Scheduling and Caching in Multi-Query Optimization

A. A. Diwan S. Sudarshan D. 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. After showing the connections of this problem with other traditional scheduling problems and graph theoretic problems we outline heuristics for MQO 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 [6] and [15] optimize queries one at a time and do not identify any commonalities in queries, resulting in repeated computations. As observed in [16, 12] 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. [13] 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 [20], view maintenance [11], XML query optimization [19] and continuous query optimization [3].

*CONTACT Address: GATES 482, Stanford University, CA-94305 USA. Phone: 1-650-723-4532 Fax: 1-650-725-4671 Email:dilys@cs.stanford.edu

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

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. [8] 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 [21]; more details are provided in Section 4.

[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.

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. [4] 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, [4] does not consider scheduling and caching.

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.

Our contributions: The following are the main contributions of this paper:

- We show that the scheduling problem, even disregarding the caching problem, is NP-complete. This complements the result of [8] that the caching problem, disregarding the scheduling problem is NP-complete.
- We provide (to our knowledge, the first) algorithms for the general case of the MQO problem, taking into account scheduling, caching and pipelining (Section 3). While the algorithms are expensive (of exponential complexity), they could form the basis for efficient heuristics. The algorithms can be applied to System R style join-order optimization as well as to Volcano style top-down optimization using a general-purpose DAG representation of queries.
- We provide efficient heuristics for scheduling in MQO, and suggest their extensions for the problem with scheduling, caching and pipelining.

The rest of the paper is organized as follows. Section 2 provides intractability results in scheduling and caching. Section 3 provides algorithms for the generalized multi-query optimization problem. Section 4 shows connections of this problem with other related problems in graph theory and in scheduling. This section also relates our work to other work in multi-query optimization. We provide heuristics in Section 5 and finally Section 6 concludes the paper.

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 [9] proved the NP-completeness of finding an optimal join order, while Chatterji et al. [2] 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

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

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.

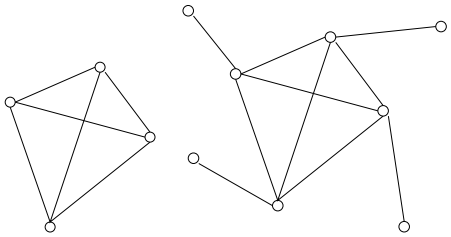

Figure 1: Initial graph G and graph G' with pendant edges added

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 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.

2.2 Intractability of Caching and Pipelining

The following theorem from Gupta et al. [8] addresses the other side of the issue, that is the intractability of the caching part of the scheduling and caching problem.

Theorem 2 *[8] The problem of deciding what expressions to cache and what to discard at each point in a given schedule in order to minimize overall cost, is NP complete.*

Given an order of queries, Gupta et al. [8] create a set of intervals as follows: for each shared expression e create an interval E with parameters of the interval E is as follows

- w_e = size of the result of subexpression e
- $benefit_e$ = benefit by caching e
- starting time and the ending time of E is defined by the sequence number of the first and the last queries using e in the given order, respectively.

The *interval fitting* problem is then to define starting and ending points for each interval such that at any point in time, the results corresponding to intervals crossing that point all fit in cache.

Gupta et al. [8] show that the interval fitting problem is intractable by the following simple reduction from the subset sum problem, i.e., given a set of objects S, find if there is a subset $s \subseteq S$ such that the sum of the sizes of objects in s equals a given value T. For each object p in S create an interval i having

- starting time $sp_i = 1$
- ending time $ep_i = 2$
- size of interval w_i = size of object p
- benefit of interval b_i = size of object p, and
- cache size $C = T$

The above result shows that the weights and benefit part of the query scheduling problem is in itself NP-complete, even without ordering of queries or adding or removal of expressions from cache as time progresses, and assuming that the benefit of a shared result is a given fixed value. The real life situation is made more complicated by the fact that the benefit of a shared result is not fixed – it depends on the plan chosen, and is thereby dependent on what other results are chosen to be shared.

The problem of finding an optimal set of edges to pipeline in a given plan, without scheduling or cache size limitations, has also been shown to be NP hard [4]. We thus summarize this section with the following result.

Theorem 3 *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] and [13]. 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 Bounds on Equivalent Query Expressions

The cost of optimization depends on the number of expressions equivalent to a given expression. For join order transformation, the number of equivalent expressions is exponential in the size of the initial query.[3] For the general case, with an arbitrary set operators, we can show that the number of equivalent rewritings of a query expression is exponential in the size of original query expression, under some reasonable assumptions.

