Anatomy of a Modular Multiple Query Optimizer

Arnon Rosenthal, Upen S. Chakravarthy

Computer Corporation of America
Four Cambridge Center
Cambridge, MA 02142.

ARPANet Address: last name@cca.cca.com

Abstract

We critically evaluate the current state of research in multiple query optimization, synthesize the requirements for a modular optimizer, and propose an architecture. Our objective is to facilitate future research by providing modular subproblems and a good general-purpose data structure. In the context of this architecture, we provide an improved subsumption algorithm, and discuss migration paths from single-query to multiple-query optimizers.

The architecture has three key ingredients. First, each type of work is performed at an appropriate level of abstraction. Second, a uniform and very compact representation stores all candidate strategies. Finally, search is handled as a discrete optimization problem separable from the query processing tasks.

1. Problem Definition and Objectives

A *multiple query optimizer (MQO)* takes several queries as input and seeks to generate a good *multi-strategy*, an executable operator graph that simultaneously computes answers to all the queries. The idea is to save by evaluating common subexpressions only once. The commonalities to be exploited include identical selections and joins, predicates that subsume other predicates, and also costly physical operators such as relation scans and sorts. The *multiple query optimization problem* is to find a multi-strategy that minimizes the total cost (with overlap exploited). Figure 1.1 shows a multi-strategy generated exploiting commonalities among queries Q1-Q3 at both the logical and physical level.

To be really satisfactory, a multi-query optimization algorithm must offer *solution quality, efficiency*, and *ease of implementation*. Solution quality requires that the optimizer: produce good *1-strategies* (strategies for a single query or single-query strategies); identify many kinds of commonalities (e.g., by predicate splitting, sharing relation scans); and search effectively to choose a good combination of 1-strategies. Efficiency requires that the optimization avoid a combinatorial explosion of possibilities, and that within those it considers, redundant work on common subexpressions be minimized. Finally, ease of implementation is crucial - an algorithm will be practically useful only if it is conceptually simple, easy to attach to an optimizer, and requires relatively little additional software.

Our goal is to suggest a way that research on MQO can be organized. Current state of research is not close to supporting a really satisfactory solution. At this early stage, it seems important that new techniques be modular, and that critical data structures be separated from specific algorithms. We also prefer to understand the entire space of legal solutions before developing heuristics that will limit the search.

The paper is structured as follows. Section 2 argues that there are many applications that generate queries suitable for MQO. The same techniques are likely to reduce *optimization time* spent by a physical database designer. Section 3 surveys relevant previous work, and identifies the need for a stronger architectural framework. Section 4 provides an overview of the proposed architecture, describes a uniform, compact representation for candidate strategies, describes a new transformation that creates common subexpressions, comments briefly on search issues and discusses evolution paths from single query optimizers (SQOs) to MQOs. Section 5 contains conclusions.

2. Applications

The literature describes sets of queries where MQO yields substantial savings, but does not indicate where such combinations could be expected. We identify here applications scenarios that generate many overlapping queries, so that the payoff from MQO is likely to be substantial.

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

```
Relations:
    CUST(c#, name, age, zipcode)
    ORDER(o#, c#, qty)

Q1: select *
    from CUST
    where credit = 'bad'

Q2: select *
    from CUST
    where age < 40
    order by zipcode

Q3: select *
    from CUST, ORDER
    where age < 50 and CUST.c# = ORDER.c#
    order by zipcode
```

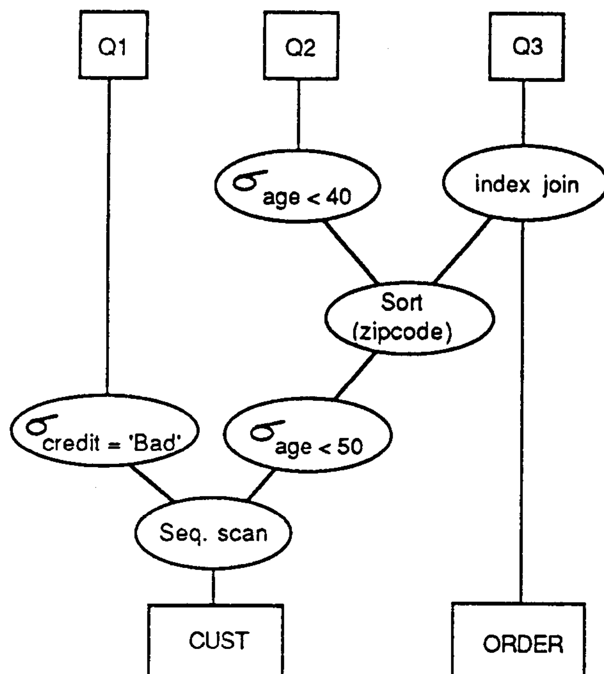

Figure 1.1

1. Sequential File Processing: Traditional data processing often batches several requests to be executed in one pass over a sequential master file (e.g., CUST in Figure 1.1). Such behavior is awkward to achieve with current relational systems. To avoid issuing separate queries (which Scan and Sort redundantly), one must embed all the requests in a host-language program that scans the master file and invokes all the types of additional processing. This requires knowledge of a host language, and can be inefficient due to passing tuples between the DBMS and programming language environments. Furthermore, it places the optimization burden on the pro-grammer.

