PELLENKOFT, A., GALINDO-LEGARIA, C. A., AND KERSTEN, M. The Complexity of Transformation-Based Join Enumeration. In *Intl. Conf. Very Large Databases* (Athens,Greece, 1997), pp. 306–315.

[40] PIRAHESH, H., HELLERSTEIN, J. M., AND HASAN, W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *ACM SIGMOD Intl. Conf. on Management of Data* (San Diego, 1992), pp. 39–48.

[41] POOSALA, V., IOANNIDIS, Y., HAAS, P., AND SHEKITA, E. Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD Intl. Conf. on Management of Data* (1996).

[42] QUASS, D., GUPTA, A., MUMICK, I., AND WIDOM, J. Making views self-maintainable for data warehousing. In *Intl. Conf. on Parallel and Distributed Information Systems* (1996).

[43] RAO, J., AND ROSS, K. Reusing invariants: A new strategy for correlated queries. In *ACM SIGMOD Intl. Conf. on Management of Data* (1998).

[44] ROSS, K., SRIVASTAVA, D., AND SUDARSHAN, S. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM SIGMOD Intl. Conf. on Management of Data* (May 1996).

[45] ROUSSOPOLOUS, N. View indexing in relational databases. *ACM Trans. on Database Systems 7*, 2 (1982), 258–290.

[46] ROY, P., SESHADRI, S., SUDARSHAN, S., AND ASHWIN, S. Garbage collection in object oriented databases using transactional cyclic reference counting. *VLDB Journal 7*, 3 (1998).

[47] ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBHE, S. Efficient and extensible algorithms for multi-query optimization. In *ACM SIGMOD Intl. Conf. on Management of Data* (2000).

[48] SALEM, K., BAYER, K., COCHRANE, R., AND LINDSAY, B. How to roll a join: Asynchronous incremental view maintenance. In *ACM SIGMOD Intl. Conf. on Management of Data* (2000).

[49] SCHEUERMANN, P., SHIM, J., AND VINGRALEK, R. WATCHMAN: A data warehouse intelligent cache manager. In *Intl. Conf. Very Large Databases* (1996).

[50] SCHEUERMANN, P., SHIM, J., AND VINGRALEK, R. Dynamic caching of query results for decision support systems. In *Intl. Conf. on Scientific and Statistical Database Management* (1999).

[51] SELINGER, P., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. Access path selection in a relational database management system. In *ACM SIGMOD Intl. Conf. on Management of Data* (1979), pp. 23–34.

[52] SELLIS, T. Intelligent caching and indexing techniques for relational database systems. *Information Systems* (1988), 175–185.

[53] SELLIS, T., AND GHOSH, S. On the multi query optimization problem. *IEEE Transactions on Knowledge and Data Engineering* (June 1990), 262–266.

[54] SELLIS, T. K. Multiple query optimization. *ACM Transactions on Database Systems 13*, 1 (Mar. 1988), 23–52.

[55] SESHADRI, P., PIRAHESH, H., AND LEUNG, T. Y. C. Complex query decorrelation. In *Intl. Conf. on Data Engineering* (1996).

[56] SHIM, K., SELLIS, T., AND NAU, D. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering 12* (1994), 197–222.

[57] SHUKLA, A., DESHPANDE, P., AND NAUGHTON, J. F. Materialized view selection for multidimensional datasets. In *Intl. Conf. Very Large Databases* (New York City, NY, 1998).

[58] SOUKUP, R., AND DELANEY, K. *Inside Microsoft SQL Server 7.0*. Microsoft Press, 1999.

[59] SUBRAMANIAN, S. N., AND VENKATARAMAN, S. Cost based optimization of decision support queries using transient views. In *ACM SIGMOD Intl. Conf. on Management of Data* (1998).

[60] TPC. TPC-D Benchmark Specification, Version 2.1, Apr. 1999.

[61] VISTA, D. Integration of incremental view maintenance into query optimizers. In *Intl. Conf. on Extending Database Technology (EDBT)* (1998).

[62] YANG, H. Z., AND LARSON, P. A. Query transformation for psj queries. In *Intl. Conf. Very Large Databases* (Brighton, August 1987), pp. 245–254.

[63] YANG, J., KARLAPALEM, K., AND LI, Q. Algorithms for materialized view design in data warehousing environment. In *Intl. Conf. Very Large Databases* (1997).

[64] ZHAO, Y., DESHPANDE, P., NAUGHTON, J. F., AND SHUKLA, A. Simultaneous optimization and evaluation of multiple dimensional queries. In *ACM SIGMOD Intl. Conf. on Management of Data* (Seattle, WA, 1998).

Acknowledgements

I thank S. Sudarshan and S. Seshadri for introducing me to the field of databases, and for their continuous enthusiasm, patience and guidance for the last five years. I have been very fortunate to have Sudarshan as my thesis advisor; his appreciation and understanding were necessary to drive things till the finishing line. Many thanks to Krithi Ramamritham for his interest, encouragement and insights. It was a pleasure working with him. I thank D.B. Phatak for inducting me into the fold of IIT-Bombay; but for him, I would have missed a lot. Moreover, I have valued his encouragement and support during my entire stay.

The Informatics Lab at IIT-Bombay is a fun place to work in, thanks to the excellent graduate and undergraduate students working here. I thank all my labmates, past and present, with whom I have had the chance to work with during my stay; in particular, P.P.S. Narayan – who taught me a lot about *real* system development – and Siddhesh Bhobe, Pradeep Shenoy and Hoshi Mistry who collaborated with me on parts of this thesis. Thanks to fellow Ph.D. students Bharat Adsul and Arvind Hulgeri for their company. I thank Arvind further for our several technical discussions; they helped a lot.

I am grateful to Paul Larson for calling me all the way to Redmond for a summer internship at Microsoft Research, and for giving me a chance to hack into the Microsoft SQL-Server code and prototype my ideas; it was a very valuable experience.

This work was supported in part by an IBM Ph.D. fellowship.

Prasan Roy