A given expression can have infinitely many equivalent expressions generated by transformation rules. For example, given the expression $(\sigma_{A.x<10}A) \bowtie_{A.x=B.x} B$, transformation rules can in general generate an arbitrarily large expression of the form:

$(\sigma_{A.x<10}(\sigma_{A.x<10}(\ldots(\sigma_{A.x<10}A)\ldots)) \bowtie_{A.x=B.x} B.$

To avoid this problem we assume that given a query of size n, transformation rules will only generate expressions bounded in size by a polynomial $p(n)$.

To illustrate another problem, consider the expression $(\sigma_{A.x<10}A) \bowtie_{A.x=B.x} B$, which can be equivalently rewritten as $(\sigma_{A.x<10}(\sigma_{A.x<100}A)) \bowtie_{A.x=B.x} B$. An infinite number of such expressions with different constants can potentially be generated. To avoid the above problem, we make the following assumption: *A query transformation rule can introduce at most a constant number of new tokens (such as operators or constant values)*, across all expressions that it is applied on. This ensures that only a limited number of expressions of the form listed above can be generated.

We also make the assumption that each application of a transformation rule takes time polynomial in the size of the expression and the transformation rule.

The above assumptions are quite reasonable, and satisfied by all commonly used transformation rules. Under the above assumptions, it is easy to show the following theorem:

Theorem 4 *The time needed to generate all the equivalent expressions of a query expression is bounded by a function that grows linearly in the number of transformation rules and exponentially in the size of the original query expression.* □

3.2 MQO without Pipelining, Scheduling and Caching

The first step of the query optimizer is to expand the query DAG. If the query has size n, the number of nodes, m, in the expanded DAG is exponential in n, i.e. $m \sim exp(n)$. A subset of these nodes will be decided to be materialized. The number of subsets, s, which is 2^m is thus doubly exponential [4] in n.
However we can iterate over all possible plans (a single plan is polynomially bounded in size) and for each plan iterate over all possible subsets of nodes in that plan to be materialized, to find the combination with lowest cost. We thus get a simple (naive) exponential algorithm for MQO without considering optimizations like scheduling and pipelining.

Theorem 5 *MQO (without pipelining, scheduling and caching) has a worst-case exponential algorithm.* □

We now consider the special case of join order optimization. Consider a single query, which is a join of n relations. Let the relations in the join be $R_1, R_2 \ldots R_n$. Then the number of different join trees is the product of the number of permutations of these n relations and the number of ways of inter-nesting n parenthesis (in a valid parenthesization), i.e. *Catalan-number*$(n-1)$ and is thus bounded by $4^n \times n!$. With l join implementations the number of plans is bounded by $(4l)^n \times n!$.

Given a batch of queries having a total of n relations, each has an exponential number of plans, and an overall plan is obtained by choosing one plan for each query. It is easy to show that the bound on the number of plans for a single expression with n relations is an upper bound on the number of overall plans for the batch of queries.

[3] A function $f(n)$ is said to grow exponentially in the size of n if $log(f(n))$ is polynomial in n.

[4] A function $f(n)$ is said to be doubly exponential in the size of n if $log(f(n))$ is exponential in n.

The time complexity of MQO on join queries is then bounded by a product of the number of equivalent plans, $(4l)^n * n!$ times the materialization choices for each plan, 2^n (the number of subsets of a set of size n), times the time n taken for cost calculation per plan. This is bounded by $(8l)^n * (n+1)!$.

Theorem 6 *Multi-query optimization (without scheduling, caching and pipelining) on a set of join queries with a total of n relations, with l join techniques can be done in time $O((8l)^n * (n+1)!)$.* □

3.3 MQO with Scheduling and Caching

We provide an exponential algorithm for the problem of MQO with scheduling and caching (with limited cache size). Though we initially restrict ourselves to joins only, we later provide an algorithm for the general case.

We describe below an exponential algorithm for finding the optimal schedule for a given execution plan with only joins. Using the algorithm on all possible execution plans, and selecting the best overall plan, gives an exponential algorithm for MQO with scheduling and caching.

Problem formulation: Given a cache size C, and a single join planDAG, where each join has a cost and size, find the minimum cost (optimal) schedule of evaluating the planDAG. A cost for storing in cache is assumed.