The following applications, though not formulated as an MQO problem, can benefit from the use of MQO techniques.

2. Condition Monitoring: CCA is designing a High Performance Active Database (HiPAC [DAYA88]) which is capable of monitoring multiple conditions on a database. A condition is encapsulated in a rule which is a triple of the form <event, condition, action>. where condition may reference information both in the event and in the database. A signal is generated if the condition is non-null. We expect that sets of conditions will often monitor different aspects of a general situation, and hence will have substantial commonality. When a set of conditions is first imposed, they may be simultaneously evaluated on the current state of the database. During monitoring, the problem of evaluating all conditions triggered by the same signal (e.g., new radar blip) can also benefit from exploiting commonalities.

3. Supporting application-oriented objects: Application-oriented interfaces permit definition of operations on objects in the user's mental model. Under the covers, a user object may be represented as sets of database views. Evaluation of these views is likely to generate overlapping queries.

For example, consider a transaction by which a documentation group downloads specifications and drawings for some product. Suppose that in the stored database, Documents and Drawings are associated with Parts and a Product contains many Parts. To make things more manageable, views Prod_Specs(Prod#, Spec#, Author) and Prod_Drawings(Prod#, Draw#, Artist) have been defined, e.g.,

```
View Prod_Specs =
   [Prod_Part(Part#,Prod#)join part#
   Specs(Doc#, Author, Part#)
```

The definition of Prod_Drawings is similar. Now the download transaction for product 1234 is:

```
[Select {Prod_Specs| Prod#= 1234};
Select {Prod_Drawings| Prod#= 1234} ]
```

When expanded, the two views represent queries with very substantial overlap.

In this example, there is no natural way to combine the two views into one relation — joining them yields a Cartesian product of Specs and Drawings for each Part. If instead the two views were union compatible (i.e., had the same column attributes), one could have formed the union

view. In this case, MQO techniques would need to be applied *within a single query* to detect common subexpressions.[1]

4. Logic programs/database views: View definitions that are composed of union compatible expressions (or equivalently intensionally defined predicates in logic programs) are likely to have substantial overlap. Evaluation of a query on such views (or predicates) can use multiple query evaluation techniques for exploiting intra-query commonalities.
As an example, the query (select [Employed where nationality .not equal. US]) has substantial overlap when the following definitions are used.

```
Base relations:

   Personnel(id, name, nationality)
   Faculty(id, rank, compensation)
   Staff(id, rank, compensation)

View Definition:

   Employed = (Faculty join Personnel)
     union (Staff join Personnel)
```

5. Database Design: In physical database design, the cost of a user query must be determined for many different decisions about supported access paths. The user wants a set of separate 1-strategies, not a multi-strategy. Our interest is to reduce the *optimization time* by using MQO techniques to avoid repeating the analysis of common subexpressions.

For each "logical" relation R, we create separate relations R_1, R_2, ... corresponding to the different options for access path selection. Now evaluation of different design choices means performing a set of single query optimizations involving different R_i. We would expect extensive commonality among the different queries.

3. Previous Work

Multiple query optimization (also called global query optimization) has been investigated in both database and logic programming. Section 3.1 summarizes important MQO techniques in the literature. Section 3.2 discusses why additional work is needed.

[1] Any MQO problem can be converted to an SQO problem by taking "outer union" of all query results: however, the conversion does not seem to aid the solution.

3.1 Applicable Techniques from Previous MQO Research

Most published MQO techniques can be broadly classified as: a) techniques for finding/exploiting commonality, and b) search techniques for choosing a multi-query strategy. This section distills useful techniques from the literature; it is not a historical survey.

Commonality Finding: This can be further broken down into (i) finding subexpressions that are same (identical) among queries and (ii) finding whether one expression subsumes another expression. (A query or subexpression e1 *subsumes* expression e2 if the result of e2 is contained in the result of e1.)

Most work on subsumption has focused on predicates and projections. [FINK82] compared an incoming query with materialized results (from earlier queries); in his application there was no need to detect subsumption of subexpressions. [CHAK86] identified equivalent and subsuming subexpressions by analyzing the query graph of a query [SELI79]. [SELL86] uses equivalence of subexpressions at the join level and subsumption at the selection level. Other kinds of subsumptions are possible — both logical (outerjoin subsumes join) and physical (Sort subsumes Group-by; a Sort on multiple attributes can subsume a Sort on a single attribute).

Search techniques: Search for an optimum usually begins with a set of 1-strategies, such as might be produced by a modified single query optimizer, plus information about equivalence of subexpressions. One difficulty is that the number of strategies can be large. State-space search algorithms are proposed in [GRAN80, SELL86], with heuristics (e.g., the A* algorithm from artificial intelligence) to reduce the labor. In contrast, [CHAK86] uses heur-istics to directly generate a strategy without search. (In this work, multi-strategies are expressed directly as pseudocode, rather than as an operator graph). Direct generation is very simple but is more likely to produce a sub-optimal strategy.

