Scheduling and Caching in Multi-Query Optimization

A. A. Diwan S. Sudarshan Dilys Thomas *

Indian Insitute of Technology Bombay Stanford University

Abstract

Database systems frequently have to execute a batch of related queries. Multi-query optimization exploits evaluation plans that share common results. Current approaches to multi-query optimization assume there is infinite disk space, and very limited memory space. Pipelining was the only option considered for avoiding expensive disk writes. The availability of fairly large and inexpensive main memory motivates the need to make best use of available main memory for caching shared results, and scheduling queries in a manner that facilitates caching. Pipelining needs to be exploited at the same time.

We look at the problem of multi-query optimization taking into account query scheduling, caching and pipelining. We first prove that MQO with either just query scheduling or just caching is NP-complete. We then provide the first known algorithms for the most general MQO problem with scheduling, caching and pipelining.

1 Introduction

Database systems are facing an ever increasing demand for high performance. They are often required to execute a batch of queries, which may contain several common subexpressions. Traditionally, query optimizers like and optimize queries one at a time and do not identify any commonalities in queries, resulting in repeated computations [6]. As observed in [7, 5] exploiting common results can lead to significant performance gains. This is known as *multi-query optimization* (MQO). There has been a significant amount of recent work on MQO. [6] demonstrates the practical applicability of MQO based on efficient algorithms for implementing a greedy heuristic.

The need for MQO has been expressed in several contexts in the recent past including mediators, view maintenance, XML query optimization and continuous query optimization. See [9] for a detailed study.

Multi-query optimization exploits the possibility of reusing (sharing) results of common subexpressions. Previous work assumed that unbounded space is available to store results of common subexpressions. Typically shared results are stored on disk, and disk space is plentiful; however, there are situations where not all shared intermediate results fit on disk. More importantly, from a practical viewpoint, the growing size of main-memory makes it possible to cache many shared results in memory, avoiding the high cost of reading from or writing to disk. The order of executing the queries and evaluating the subexpressions affects the amount of space needed to keep the intermediate expressions.

We first deal with the problem of finding the best order of evaluation of expressions, which we call the *scheduling problem*, and the problem of deciding when to admit (store) a shared result in cache, and when to discard it, which we call the *caching problem*, to minimize evaluation cost under cache space constraints.[1]

The following example motivates scheduling and caching in MQO.

Example 1 *Suppose the set of queries $Q_1, Q_2, ..Q_n$ needs to be optimized and let the only common sub-expressions be those between Q_i and $Q_{n/2+i}$, say S_i, and assume the size of all S_i's is equal to S. Multi-query optimization disregarding cache space constraints would decide to cache results of all the common subexpressions, instead of recomputing each twice (assuming the cost of storing and retrieving from cache is lower than the cost of recomputation). The cache requirement will thus be $nS/2$. But a cache of size $2S$ is sufficient if the queries are evaluated in the order Q_1, $Q_{n/2+1}$, Q_2, $Q_{n/2+2}$, ..., $Q_{n/2}$, Q_n and each subexpression S_i is kept in cache only between the evaluations of Q_i and $Q_{n/2+i}$.* □

Gupta et al. [3] studied scheduling and caching in MQO, and presented results on intractability of the caching problem, as well as approximation algorithms for special cases of the caching problem. The problem of caching shared results was also addressed by Tan and Lu [8]; more details are provided in Section 4.

It is also possible to execute multiple subexpressions concurrently, *pipelining* the output of a shared subexpression to multiple uses of the expression. Dalvi et al. [2] showed that not all ways of pipelining results to their uses are realizable with limited buffer space, and outlined a class of pipeline schedules called valid pipeline schedules that can always be realized without any buffer space. They also showed that the problem of finding the best pipeline schedule is intractable, and provided greedy heuristics for finding pipeline schedules. However, [2] does not consider scheduling and caching.

©CONTACT Email Addresses: dilys@cs.stanford.edu, sudarsha@cse.iitb.ac.in, aad@cse.iitb.ac.in

International Conference on Management of Data
COMAD 2006, Delhi, India, December 14–16, 2006

[1] There is a dual problem of minimizing the cache space, given bounds on execution time; we do not address that problem here, although the complexity would be the same, and our algorithms can be extended to handle this case.