This can be reformulated as a shortest path problem on a graph constructed as follows. Nodes of the graph correspond to evaluation "states". At any intermediate step of the evaluation, the state can be represented by which joins have been evaluated, and which join results are available in cache. Thus a state is a tuple (*Evaluated joins, Joins in Cache*). There are at most 3^n such possible states since join results in cache are a subset of joins evaluated. Only states where the total size of the join results in cache is less than the cache size C are considered, others are discarded. Given a state, we can draw edges representing transitions to other states. The transitions are of the following kinds:

1. Remove a join result from the cache.
2. If the inputs of a join are in the cache, evaluate it and put the result in cache.

Now finding the minimum cost evaluation plan can be achieved by just finding shortest path from (ϕ, ϕ) to (S,*) where S contains all plan root nodes; * is a wildcard, since cache contents at the end are irrelevant. This takes time at most O(9^n) using Dijkstra's shortest path algorithm.[5]

An enumeration algorithm that tries all possible evaluation plans combinations gives the overall best plan. Its cost is the product of the cost of the above algorithm and the number of plans, and is bounded by $(36l)^n \times n!$

[5]It is interesting to note that the above algorithm is a generalization of dynamic programming in the following sense. Standard dynamic programming algorithms can be viewed as finding shortest paths in DAGs: each memoed result corresponds to a node, and there is an edge from each memoed result to each other result in whose computation it is used. In our context, the problem is more general since it requires finding shortest paths in arbitrary graphs that may have cycles.

Theorem 7 *MQO with scheduling and caching, for join queries, has an exponential algorithm of complexity $O((36l)^n * n!)$ where there are a total of n relations and l join techniques.* □

We now provide a description of the generalized version of the algorithm which holds for arbitrary DAGs and operations other than joins.

A query is expanded into its LQDAG (Logical Query DAG) and PQDAG (Physical Query DAG). The number of different plans of a query of size n is bounded by a function $exp(n)$, that grows exponentially in n. Consider a single execution plan. The size of this DAG can be assumed to be polynomial in n. As mentioned earlier, we assume the LQDAG has a pseudoroot whose children are the roots of individual queries.

As explained for the case of joins, at any time the state of execution of the query can be given by the tuple (*Done*, *Cached*) where *Done* is a set of vertices whose sub-DAGs have been evaluated. And *Cached* is a set of vertices that are presently stored in cache. *Cached* is a subset of vertices present in the sub-DAGs rooted at the *Done* nodes.

An exponential algorithm for MQO with scheduling and caching is presented in Algorithm 1.

Theorem 8 *There is an exponential algorithm for MQO with scheduling and caching.* □

The algorithm can be easily extended if both memory and disk are used for caching. In this case the state must be extended to be of the form (*Done*, *InMemoryCache*, *InDiskCache*). The edges (with appropriate costs) that have to added must now also include

1. Moving a expression from memory cache onto disk cache or the other way round.
2. If the predecessors of a node are all in the memory cache or disk cache as required by the implementing algorithm evaluate it and put the result in memory or disk cache.
3. Discarding an expression from memory or disk cache.

In case scheduling were not required we could assume at any instant the set of intermediate vertices that have been computed is always a set of sub-DAGs rooted at a single vertex. In other words, we could assume that evaluation of nodes take place such that a single subDAG of evaluated nodes steadily expands to the complete DAG. However for an optimal schedule the intermediate vertices computed is a general set of rooted DAGs which may be rooted at different points. That is, the evaluation of different parts of the subDAG which are not directly connected to each other may be interleaved.

By swapping portions of unrelated computations of different queries and finally assuming that queries are evaluated in the order they were completed it may appear that it is not necessary to interleave the evaluation of subexpressions from different queries of a batch. However, it may

be necessary to compute expressions which are not used by a query, before the query is completely evaluated, in order to get an optimal schedule. Consider for example a query batch of Q1 and Q2, Q1 has $Count_{GroupBy\{A,B\}}$ as a subexpression while Q2 has $Count_{GroupBy\{A\}}$ as a subexpression. It may be the case that in an optimal schedule, $Count_{GroupBy\{A,B\}}$ is evaluated first and given to its parent operator during the computation of Q1 and concurrently $Count_{GroupBy\{A\}}$ (smaller size) is calculated from $Count_{GroupBy\{A,B\}}$, before discarding $Count_{GroupBy\{A,B\}}$ from cache and then continuing with the computation of Q1, before computing Q2.