Techniques from single-query optimization will also be useful in our work. In particular, we use an AND-OR operator graph [ROSE82, ROSE86] to represents alternative strategies and their commonalities, and divide the optimization process into explicit levels of abstraction (essentially the same levels as are used in ([BATO87, FREY87, GRAE87]).

3.2 What More is Needed to Provide a Robust MQO Architecture

In this section, we discuss some problems with existing approaches, that need to be resolved before a powerful, modular MQO could be built. The main focus of this paper is an architecture with clear subproblems, so that results

on those subproblems could eventually be combined into an overall architecture.

Most existing approaches use a three-step procedure: 1) generate a set of alternative 1-strategies; 2) identify commonalities (exploiting equivalence and subsumption among subexpressions); 3) search for an optimum multi-strategy. Each step tends to be indivisible, and to assume that all operators and heuristics are at the same level of abstraction. Below, we describe reasons why this approach cannot be fully satisfactory (Our main point of reference will be [SELL86], since it proposes the most complete and appealing solutions to date).

Modularity: It is hard to "mix and match" to combine techniques from different papers. We need to define natural, limited subproblems on which individual researchers could concentrate. We also need interfaces and common representations that would permit results to be combined. For example, there is no common model in which one might combine query-graph techniques for subsumption [CHAK86] with sets of strategies in [SELL86]. And improving the search algorithm is difficult when it is intertwined with exploitation of commonalities, as in [CHAK86] and most SQOs.

Organizing work by levels of abstraction: Figure 1.1 shows a multi-strategy that exploits commonalities in both logical-level operations (e.g., selections) and physical operations (sorts and scans). It is necessary to find both types of commonalities. The problem of organizing this process is not dealt with explicitly resulting in limited exploitation of commonalities [CHAK86, SELL86].

Existing algorithms do all their work at one level of abstraction. [CHAK86] detect commonalities only at logical level. This results in ignoring sharing at the physical level (e.g., tradeoffs about whether each scan and sort *ought* to be shared). [SELL86] seems to elaborate entirely to physical level, then detect/use commonalities resulting in analysis over a vastly expanded graph and complex implementation.

Representation: Most existing work generates sets of 1-strategies. This is a natural representation when using the results of an existing SQO, but has problems in convenience and efficiency. First, algorithms need to deal conceptually with two data structures, since one needs an associated data structure to identify equivalent subexpressions. Second, if one wants to perform a fairly complete search, there is tremendous redundancy — a low-level subexpression may be repeated many times.

The architectural issues (above) are the focus of this paper, and we hope our solution can amplify the effectiveness of future research. We identify narrower, more specific problems for further research; the architecture helps ensure that the results will combine easily. We also give a new algorithm for creating and merging common subexpressions. Furthermore, we discuss how MQO architectures relate to architectures for the "bread and butter" problem of single query optimization.

4. Architectural Description

4.1 Overview of Our Approach to MQO

Our approach has three key ingredients: *multilevel decomposition* by level of abstraction; an "AND-OR operator graph" that provides an *efficient representation* of alternative strategies and their commonalities; separation of search from the generation of the strategy set.

1. Multilevel decomposition: MQO must apply heuristics and query transformations at several levels of abstraction. There is a *logical level (or join-level)*, corresponding roughly to the implementable relational algebra. The principal operators here are selection, projection, and 2-input joins. There is a *physical level* that includes relation scans and sorting, and distinguishes different implementations of join. The initial queries can be considered to be at zero'th level, at which each query is a single operator and leaf nodes (base relations) are merged. Other levels are possible, to deal with higher level processing and optimizations (nested subqueries, high-level recursion operators), but will not be discussed here.

Our idea is that each level of processing starts and ends with a graph that contains (as subgraphs) all the interesting 1-strategies for all the queries. To pass between levels, it is necessary to:

- *Elaborate* nodes to substrategies at the next more detailed level
- *Create additional alternatives.* An MQO must consider types of 1-strategies that would not be of interest in SQO, e.g., to delay a selection in one query to permit the result to be shared with another. Another example is to exploit subsumption by splitting an operation into two, one of which may be shared.
- *Finding and merging identical subexpressions:* By merging them at this level, we avoid the need to track the relationship at the next more detailed level. We also avoid repeated analysis of the same subexpression.

The above list of tasks is not intended as an algorithm. Rather, at each level the implementer can choose any algorithm appropriate for that level. For example, at the physical level SQO techniques that group results by "interesting order" may be used; at the logical level,

[CHAK86] exploits the query graph [SELI79] to identify shared subexpressions.

2. Efficient representation of alternative strategies and their commonalities: We need an efficient way to generate and represent {all 1-strategies} that minimizes redundancy. MQO algorithms constantly need to recognize equivalent subexpressions and (ideally) to avoid processing each of the equivalents separately.

We meet these requirements through an operator graph, here called the AND-OR graph. It contains operator nodes (ANDs, since all inputs are needed) and "OR" nodes that represent the fact that any of a set of alternative inputs can be used. (See section 4.2 for more details). Common subexpressions are represented just once.