To get the best plan for evaluating a batch of queries, the query optimizer has to take into account scheduling, caching and pipelining. No earlier work, to our knowledge, has addressed this general version of the MQO problem.

2 Intractability Results

In this section we address the intractability of scheduling, caching and pipelining decisions, given a plan with common subexpressions (the plan could be for a batch of queries, or for a single query with common subexpressions internally). We consider each of the above aspects individually, and show that intractability is intrinsic to each aspect, even with greatly simplifying assumptions.

Ibaraki and Kameda [4] proved the NP-completeness of finding an optimal join order, while Chatterji et al. [1] showed that getting a polylogarithmic approximation to this problem is also NP-complete. These results prove the intractability of finding an optimal query plan for a single query, ignoring the issue of common subexpressions. Our work complements these results.

2.1 Intractability of Scheduling

The problem of finding an ordering of the queries is an important part of scheduling in multi-query optimization. In this section, we show that this problem is strongly NP-complete.

Each expression has a size and there is some benefit associated with caching it, while evaluating some bigger expression. We prove intractability even with the following simplifying assumptions: all queries are the join of just two relations, and the relations are of unit size, and with unit cost of evaluation (i.e. reading from disk). The subexpressions are thus simply database relations, and the benefit of caching them in memory is unit.

Formally, the simplified version of our problem is as follows. Let $r_1, r_2, \ldots, r_m$ be a set of database relations and let $q_1, q_2, \ldots, q_n$ be a set of queries such that each q_i is a join of two database relations. The size of each r_i is 1 and the total cache size available is 2. The cost of reading any r_i from disk is 1, as is the cost of evaluating a query. While evaluating each query the constraint is that both the base relations must be in cache. Find a permutation of these n queries such that the the computation cost of the entire set is minimum.

Theorem 1 *The problem of finding the best order among a set of 2 relation join queries where all relations are of size 1, with cache size 2, is strongly NP-Complete.*

PROOF: Without loss of generality we can assume that $(i \neq j) \Rightarrow ((a_i, b_i) \neq (a_j, b_j))$. Now, no 2 of the queries in the query set are identical and hence 2 adjacent queries in the permutation can have at most 1 relation in common and save a cost of 1. We will prove that the problem of finding an optimal permutation is NP-complete by reducing the problem of finding a Hamiltonian path in a cubic graph [2] which is known to be NP-complete, to it.

We show this by first considering the decision problem: Does there exist an optimal permutation of queries such that the cost saved is $n-1$, i.e. every two adjacent queries in the permutation have a common relation. We first show that this problem is equivalent to finding a dominating trail of a graph and this in turn is equivalent to finding a Hamiltonian path in a cubic subgraph.

For the given set of queries, define the *underlying graph* as a graph whose nodes are the various base relations that occur in the queries, and two nodes r_i and r_j are connected if there is a query in the query set computing the join of r_i and r_j. A *dominating trail* of a graph is a trail (a walk through the vertices so that no edges are repeated; note however, vertices could be revisited) so that all edges of the graph are incident on one of the vertices of the trail. Now suppose there is a permutation such that the cost saved is $n - 1$. This means for every pair of adjacent queries one relation remains in cache and is common to both queries. That is, if the order of evaluation is $q_{i1}, q_{i2}, ..., q_{in}$ then some r_{j1} is common to all queries from $q_{i_{s0}}$ to $q_{i_{s1}}$, r_{j2} is common to all queries from $q_{i_{s1}}$ to $q_{i_{s2}}$, r_{j3} is common to all queries from $q_{i_{s2}}$ to $q_{i_{s3}}$, and so on till r_{jm} is common to all queries from $q_{i_{s(m-1)}}$ to $q_{i_{sm}}$ where $1 = s0 < s1 < s2..... < sm = n$. Now this corresponds to the dominating trail $r_{j1}, r_{j2}, ...r_{jm}$ in the underlying graph. So the problem of finding a dominating trail is equivalent to the problem of finding a permutation with cost saved = $(n - 1)$.