Another example of the interleaving of computation of expressions from different queries is in the case of pipelining. Pipelining the same expression to two operators in different queries needs the evaluation of both queries to be interleaved.

3.4 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 [4].

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 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 [4], 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 as before, but with states as (*Done*,*Cached*) where *Done* is a subset of equivalence

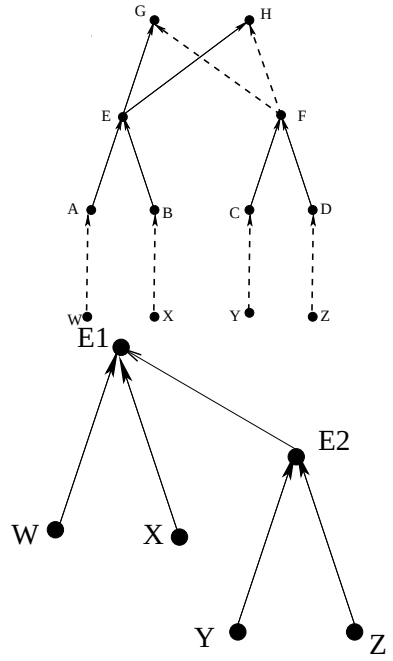

Figure 2: Pipelining in MQO: The compressed graph.

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.

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.

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 2.

Theorem 9 *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,[6] 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 $(36ln)^n * (n + 2)!$.

The caching-scheduling-pipelining plan obtained from the Algorithm 2 gives the optimal plan for:

1. *Static pipelining decision:* Dynamic pipelining decisions are not considered. Pipelining decisions of edges are dynamic if they are made at run time when memory buffer overflows occur, writing the overflow data to disk. Dynamic materialization can lead to expressions being partly pipelined and partly materialized.

2. *A given set of operators:* New operators could be added to the system to consider new algorithms, for example a newly invented method of an optimized bit-mapped join. The plan returned is however the optimal for the given set of operators.

   An interesting optimization for extending the operators is as follows. The hash join algorithm could be split into two operations, one which partitions the data and one which performs the actual join. This allows the partitions to become units that can be cached, and can be reused later. Such a plan could be cheaper than plans that consider the entire hash join as an atomic operation. The runs created during sort-merge-join can be viewed in similar way.

4 Related Work

We first show connections of the problem of scheduling and caching in MQO to other traditional scheduling problems and graph theoretic problems.

The well known register allocation problem [17], is to find the optimal order of evaluating an expression tree, where all intermediate expressions are of unit size, so as to minimize the number of intermediate locations to save data. [18] gives a dynamic programming algorithm for register allocation for expression trees where nodes are unweighted (i.e. each result can be held in one register) but operations have implementations of different costs. [1] provides an algorithm for evaluation of expression trees in the context of expression evaluation and environment passing in functional languages, which reduces to evaluating trees with weighted nodes. None of these results can be extended for DAGs.

The problem of minimizing cache requirement given the query in the form of a DAG even when the nodes are unweighted is proved to be NP Complete [17]. The treewidth, pathwidth and vertex separation numbers [5] of a graph are closely related to the scheduling problem. These are not only NP-complete in the general case, but even the special results have very large constant factors and are highly impractical. [10] provides an extensive and comprehensive coverage of these graph theoretic problems.

Earlier approaches to multi-query optimization (with the exception of Tan and Lu [21]) assume there is infinite disk space, and very limited memory space. Pipelining was the only option considered for avoiding expensive disk writes. [4] provides algorithms for the problem of pipelining in MQO but does not consider the problems of scheduling and caching. The availability of fairly large and inexpensive main memory motivates the need to make best use of available main memory by for caching shared results, and scheduling queries in a manner that facilitates caching.

Tan and Lu [21] provide heuristics for scheduling in MQO, but they require a pairwise test for deciding a good order of queries, unlike a single MQO invocation as we propose, which adds minimal overhead. Also our technique is more general as it grows many segments at the same time allowing better schedules as explained shortly. Though Gupta et al. [8] provide approximation techniques for caching given a fixed schedule, they only provide heuristics for finding good schedules; moreover, their heuristics require a large number of MQO invocations to decide a good schedule and is seen to add a very large overhead even for very small query batches. Our work complements theirs well, since our heuristics for scheduling (described in Section 5) can be combined with the approximate caching algorithms, to get good plans without the overhead of a large number of MQO invocations.