We intend to use the AND-OR graph at all levels. Each node that appears in an AND-OR graph appears in one (or more) 1-strategies. In Section 4.4 we compare the AND-OR graph with an explicit set of 1-strategies, from the point of view of search algorithms.

3. Search: Our main contribution on search is architectural — it is treated as a strictly combinatorial problem decoupled from the query processing work needed to generate the AND-OR graph. In an architectural context, the contribution is that those search algorithms can be compared directly with other pure search algorithms. A side benefit is that the AND-OR graph gives a sense of the entire space, so one can judge how much is lost by various search heuristics.

The architecture is tied together by a very simple algorithm.

```
Create an AND-OR operator graph at
level zero for a set of queries.

Repeat for each level of abstraction

  Elaborate the graph to the next (more
  detailed) level of abstraction. In-
  clude transformations that create
  subgraphs for new alterna-tives
  (e.g., operator movement and split-
  ting). Merge common subexpressions.

Until (the most detailed level of ab-
straction)

Evaluate operator costs[3] and Search for
an optimal multi-strategy
```

[3] For example, Figure 4.2 has 27 nodes representing 8 1-strategies, but 56 nodes would be needed if 1-strategies were separately represented.

To summarize, our architecture requires: (i) use of AND-OR graphs for representing sharing and alternate strategies; (i) ability to generate 1-strategies not considered in SQO, (iii) ability to retain 1-strategies that would be deleted as suboptimal by SQO; (iv) Modularity — ability to replace processing modules for any level of abstraction, or for search.

More details of the AND-OR graph are given in 4.2. Section 4.3 comments on transformations performed at each level, and then presents a new algorithm for creating common subexpressions. The combinatorial search problem is discussed in Section 4.4. Section 4.5 discusses migration from an SQO to an MQO.

4.2 AND-OR Operator Graph and Multi-strategies

An *AND-OR operator-data graph* is an acyclic digraph composed of *operator nodes* (e.g., join, sort) and *data nodes* (inputs to and results from operator nodes). During optimization, the currently-known situation is represented as an AND-OR operator graph, denoted by G.

A subgraph is *well-formed* if for each of its operator nodes (AND nodes) it contains all input edges and for each non-leaf data node (OR node) it includes one input edge, representing a choice of how to compute the result of the data node. We will abbreviate by using "subgraph" for well-formed subgraph, and AND-OR graph for "AND-OR operator-data graph". An *elementary subgraph* is a graph with one operator node, its input and output data nodes, and edges connecting them. Diagrams use rectangles for data nodes, ovals for operator nodes. Data nodes (i.e., OR nodes) of indegree 1 are shown only if they are query results.

A *1-strategy* for query Q is a subgraph containing the result node for Q as the only node with out-degree 0; a *multi-strategy* for Q_1, Q_2,... is a subgraph containing the result nodes for the indicated queries as the only nodes with out-degree 0); it is analogous to a *solution graph* of an AND/OR graph [NILS80]. [2] 1-strategies for intermediate nodes will generally be called *subexpressions.*

Figure 4.1 illustrates an AND-OR operator graph for three queries (Q1-Q3 in Figure 1.1) at the logical level. For economy, we have omitted most of the alternatives for Q3. As is evident from the figure, shared subexpressions need be represented only once. Therefore the representation is compact[3], and no extra data structure is needed to represent equivalence between subexpressions. The AND-OR graph produced by an optimizer is an indication of the power of

[2] In AI the edges would be directed down; in database, up. Since we never use the direction, we show them without arrows. For a closer analogy with AI AND/OR graphs, a dummy AND node can receive input from each query node.

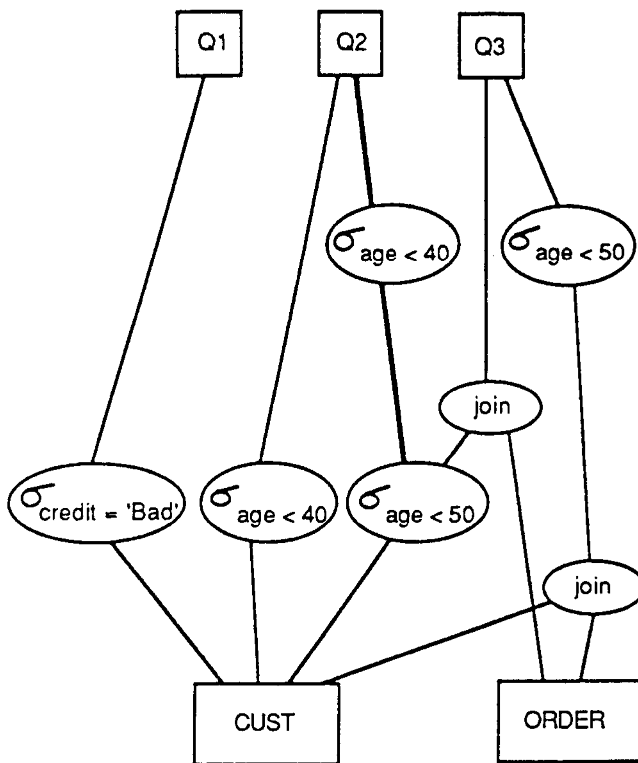

Figure 4.1

the optimizer. If operator costs have been computed, pruning heuristics can be applied.

