Query Scheduling in Multi Query Optimization

Amit Gupta*
P.S.P.L. Pune
amitg@pspl.co.in

S. Sudarshan
I.I.T. Bombay
sudarsha@cse.iitb.ernet.in

Sundar Vishwanathan
I.I.T. Bombay
sundar@cse.iitb.ernet.in

Abstract

Complex queries are becoming commonplace, with the growing use of decision support systems. Decision support queries often have a lot of common sub-expressions within each query, and queries are often run as a batch. Multi query optimization aims at exploiting common sub-expressions, to reduce the evaluation cost of queries, by computing them once and then caching them for future use, both within individual queries and across queries in a batch,

In case cache space is limited, the total size of sub-expressions that are worth caching may exceed available cache space. Prior work in multi query optimization involves choosing a set of common sub-expressions that fit in available cache space, and once computed, retaining their results across the execution of all queries in a batch. Such optimization algorithms do not consider the possibility of dynamically changing the cache contents. This may lead to sub-expressions occupying cache space even if they are not used by subsequent queries.

The available cache space can be best utilized by evaluating the queries in an appropriate order and changing the cache contents as queries are executed. We present several algorithms that consider these factors, in order to reduce the cost of query evaluation.

1 Introduction

Multi-query optimization is an issue of increasing importance today. Decision support queries today are often rather complex, involving many relations or views, and operating on large volumes of data (particularly in data warehouses). Such complex queries are often defined as a sequence of SQL queries, involving mostly the same set of relations. Complex queries also often involve extensive use of views, such views may be referred to multiple times in the same query. In either of the above cases, the resultant queries may contain many sub-expressions that are used multiple times. Recognizing the presence of common sub-expressions and reusing sub-expression results can reduce query cost significantly.

Multi query optimization exploits the fact that result of common sub-expressions can be cached and reused when required, so that re-evaluation cost of common sub-expression is saved. For practical purposes we assume that the cache size is finite. Prior work in multi query optimization either ignore the cache space constraints, or at best, consider the problem of choosing the best set of shared sub-expression that will fit in the available cache space [8, 5]. The assumption is that, once computed, these results will be present in cache across the evaluation of all the queries in a batch. Such an approach may waste cache spaces: for instance, results that are never used again in a batch may remain in the cache, whereas if the query schedule has been taken into account, they could have been discarded and other shared results chosen to replace them.

In this paper we present scheduling algorithms that efficiently utilize a fixed size cache. Given an execution plan for each query, our scheduling algorithms decide (a) the order in which queries should be executed and (b) which shared sub-expressions should be admitted and discarded from cache as queries are executed. The scheduling and cache admission and eviction decisions are made in a cost based manner.

1.1 Motivating Example

In this section we show how an efficient ordering of the queries can reduce the cache space required to store results of all the shared sub-expressions of a query batch. In other words, by properly ordering the queries, the cache can store more shared sub-expressions, which can speed up evaluation of the batch of queries. Let the set of queries to be evaluated be $Q_1, Q_2, \ldots Q_n$, and let each pair of queries Q_i and Q_{i+1} share a common sub-expression s_i in their evaluation plans, as shown in Figure 1.

Here, $s_1, s_2 \ldots s_{n-1}$ are the common sub-expressions. For simplicity let us assume that result size of all s_i's is equal to S, and let us assume that all common sub-expressions are worth caching. A multi-query optimizer

*Work done while at I.I.T. Bombay

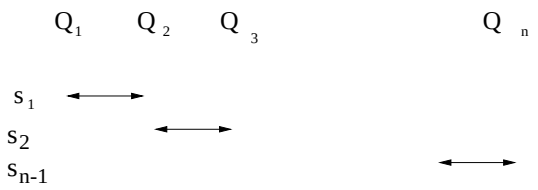

Figure 1: Q_i and Q_{i+1} shares s_i

would decide to cache all shared sub-expression, and to get the full benefit, a cache space of $(n-1)S$ would be required. If available cache space is less than this, several shared results would not be chosen for caching, and would be recomputed, resulting in increased cost. But in case queries are executed in the order $(Q_1, Q_2 \ldots)$ as shown in Figure 1, then only two of the shared subexpression needs to be cached at any point in the execution. Thus, a cache size of $2S$ is sufficient for answering the queries efficiently. In other words if the available cache space is just $2S$, proper scheduling can result in the same execution cost as a much larger cache, whereas current multi-query optimization techniques would decide to cache only two sub-expressions, and give a suboptimal result.