Now to show that finding the dominating trail is NP-complete we show a reduction from Hamiltonian path.

Consider any cubic graph G. Now obtain graph G' from G by adding a edge to each vertex of G as shown in Figure 1.

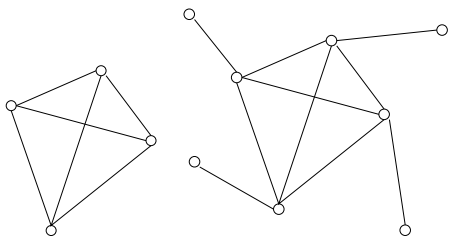

Figure 1: Initial graph G and graph G' with pendant edges added

Now the problem of finding a Hamiltonian path in G is the same as finding a dominating trail in G'. A dominating trail in the graph G' must dominate all edges in the graph, including all the pendant edges, so the trail must go through all the vertices in the original graph G. The degree of each vertex of G in G' is 4, and one of the edges is a pendant edge, which can come only at the beginning or at the end of a trail. Hence the trail can be reduced to a path through all vertices in the original graph G by removing pendant vertices at each end of the trail if present, which is a Hamiltonian path. Thus G has a Hamiltonian path iff G' has a dominating trail.

Thus the problem of finding a Hamiltonian path on a cubic graph, which is strongly NP-complete, can be reduced to a special case of the problem of scheduling in MQO, completing the proof. □

The above result proves that the scheduling part of the query scheduling and caching problem, by itself, is NP-complete. We had made the trivializing assumptions that

[2] A cubic graph is a graph which has all vertices of degree 3.

size of each relation is 1 and the benefit (cost saved) of saving an expression in cache is 1. We also trivialized the general query structure by considering only joins of two relations as queries. In spite of all these assumptions the problem still turns out to be NP-complete.

Combining the NP hardness of pipelining from [2], caching from [3] and scheduling from this Section we get the following result.

Theorem 2 *Scheduling, caching and pipelining for a fixed query plan are all independently intractable.* □

3 Generalized MQO Algorithms

In this section we provide exponential algorithms for a generalized version of the multi-query optimization problem, taking into account scheduling, caching and pipelining. Our exposition is based on the Volcano representation of query plans, outlined in [6]. Join order optimization as in System R is a special case, and for this case we give more precise bounds on time.

Given the input query, the Volcano optimization algorithm first generates all possible semantic rewritings of the input query. For example, $(A \bowtie (B \bowtie C))$ can be rewritten using *join commutativity* as $(A \bowtie (C \bowtie B))$. The Logical Query DAG (LQDAG) is an AND-OR DAG representing the space of all possible equivalent relational algebra expressions. The Physical Query DAG (PQDAG) is used to completely specify the various algorithms available to evaluate a relational algebra expression (hash-join, merge-join etc) and also the various physical properties that are satisfied.

A specific plan is a sub-graph of the PQDAG; plans without sharing would be trees, whereas plans that share subexpressions would themselves be DAGs; we use the term *planDAG* to refer to a specific plan. Note that in case of multi-query optimization we look upon the batch of queries as a single Query DAG with a pseudo root having as children the roots of the individual queries. Note also that equivalent nodes (i.e., those representing the same expressions) are replaced by a single node that may be shared by more than one query.

In the rest of the paper, for ease of exposition we frame our descriptions in terms of the LQDAG; in reality our algorithms would be applied on PQDAGs, and would work correctly, in exactly the same manner.

3.1 MQO Algorithm with Scheduling, Caching and Pipelining

We provide an exponential algorithm for multi-query optimization in full generality (with scheduling, caching and pipelining). For a query of size n, we know the number of distinct planDAGs (each of whose size is bounded by polynomial in n) is exponential in n. Pipelining of arbitrary subset of edges of a given planDAG may result in an infeasible pipelining plan (i.e. one that may take more than a constant amount of buffer space). However, feasibility can be tested in polynomial time by searching for C-cycles [2].