Once the physical-level graph has been generated, search for an optimal multistrategy is a separate subproblem. Furthermore, the graph is usually much smaller than the total number of nodes in all 1-strategies (let alone the number of multi-strategies), so there is relatively little extra cost in generating it explicitly. However, if for some subexpression it is known that no node can be shared between two queries, then if the subexpression is suboptimal for its data node the edge into the data node can be ignored.

4.3 Transforms to Exploit Commonality

Several types of transformation are relevant at each level of abstraction. The transformations (at any level of abstraction) can be broadly classified into: a) elaboration transformations - that generate many alternative implementations b) algebraic transformations -that use properties of the operator to improve strategies, and c) commonality-finding transformations - that detect, create and merge common subexpressions among queries.

Typical kinds of commonalities that are exploited at the logical level are equivalence of subexpressions and subsumption. Merging of equivalent subexpressions is the simplest case of commonality and is exploited widely in earlier work. Subsumption of expressions (based on operator semantics) can create additional shared subexpressions. Earlier attempts do not exploit subsumption to the fullest extent. Below we provide an algorithm for creating additional subexpressions.

The join-level graph is then elaborated to the physical level. At this level, operations like relation scans and sorts are considered sharable among queries as shown in Figure 4.2 for queries in Figure 1.1. (Most proposals do not consider such sharing). The benefits of sharing (or not sharing) these low-level operators will be visible to and exploited by the search process, which runs off the physical-level AND-OR graph.

4.3.1 Subsumption Based Algorithm for Creating Shared Subexpressions

This section presents a new algorithm that *creates* large shared subexpressions exploiting subsumption at the smaller expression level. Although the algorithm is described at the logical level of abstraction, it is based on the general properties of operators involved and hence is equally applicable at other levels also. As an example, at the logical level selection conditions can be split and propagated beyond join. As another example, at the physical level, selections can be moved beyond sorts to identify a large subexpression.

Our algorithm extends and generalizes the work of [FINK82]. In [FINK82] a test is made to detect whether a given query is subsumed by a previously computed query in order to utilize the temporary created by the query. That algorithm works at the query level and hence cannot make use of subsumption at the subexpression level. On the other hand, [SELL86] concentrates on the use of identical subexpressions and detects and uses subsumption only at the selection level.

Algorithm:

/* We assume that any pair of operators can be tested. At the logical level, this means that selection and join conditions can be tested for equivalence and subsumption. Similarly, at the physical level selection and sorts can be tested for equivalence and subsumption. Furthermore, when an expression e1 subsumes e2, we assume that e1/e2 which is the remainder of e1 that differs from e2 is available. */

/* To simplify the presentation, we omit the steps that detect and merge equivalent subexpressions. */

Perform the following transformations

for each pair of operator nodes n1 and n2 (from distinct queries) that are roots of an elementary subgraph and share common input data nodes do