As a more complicated example, consider the case where a shared sub-expression s_1 is shared by Q_1, Q_2 and Q_4 and queries are evaluated in the same order $(Q_1, Q_2, ...Q_n)$ and cache size is $2S$. In this case one can either discard s_1 after Q_2 is executed, to make space for s_3, or discard s_2 after Q_2, and retain s_1 until after Q_4 is executed. This decision will have to be made in a cost-based fashion.

Multi query optimization taking into account query scheduling involves several decisions. The first two decisions are part of the job of a any multi query optimizer, but the next two decisions complicate the task.

1. Choosing what plan to use for each query; the choice for different queries is inter-related since a choice of a particular plan for one query can result in a subexpression that reduces the cost of some plan for another query.

2. Choosing what common sub-expression results are worth caching, given the cost of writing results to the cache (particularly relevant if the cache is on disk).

3. Deciding an efficient order of evaluation of queries.[1]

4. Choosing which common sub-expressions results should be admitted to or discarded from the cache, as each query is executed.

We present our approach to the overall problem in Section 2, concentrating on the last two decisions; we call this aspect of the problem as **query scheduling**.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 describes our basic approach to the problem of query scheduling. Section 3 lists a few assumptions to simplify the problem of query scheduling. The simplified problem and its approximation algorithm is discussed in Section 4. In Section 5 we describe two greedy algorithm for ordering the queries and later in the Section we combine the basic idea of these algorithms to get an efficient ordering algorithm. Related work and conclusion is discussed in Section 6.

2 Overall Approach to Optimization

As problem of multiquery optimization with query scheduling is complex, we break it in two parts.

1. In the first phase, an execution plan for the given set of queries is computed. The execution plan is a DAG structure that specifies the way queries are evaluated, without specifying an order of evaluation. For example, the possible execution plans for computing the expression $(A \bowtie B \bowtie C)$ are shown in Figure 2.

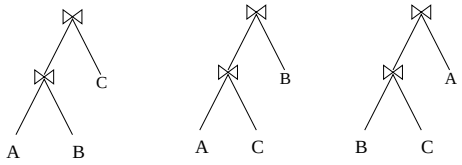

Figure 2: Possible plans of $(A \bowtie B \bowtie C)$

   A batch of queries can have many execution plans, and the multiquery optimizer chooses one such plan based on its cost estimates. The cost estimates can be got, heuristically, assuming infinite cache space.

2. In the second phase, given the execution plan of set of queries, the problem is to decide an optimal order in which the queries should be evaluated, and choose the nodes (sub-expressions) in the execution plan whose results are added to or discarded from the cache, as queries are executed.

   In general, it is possible that a result can be discarded before all its uses, and then recomputed, and perhaps even cached again, for later uses.

[1] It is possible to break up even non-shared parts of a query into smaller parts that are evaluated separately, inter-mingled with the evaluation of parts of other queries, but we ignore this possibility since it increases the cost of optimization while not having any obvious benefit.

The two phases described above are dependent on each other, since the optimizer cannot decide an optimal execution plan of set of queries unless it knows which results of sub-expression will be present in cache when a query is evaluated. In other words, if the optimizer knows the content of cache at the time when a particular query is evaluated, then it can generate an execution plan taking the cache contents into account. But the cache contents are not known till the end of the second phase. On other hand the second phase needs to know the evaluation plan of all queries.

While it is possible to combine both phases to get an overall optimal plan, doing so would be rather complex and the optimization cost is likely to be very high. We therefore keep the phases separate, by ignoring the cache space limitations during the first phase.