5 Heuristics

Section 4 motivated the intractability of the problem of scheduling and caching in MQO. We present here heuristics that find good query ordering with negligible overhead.

The problem of scheduling and caching of a batch of queries has two main aspects:

1. Choosing a good order of queries

2. Choosing the results of sub-expression that should be admitted to or discarded from cache, as we execute

[6] 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 [4], 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 [7].

queries.

5.1 One Phase Optimization

In this approach the decisions of ordering the queries and what expressions must be added to or removed from cache are taken in a single phase. We provide details of a one-phase heuristic in [22]. In that algorithm, the schedule is built up by adding one query at a time; the next query to add, and expressions to be cached are decided on the basis of queries already added to the schedule, and all possible cached expressions in the currently existing schedule. The heuristic requires calculation of the benefit for each combination above, and results in many calls MQO, and is thus expensive and impractical.

5.2 Two Phase Optimization

In this approach, scheduling is done first followed by caching decisions. To decide the query ordering the maximum weight Hamiltonian path heuristic is used. Let C be the cache size. Let $\{Q_1, Q_2, ...Q_n\}$ be a set of n queries to be optimized. Benefit(Q_i,Q_j), defined as

cost(Q_i) + cost(Q_j) - MQO_cost(Q_i,Q_j)

denotes the benefit gained by executing both the queries as a batch using with cache space restricted to C. We define the QuerySet Graph G, as a graph with nodes as queries belonging to the set of queries and node Q_i connected to node Q_j with an edge of weight Benefit(Q_i,Q_j). As an heuristic, the problem of scheduling can be considered as the problem of finding the maximum weight Hamiltonian path in the QuerySet Graph.

The Hamiltonian path problem itself is NP-complete, so we use the following greedy heuristic. Let G be a graph with n vertices. A subgraph of G with $n-1$ edges, such that the degree of each vertex is at most 2 is a collection of cycles and a path, all being disjoint. If we ensure there are no cycles then the subgraph is a Hamiltonian path.

Greedy edge selection heuristic: Each vertex has a count associated with it, which is initially zero for all vertices. All vertices whose present count is < 2 are candidate vertices. So initially all vertices are candidate vertices. The edges selected during the process partition the graph into connected components. Select the largest weight edge not selected till now and incident on two candidate vertices not in the same component. Increment the count of both these candidate vertices by 1 and continue selecting edges till a total of $n-1$ edges are selected.

This heuristics gives a Hamiltonian path of high weight. The cost of invoking MQO for every pair of queries can be large especially on large query batches. In order to decrease the number of invocations of MQO in the above procedure, instead of running MQO-benefit for every pair of queries in the large query batch, MQO is ran once on the entire batch of queries (with a pseudo root), assuming infinite cache space.

The edge $Q_i - Q_j$ is then given weight equal to the sum of the cost benefits obtained by sharing the common subexpressions of Q_i and Q_j in the plan obtained for the entire batch. Specifically, if *benefit(expr)* is the benefit (to the entire query batch) of materializing the expression *expr* in the MQO plan, and *numuses(expr)* is the number of distinct queries in the query batch that utilize *expr*, then define $Benefit(Q_i, Q_j)$ as

Σ_{expr} *benefit(expr) /numuses(expr)*

where the summation is over all expressions *expr* that are common to Q_i and Q_j and chosen to be materialized (by MQO without cache space limits). The reason we divide by *numuses* is that if an expression is shared by many queries it will have a high benefit in materializing, but its contribution to two queries being scheduled one after another must depend on the benefit expended to these two queries.

Thus a single MQO invocation on the entire batch gives a query ordering.

To test its effectiveness, we implemented the maximum weight Hamiltonian path heuristic on top of the multi-query optimizer described in [14] and ran it on some sample query batches. We used the batched TPCD workload from [14], which models a system where several TPCD queries are executed as a batch. The workload consists of subsequences of the queries Q_1, Q_3, Q_5, Q_7 and Q_9 from TPCD. These queries have common subexpressions between themselves. The batch query BQ_i contains two different (changes in selection parameters) copies of the the first i queries from the above sequence, as in [4].