Finding the optimal scheduling and caching strategy for a fixed set of pipelined edges in a fixed planDAG is reduced to a minimum weight path problem as described below. By applying this on all feasible pipelining plans, we can get an optimal overall plan, taking pipelining, scheduling and caching into account.

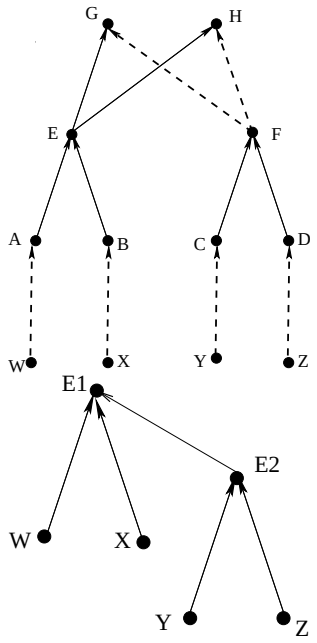

Figure 2: Pipelining in MQO: The compressed graph.

Figure 2 shows a fixed set of pipelined edges in a fixed planDAG. Dotted edges are materialized, whereas solid edges are pipelined.

We can partition the vertices in the DAG into equivalence classes, such that any two vertices are in the same equivalence class iff they are connected by a path of pipelined edges (in the underlying undirected graph). In the above example, A,B,E,G,H form one equivalence class say E1 while C,D,F forms the other say E2. The other vertices are in singleton partitions. Figure 2 also shows a compressed graph where equivalence classes have been replaced by single vertices.

All base relations are materialized and extra scan (read) nodes are created for shared read optimization [2], i.e. for pipelining a subset of the edges coming out of a expression, while materializing the expression and reading from cache for other uses.

As pipelining forces many operators to be executed concurrently, an entire equivalence partition of vertices have to be evaluated concurrently. The scheduling problem is reduced to a shortest path problem on a graph. The nodes of this graph correspond to evaluation states. At any intermediate step in this evaluation, a state is represented by which joins (equivalence classes) have been evaluated and which join results (equivalence classes) are available in cache. Thus the states are represented by (*Done*,*Cached*) where *Done* is a subset of equivalence partitions described above and *Cached* is a subset of nodes in the original uncompressed PlanDag, which are in the subDAGs rooted at the *Done* nodes. Only states where the total size of join results in cache is less than the cache size C are considered, the rest are discarded.

The edge transitions in the above graph, called schedule

graph, are of the following types:

1. *Removal of node:*
   (*Done,Cached* $\cup\{c\}$) $\rightarrow$ (*Done,Cached*) having cost of removal of expression from cache which may be zero. An example of such a transition for the DAG in Figure 2 is $(E1, \{A, B, G, H\}) \rightarrow (E1, \{B, G, H\})$.

2. *Evaluation of a Component:*
   (*Done, Cached*) $\rightarrow$ (*Done* $\cup$ {*EquivComp*}, *Cached* $\cup\{e_1, e_2, .., e_m\}$) where *Cached* includes all nodes having edges going into the equivalence component *EquivComp* and $\{e_1, e_2, ..., e_m\}$ is a subset of nodes in *EquivComp*. The cost of this edge is the cost of pipelining and evaluation of all the intermediate vertices of the *EquivComp* and the cost of writing out e_1, e_2,... , e_m to the cache. Examples of such a transitions for the DAG in Figure 2 include $(\phi, \{A, B, F\}) \rightarrow (E1, \{A, B, E, G, H, F\})$ and $(\phi, \{A, B, F\}) \rightarrow (E1, \{A, B, G, H, F\})$.

3. If cache is on disk then base relations can always be assumed to be cached. If cache is only in memory, edges are present for reading base relations into cache. If both disk and memory can be used as cache, then states must be represented as (*Cached, InMemoryCache, InDiskCache*) as explained previously.