3 Simplifying Assumptions

Even after breaking up the problem into two phases, query scheduling is still a complex problem. In this section we list several simplifying assumptions that we make in order to obtain query scheduling algorithms, and later we will discuss how to relax these assumptions.

The assumptions made are:

1. (As mentioned earlier:) Optimization is broken up into two phases, and the execution plans of the queries, generated by the first phase of optimization, are available for query scheduling.

2. The optimizer is given an order in which queries are to be evaluated. (Later we will see how to come up with such an order.)

3. The *benefit* of caching a common sub-expression is independent of the cache content.

   Every common sub-expression has a *benefit* associated with it. The benefit of a sub-expression is defined as the reduction in the overall evaluation cost if the result of that sub-expression is cached, taking into account the cost of storing the result in the cache. (The reduction in cost is because one or more recomputations are replaced by reading of the common sub-expression from the cache.)

   In reality, the benefit of an expression depends on the cache content. For example, if a query uses an expression $Q = A \bowtie B \bowtie C$ and nothing is in cache, then
   Benefit(Q, $\emptyset$) = (computation *cost* of Q - *cost* of reading the result of Q from cache)

   On the other hand, if $(A \bowtie B)$ is already in cache, then *Benefit(Q,$A \bowtie B$)* = (*cost* of computing Q from cached result of $(A \bowtie B$) - *cost* of reading the result of C from cache).

   If cost of computing Q by evaluating $(A \bowtie B \bowtie C)$ is more then *cost* of computing Q by reusing the result of $(A \bowtie B$) then, *Benefit(C, $\emptyset$)* is greater then *Benefit(C, $A \bowtie B$)*.

4. Common sub-expression are cached from their first use to their last use.

   In other words, we do not consider the possibility of discarding a result and recomputing it later. (We relax this assumption later.)

4 Solution to the Simplified Problem

In this section we formulate the problem, considering the assumptions made in Section 3.

Given an ordering of queries, we define an *interval* e which represents a shared sub-expression E; the interval is from its first computation to the time when it is discarded from cache.

Other attributes of e are:

- s_e = size of result of subexpression E
- $benefit_e$ = *benefit* in caching E
- *starting time* of e = sequence number of the first query using E in the given order (Section 3 assumes that query order is given)
- *ending time* of e = sequence number of the last query using E in the given order

The simplified query scheduling problem can be formulated as follows:

> Given a set of intervals S, where each interval i has a starting point sp_i and ending point ep_i, a benefit b_i and a size s_i, and given that the size of the cache is C, find a set $P \subseteq S$, of intervals such that for every point in time t $\Sigma s_p \leq C$, where $p \in P$ and p exists at time t and $\Sigma_{p \in P} b_p$ is maximum. We will call this problem as the *interval packing* problem.

Theorem 4.1 *The decision version of the interval packing problem, which checks for the existence of a set with $\Sigma_{p \in P} b_p = T$, is NP-hard.*

Proof: We show NP hardness by reducing *subset sum*, which is a standard NP-complete problem, to the interval packing problem.

The *subset sum* problem is defined as, given a set S, of objects, where each object in S has a size associated with it, and given a value T, is there a subset S' of S,

such that sum of size of all the objects in S' is equal to T.

The subset sum problem can be reduced to interval packing as follows. For each object p in S create an interval i having

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

The decision version of the interval packing problem tells us if there is a set of intervals P such that $\Sigma benefit_p = T$. The answer to the subset problem is yes if any only if the answer to the interval packing problem is yes. □

4.1 Approximation Algorithm for Subset Sum

We first describe an approximation algorithm for the subset sum problem from [3], and later extend it to get an approximation algorithm for the interval packing problem. As the subset sum is a decision problem, its approximation algorithm does not makes much sense, so we use the optimization version of the subset sum problem, defined as follows.