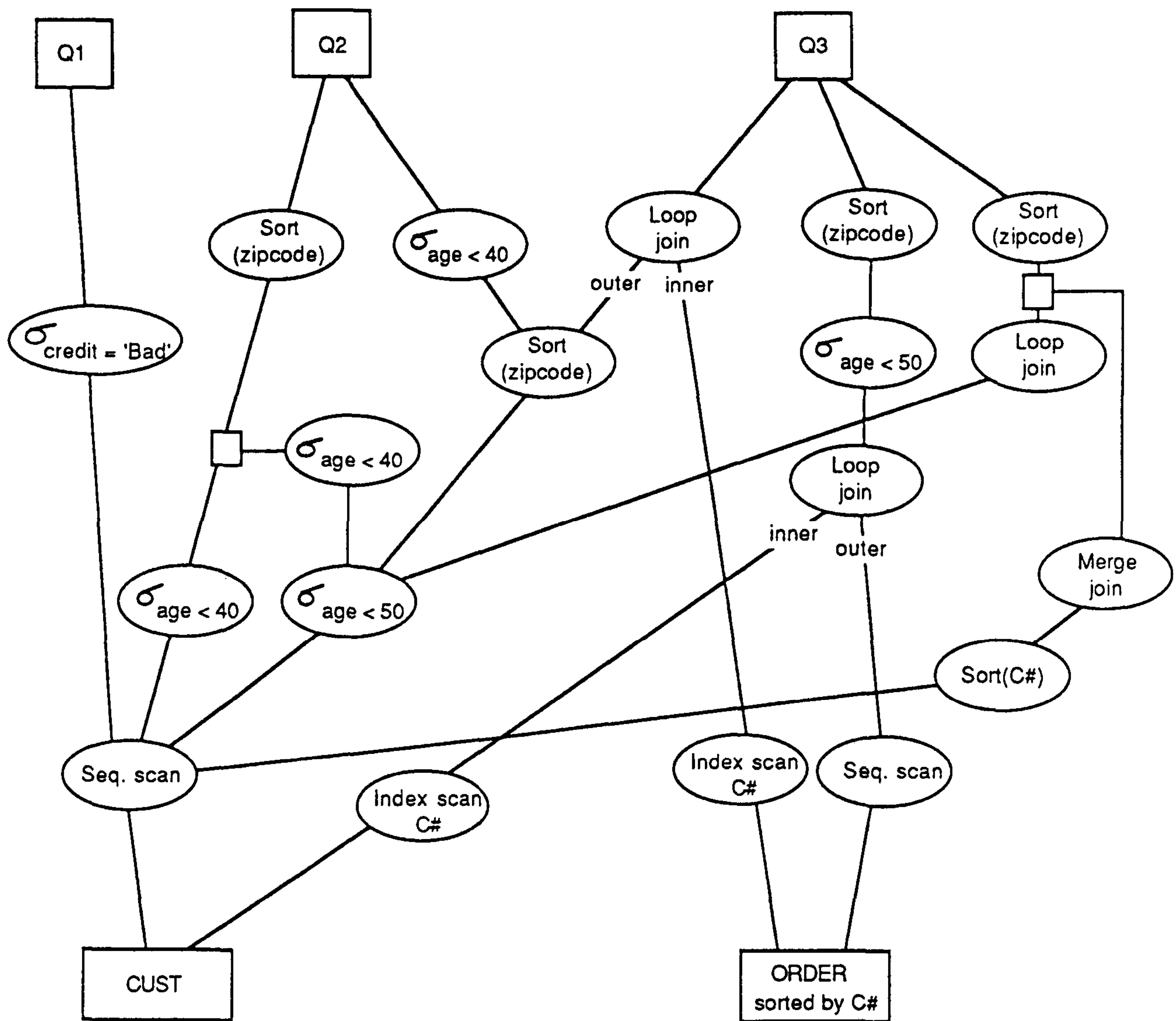

Figure 4.2

case:

operators are unary (e.g., select):

let sc1 and sc2 be the selection conditions for n1 and n2 respectively.

If sc1 subsumes sc2 then create an elementary graph with sc2 and create a new operator node sc1/sc2 which receives the result of the node sc2.

operators are binary (e.g., join)*

let j1 and j2 be the operators at the root nodes; let sc11, sc12 and sc21, sc22 be the selection condition, if any, along the two inputs (true is a valid selection condition).

If sc11 subsumes sc21 and j1 subsumes j2 then push sc11/sc21 and j1/j2 beyond the join (j2) and merge. If sc12 subsumes sc22 and j1 subsumes j2 then push sc12/sc22 and j1/j2 beyond the join (j2) and merge.

until no more transformations are applicable

Figure 4.4 is obtained after the application of the above algorithm to Figure 4.3. Two additional strategies (one for

* In practice the cases under the binary operator are combined for efficiency; other subsumption possibilities are omitted for the sake of brevity.

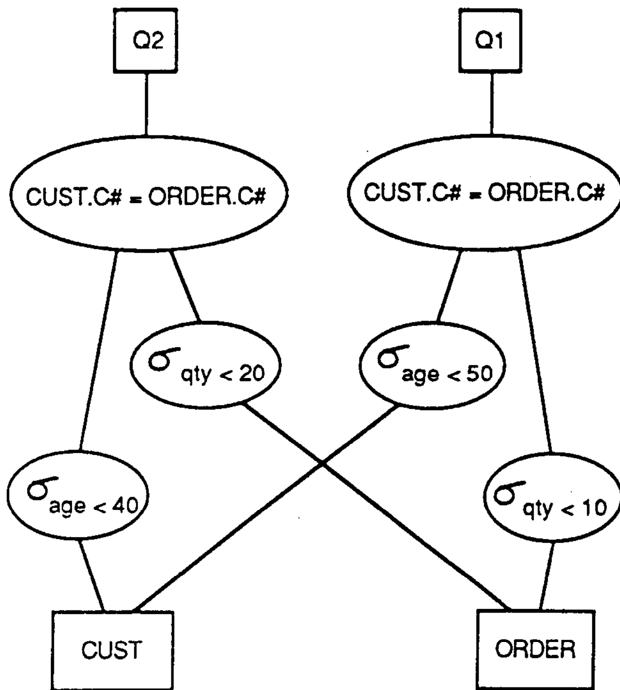

Figure 4.3

each query) are created which share a large subexpression consisting of a join and two selections. To summarize, the algorithm progressively creates larger shared subexpressions applying the transformations outlined on elementary subgraphs. Conditions belonging to different queries are split in order to detect a larger equivalent subexpression.

4.4 Search

The search problem is to take the AND-OR operator graph produced above and find the cheapest solution subgraph, i.e., multi-strategy. This strictly combinatorial problem can be attacked without complications from query elaboration and commonality detection.

The problem of finding an optimal multi-strategy is NP-hard. (It is equivalent to finding a minimum-weight subgraph of an AND/OR graph [GARE79]). Hence we must resort to heuristic solutions.

Sharing is the heart of the computational difficulty. Without shared intermediate results, the search problem could be decomposed into separate search for each query, and time would be linear in the size of the AND-OR graph. The multi-query problem can be made more decomposable if shared nodes are deleted from the graph — this can sometimes be achieved by examining bounds on costs of operators and substrategies.