We found that on all the batches BQ_i, the maximum weight Hamiltonian path heuristic ordered the queries with common subexpressions together (both within and across query pairs) to always give the best query ordering, which was also obtained by the brute force enumeration of all the exponential possible orderings, as done by [8]. In all the experiments we conducted, the time taken by the heuristic is negligible as compared to the MQO invocation cost, as the exponential search in [8] is obviated.

We now suggest a heuristic to integrate pipelining. The techniques of [4] are used on the combined query batch to select the pipelined edges The planDAG graph with edges selected for pipelining is then partitioned as explained in Section 3. Scheduling and caching techniques can now be applied to these partitions, respecting the partial order of the compressed graph. Our scheduling heuristic which gives a good schedule at minimal overhead can be combined with the approximation techniques for caching in [8] to avoid the large number of MQO invocations, to get a good plan at low overhead.

6 Conclusions

In this paper, we studied the issue of scheduling, caching and pipelining in multi-query optimization. We began by motivating the need for scheduling, caching and pipelining. We provided intractability results for simplified versions of the problem. We then provided algorithms for the general version of the MQO problem, with scheduling, caching and pipelining. Although very expensive, these algorithms can serve as the basis for the developments of efficient heuristics.

We then proposed novel heuristics for query scheduling, and preliminary tests showed that the heuristic provides a good query ordering using a single MQO invocation on the entire batch.

References

[1] S. K. Biswas and S. Kannan. Minimizing space usage in evaluation of expression trees. In *Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science 1026.*, pages 377–390, Dec 1995.

[2] 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.

[3] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. *IEEE Intl. Conf. Data Engineering*, 2002.

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

[5] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Information and Computation*, 113(1):50–79, 1994.

[6] G. Graefe and W. J. McKenna. Extensibility and Search Efficiency in the Volcano Optimizer Generator. In *IEEE Intl. Conf. Data Engineering*, 1993.

[7] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison Wesley, 1995.

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

[9] 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.

[10] T. Kloks. *Treewidth: Computation and Approximations.* Lecture Notes in Computer Science 842., 1994.

[11] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Intl. Conf. Management of Data*, 2001.

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

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

[14] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobhe. Efficient and extensible algorithms for multi-query optimization. In *ACM SIGMOD Intl. Conf. Management of Data*, 2000.

[15] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. Price. Access path selection in relational database management system. In *ACM SIGMOD Intl. Conf. Management of Data*, 1979.

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

[17] R. Sethi. Complete register allocation problems. In *SIAM Journal of Computation*, pages 226–248, Apr. 1975.

[18] R. Sethi. Pebble games for studying storage sharing. *Theoretical Computer Science*, 19:69–84, 1982.

[19] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Intl. Conf. Very Large Databases*, 1999.

[20] S. N. Subramanian and S. Venkataraman. Cost based optimization of decision support queries using transient views. In *ACM SIGMOD Intl. Conf. Management of Data*, Seattle, WA, 1998.

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

[22] D. Thomas. Scheduling in multiquery optimization. BTech Report, IIT Bombay, 2002.

Algorithm 1 Procedure BestScheduledMQO-Plan(QueryExpression E)

Create the LQDAG for E
bestplan=Null
bestcost=infinity
for all QueryPlan Q in the LQDAG **do**
Create the Schedule Graph G for Q as follows

1. The vertices are the states (*Done*,*Cached*). Only states where the total size of nodes in *Cached* is less than the cache size are included.

2. Edges representing transitions as created as follows

   (a) Removing a node c from cache is represented by the transition (*Done*,*Cached*) $\rightarrow$ (*Done*, *Cached*-{c}). The cost of this edge is the cost of clearing the cache which could be zero.

   (b) Evaluation of a new expression corresponds to: If e_1, e_2, e_3,... ,e_m belong to *Cached* and $E = op(e_1, e_2, ..., e_m)$ is another expression then a transition from (*Done*,*Cached*) $\rightarrow$ (*Done* $\cup\{E\}$, *Cached* $\cup\{E\}$) and the cost of the edge is the cost of evaluating E and writing it to Cache . Also (*Done*,*Cached*) $\rightarrow$ (*Done* $\cup\{E\}$, *Cached*) does not have the write-to-cache cost.

cost= cost of shortest path from (ϕ, ϕ) 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=plan corresponding to shortest path
end if
end for

Algorithm 2 Procedure BestPipelinedScheduledMQO-Plan(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 $(\phi, \{$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