> Given a set $S = (a, b, c, \ldots)$ and T, find $S' \subset S$ such that, $\Sigma s' \leq T$ and $\Sigma s'$ is maximum, where $s' \in S'$.

Firstly we define an *addition* operation on a set $P = (a, b, c, \ldots)$, where $a, b, ..$ are numbers, as

$$P + x = (a + x, b + x, c + x, \ldots)$$

Let $S = (s_1, s_2, s_3, \ldots, s_n)$ be the given set in the *subset sum* problem, and let T be the required sum. We begin with set $P_0 = (\ 0\ )$, at each step we pick an element from the set S and add it to the set P_i, as shown below:

$$\begin{aligned} P_0 &= (0) \\ P_1 &= (P_0 + s_1) \cup P_0 = (0, s_1) \\ P_2 &= (P_1 + s_2) \cup P_1 = (0, s_1, s_2, s_1 + s_2) \\ &\vdots \\ P_i &= (P_{i-1} + s_i) \cup P_{i-1} \end{aligned}$$

Every element of P_i which is greater than T is thrown out of P_i. P_i represents set of all possible combination of the first i elements of the set S. And each element of P_n represents a possible solution of *subset sum* problem. For example P_2 stores all possible combination of a and b. So in the set P_2, 0 represents a null set, a represents set containing only a, and $a + b$ represents set containing both a and b. The addition step is performed for all the elements in the set S and P_n is computed as the last set representing all the combination of all the objects.

Now the largest element in P_n gives the maximum subset sum. (Correspondingly, if T is present in P_n then the answer to decision version of the *subset sum* problem is yes otherwise the answer is no.)

Each time a new element element is added to the P_i in the worst case it doubles its size. So the length of P_i in worst case is 2^i, thus above algorithm is exponential. However, we can get a polynomial time approximation algorithm by trimming P_i to polynomial space, as described below.

We denote the trimmed version of P_i as L_i. The trimming procedure is applied each time a new element in L_i is added is as follows. At step i when the i^{th} element of S is added to L_{i-1} to get the set L_i. All elements in L_i that are greater than T are deleted first. Then, L_i is trimmed by repeatedly deleting an $z \in L_i$ such that there is an element $y \in L_i$ that satisfies

- $(1 - \delta/n)z \leq y < z$, where $y \in L_i$, δ is a given error constant, and n is the cardinality of S.

The condition implies that y and z in L_i represent subsets of similar sums, and keeping one is sufficient to get an approximate solution.

The complexity of the approximation algorithm for subset sum is $O(n^2)$ and the approximation ratio of the algorithm is $1/(1 - \delta)$ (see [3] for details).

4.2 Approximation Algorithm for Interval Packing

As we did for the subset sum problem we first describe an exponential algorithm for interval packing problem, and then convert it to a polynomial time approximation algorithm by trimming its search space. We define a *schedule* as a set of intervals that can fit in the given cache size, i.e. a set of intervals such that at every point in time the sum of the sizes of the intervals that overlap the point does not exceed the cache size.

4.2.1 Exhaustive algorithm

First we sort all the intervals in non-decreasing order of their ending time. At every step of algorithm we have a set of schedules represented by L_i. We begin with $L_0 = ((\phi))$, which represents that only one schedule is present in L_0 and that schedule does not contains any intervals.

At each step add an interval, in the non-decreasing order of their ending time (the interval having least ending time is added first and so on) to all the schedules present in the set L_i, as follows
$L_{i+1} = (L_i + i^{th}$ interval $) \cup L_i$

Addition of interval to a schedule may produce some schedules which do not fit in cache, such schedules are discarded.

The exhaustive algorithm explores all the possible schedules and chooses the schedule with maximum benefit Σb_i. The benefit Σb_i for a schedule can be computed by adding benefit of all the intervals present in the schedule. The number of schedules present in L_i is exponential.