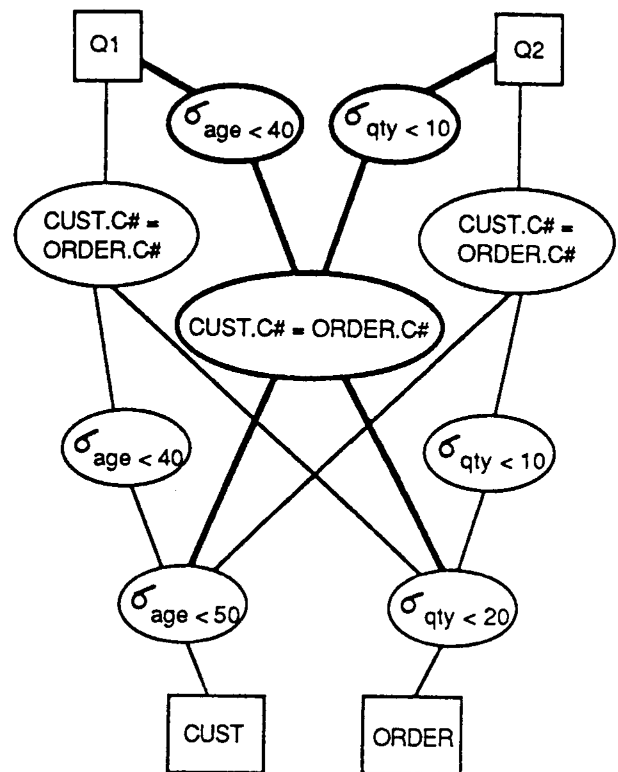

Figure 4.4

How Suitable is the AND-OR Graph for Search?

Some search algorithms run better on an AND-OR graph than on a separate list of {1-strategies that are subgraphs of the AND-OR graph}. The necessary condition for a multi-strategy search algorithm to run on the AND-OR graph is that its steps be concerned only with the cost of producing a resulting data node, and not with knowing what vertices were used within the subexpression. We have identified several useful computations that compute bounds by making extreme assumptions about sharing, e.g., that no nodes are shared, or that all shared nodes are available free. These bounds can then be used to determined that certain nodes and edges do not appear in the optimum multi-strategy, and hence may be deleted from the graph. (The exact arguments are technical and too long for inclusion here).

Other search algorithms need more information about the interior of a subgraph. The complete set of 1-strategies can always be generated from G. As a middle ground, a dynamic programming solution would need to know which sharable intermediate results had been made available.

4.5 Compatibility with SQO Architectures

A single-query optimizer is a necessary part of any DBMS that supports a high-level language. If it is ever implemented, MQO will be an add-on. Hence, we first propose an SQO that can be easily extended to an MQO. We then address the problem of extending currently available traditional [SELI79] and extensible optimizers [BATO87, GRAE87, FREY87] to an MQO.

What is the Minimum Extension Required?

One approach is to add a front or back end to an existing SQO [FINK82, SELL86]. This is easy to implement, but will not always produce a good multi-strategy, since the optimum multi-strategy may not be a composition of optimal 1-strategies. In fact, the optimum may involve 1-strategies that would not even be considered by the SQO.

An MQO needs several capabilities that are not present in an SQO — generation of additional 1-strategies to increase the opportunities for sharing, recognition and merging of equivalent subexpressions, and search algorithms that examine 1-strategies (in the context of the AND-OR graph) to find the optimum *multi*-strategy.

To facilitate extension to MQO, an SQO would ideally be divided into levels of abstraction matching the MQO design, and the interface at each level would be an explicit AND-OR graph for the entire query. Creation of new 1-strategies and recognition of commonalities need to be added at each level.

At the logical level, there would be two ways to implement the creation and merging of the new strategies. First, one could take the AND-OR graph output by the logical level of the SQO and apply transformations to generate and merge the new strategies. Alternatively, one could go inside the SQO routines for this stage. The algorithm that analyzes the query graph and produces join strategies at the logical level could be extended to consider multiple queries simultaneously, and generate the AND-OR graph with all commonalities exploited. [CHAK86] shows how such an algorithm can generate a single multi-strategy combining all the queries; the algorithm could be extended to consider all possibilities.

At the physical level, new alternatives need to be generated reordered selections and sorts, and subsumptions among multiattribute sorts. Also, it is natural for an SQO to discard all but the cheapest path into a data node. For MQO the alternatives need to be retained if they contain intermediate results that might be shared with other queries.

MQO also has implications for the run-time control of execution strategies. Many DBMSs use tuple streams to buffer intermediate results. A simple control protocol is to have each stream produce the next tuple on demand from its consumer. Unfortunately, this implies that each stream has a single consumer, violating the spirit of MQO (as in Figure 1.1). Instead, MQO appears to need a more data-driven protocol including extensive buffering and synchronization.

Extending Conventional Optimizers

Traditional SQOs (as in System R [SELI79]) do not proceed one level at a time — instead, they use a depth-first approach that elaborates a logical-level node as soon as it is generated. This depth-first approach is not compatible with finding commonalities at the logical level — one would need to apply all multi-query analysis to the graph composed of physical operators.