The minimum cost evaluation plan is the shortest path from (ϕ, ϕ) to $(S, *)$ where S contains all plan root nodes; $*$ is a wildcard since the cache contents at the end are irrelevant. Here is an example of a complete schedule corresponding to the pipelining decision shown in Figure 2. Neglecting W,X,Y,Z and assuming A,B,C,D have to be read in we get (to get a smaller example):
$(\phi, \phi) \rightarrow (\phi, \{C\}) \rightarrow (\phi, \{C, D\}) \rightarrow (\{E2\}, \{C, D, F\})$
$\rightarrow (\{E2\}, \{C, F\}) \rightarrow (\{E2\}, \{F\}) \rightarrow (\{E2\}, \{F, A\}) \rightarrow$
$(\{E2\}, \{F, A, B\}) \rightarrow (\{E1, E2\}, \{A, B, F, G, H\})$
$\rightarrow (\{E1, E2\}, \{A, B, G, H\}) \rightarrow (\{E1, E2\}, \{A, G, H\})$
$\rightarrow (\{E1, E2\}, \{G, H\})$

An exponential algorithm for MQO taking into account pipelining, scheduling and caching, is presented in Algorithm 1.

Theorem 3 *Multi-query optimization with scheduling, caching and pipelining has an exponential algorithm.* □

If we restrict to join orders only, with l join implementations, then the cost is bounded by the product of number of plan alternatives, $(4l)^n * n!$, times n^n which is the maximum number of choices in pipelining edges,[3] times the cost of Dijkstra's algorithm which is bounded by $(3^n)^2$, times the cost of testing feasibility and evaluating cost, which is bounded by n^2. The total cost is hence bounded by $(36ln)^n * (n + 2)!$.

4 Related Work

For space reasons, please see extended report [9] for a discussion of related work.

[3]Without shared read optimization we only have to select a subset of edges and hence would get 2^n, in place of n^n. But with shared read optimization [2], which is essential for finding optimal plans, any subset of edges from a node could be combined. A better bound can be got by using Bell numbers B(n) or by Sterling numbers of the second kind as explained in [9].

Algorithm 1 Procedure BestPipelinedScheduledMQOPlan(QueryExpression E)

```
Create the LQDAG for E
bestplan=Null
bestcost=infinity
for all QueryPlan Q in the LQDAG do
  for all Subset E of pipelineable edges in Q do
    Test if it is feasible to pipeline all edges in E
    if feasible then
      Develop the compressed graph by fusing the
      equivalence vertices into one vertex
      Develop the Schedule Graph G for this compressed graph
      cost= cost of shortest path from (φ, {Base-relations}) to (HasRootNode, *) where HasRootNode is any state containing the root node.
      /* Obtained by running Dijkstra's shortest path Algorithm on Schedule Graph G.*/
      if cost < bestcost then
        bestcost=cost
        bestPlan=(QueryPlan Q, Pipelined edges E, Schedule given by shortest path)
      end if
    end if
  end for
end for
```

5 Conclusions

Although the techniques presented here are expensive, they provide the first complete solution to the multiquery optimization problem, taking pipelining, caching and scheduling into account. Further, these techniques form the basis for efficient heuristics, which are described in detail in [9].

References

[1] S. Chatterji, S. S. K. Evani, S. Ganguly, and M. D. Yemmanuru. On the complexity of approximate query optimization. *ACM Symp. Principles of Database Systems*, 2002.

[2] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multiquery optimization. *ACM Symp. Principles of Database Systems*, 2001.

[3] A. Gupta, S. Sudarshan, and S. Viswanathan. Query scheduling in multiquery optimization. In *IDEAS*, pages 11–19, 2001.

[4] T. Ibaraki and T. Kameda. on the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, Sept. 1984.

[5] A. Rosenthal and U. S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Intl. Conf. Very Large Databases*, pages 230–239, 1988.

[6] P. Roy. *MultiQuery Optimization and Applications.* PhD thesis, IIT Bombay, 2000.

[7] T. K. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, Mar. 1988.

[8] K.-L. Tan and H. Lu. Workload scheduling for multiple query processing. *Information Processing Letters*, 55(5):251–257, 1995.

[9] D. Thomas. Scheduling in multiquery optimization. BTech Report, IIT Bombay, 2002. http://www.cse.iitb.ac.in/dbms/Pubs/mqo-dilys.pdf.