One is forced to thoroughly rewrite the search and control routines to permit layering work in our levels. The routines that elaborate to the join level, and also to the physical level would survive pretty well, although they would need the extensions for new functionality (discussed above). The model of operator costs should also survive.

An alternative is to just change the search to retain suboptimal strategies. All strategies would be elaborated to the physical level, and only then would inter-query commonalities be processed. This approach is feasible, though working at the physical level means a cost in both speed and complexity, as discussed in Section 3.2.

Extending 'Extensible' Optimizers

A new generation of "extensible" optimizers are described in [BATO87, FREY87, GRAE87, ROSE86]. The idea is that the optimizer has a general-purpose control algorithm, plus an easily extended library of transformations. For example, one could write transformations that create new, sharable subexpressions, and other transformations that combine them.

But transformations must be executed in an ap-propriate order, e.g., common subexpressions merged be-fore the next level of elaboration. Extensible optimizers' control algorithms generally do not allow additional control rules, or have a convenient notion of level of abstraction or priority.[4] Another problem lies in the cost model, which generally does not adjust for shared subexpressions. Finally, it is not clear whether the transformations' pattern matchers will always be able to search effectively for commonalities — instead of searching for a local pattern, the search must compare patterns.

[4] Exodus has two levels, but the physical ("method") level is second class — one does not transform physical strategies, but just creates them [GRAE87].

5. Conclusions

As multiple query processing research is in its early stages, it is important to have a framework which is modular and accommodates different solutions to subproblems. We discuss here how the approach helps meet the Introduction's criteria for solution quality, and ameliorates the problems identified in Section 3.2.

1. *Allows mix and match of techniques:* We have separated several independent subproblems — strategy generation at each level of abstraction, plus the search problem. The AND-OR graph acts as the interface between steps. Special techniques and data structures (e.g., query graphs) can be applied at any level.

2. *Work at appropriate level of abstraction:* Logical commonalities are handled at the logical level (or earlier), on a small graph without physical complications. SQO elaboration techniques can be applied to obtain the physical level, and additional commonalities can be detected.

3. *Uniform and efficient representation:* The AND-OR operator graph provides a very efficient and convenient way to represent all useful information about 1-strategies and their common subexpressions. It is very small compared with {all 1-strategies} or {all multi-strategies}. It is suitable as an interface at all levels, and is not restricted to SPJ queries. 1-strategies can easily be generated from our graph.

4. *Solution Quality and Optimizer Efficiency:* The main ingredients in finding a good solution are a wide range of possibilities, and a search that is fast enough to consider many of them. The approach's modularity helps in introducing new transformations and new search techniques. Other aspects that speed the search (and make possible a more thorough optimization) are being investigated.

6. Acknowledgements

The authors thank Umesh Dayal for helpful comments on an earlier version of the paper.

7. References

[BATO87]
D. Batory. "A Molecular Database Systems Technology," Computer Science Department TR-87-23, University of Texas at Austin (1987).

[CHAK86]
U. S. Chakravarthy and J. Minker. "Multiple Query Processing in Deductive Databases using Query Graphs," *Proc. of Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan (August 1986).

[DAYA88]
Dayal, U., et al. "The HiPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD-Record* (March 1988).

[FINK82]
S. Finkelstein. "Common Expression Analysis in Database Applications," *Proc. of the 1982 ACM-SIGMOD Conference*, Orlando (June 1982).

[FREY87]
J. C. Freytag. "A Rule-Based View of Query Optimization," *Proc. ACM-SIGMOD Conference on Management of Data*, San Francisco, (1987)

[GARE79]
M. Garey and D. S. Johnson. "Computers and Intractability - A Guide to Theory of NP-Completeness," W. H. Freeman and Company (1979).

[GRAE87]
G. Graefe and D. DeWitt. "The Exodus Optimizer Generator," *Proc. ACM-SIGMOD Conference on Management of Data*, San Francisco (1987).

[GRAN80]
J. Grant and J. Minker. "Optimization in Deductive and Conventional Relational Database Systems," pp. 195-234 in *Advances in Database theory 1*, H. Gallaire, J. Minker and J. M. Nicolas, eds.(Plenum Press, 1980)

[NILS80]
N. J., Nilsson. *Principles of Artificial Intelligence* (Tioga Publishing Company, 1980).

[ROSE82]
A. Rosenthal and D. Reiner. "An Architecture For Query Optimization," *Proc. ACM-SIGMOD Conference on Management of Data*, Orlando, FL (1982).

[ROSE86]
A. Rosenthal and P. Helman. "Understanding and Extending Transformation-Based Optimizers," *IEEE Database Engineering 9*, No. 4, pp. 44-51.

[SELI79]
P. Selinger, et al. "Access Path Selection in a Relational Data Base System," *Proc. of the 1979 ACM-SIGMOD International Conference on the Management of Data*, Boston, MA (June 1979).

[SELL86]
T. K. Sellis. "Global Query Optimization," *Proc. of ACM SIGMOD*, Washington, D. C. (May 1986).