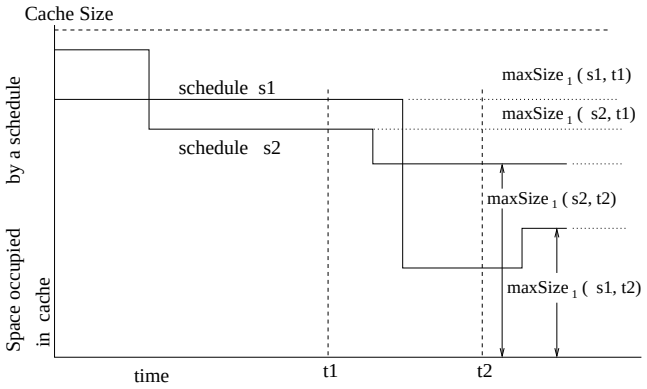

Figure 3: Graph of space occupied by schedule versus time

4.2.2 Polynomial time Approximation Algorithm

Just as in subset sum problem, the approximation algorithm for interval packing problem trims the set of schedules so that the number of the schedules present in the set L_i is polynomial. Again, we assume we are given a constant δ.

For every schedule in the set L_i, we store its benefit; the benefit of a schedule is the sum of the benefits of all the intervals present in the schedule. Let L_i be the newly created set of schedules created by adding an interval to L_{i-1} and discarding sets that do not satisfy the cache space limit. The trimming procedure is applied to L_i as follows:

1. Sort the L_i in non-decreasing order of the benefit of the schedules.

2. Divide the schedules present in L_i in groups, such that ratio of the minimum and maximum benefit of schedules, in a group is less then $(1 - \delta/n)$, where n is number of *intervals* in set S (the set of all the intervals).

   Thus the number of groups present in the set is $n \log(B)/\delta$, where B is the maximum benefit of any schedule in L_i. Derivation of this result is same as that of the number of elements in the trimmed set of *subset sum* problem described in section 4.1.

3. L_i is trimmed such that, in each group number of schedules retained is polynomial. For this, firstly consider the graph of space occupied by the schedule versus time, shown in Figure 3. In this graph we will only consider the q discrete points in the time axis, which represent the change in the cache content when a query completes execution.

   We define $maxSize_1(s,t)$ for a schedule s at time t as the maximum cache size occupied by cached sub-expressions after time t when the queries are executed according to schedule s. For a group, at every point in time t when an interval ends, only the schedule s whose $maxSize_1(s,t)$ is least at some point in time is retained and rest of the schedules are discarded from L_i.

   Thus after trimming each group will have $O(q)$ number of schedules, where q is the number of discrete points at which an interval can end or begin, which is twice the total number of queries.

Since we have $O(n)$ number of groups, the total number of schedules in the set L_i after trimming will also be polynomial. The complexity of this algorithm can be shown to be $O(n^2 q)$.

The basic intuition for this trimming procedure is that we should not discard schedules from L_i that can be combined with the newly added interval to form a high benefit schedule. And at the same time existing schedules cannot be combined with the newly added interval. Let a newly added interval start at the time t (intervals are added in ascending order of their finishing time). Suppose the interval can be combined with a schedule s, that was discarded from L_i in the trimming procedure. Then there will exist a schedule p, whose group is same as group of s, that will have $maxSize_1(p,t)$ less then $maxSize_1(s,t)$. So the newly added interval can also be combined with p. As s and p are in same group so they have almost same benefit thus we are losing benefit of $(1 - \delta/n)benefit_s$ (in the worst case) by discarding s.

Let S be any schedule consisting of a subset of the first j intervals sorted by finishing times. The following statement can be proved by induction on j. For any time t, at the end of the jth iteration, there exists

a schedule S' saved by the algorithm, with benefit at least $(1 - \frac{\delta}{n})^j$ times the benefit of S, such that, the $maxsize(S', t)$ is at most $maxsize(S, t)$. The rest of the steps required to prove that this algorithm is a constant ratio algorithm are similar to the steps of the proof for the approximation algorithm for the *subset sum* problem.

The approximation ratio of this algorithm is same as that of approximation algorithm of *subset sum*, namely $\frac{1}{1-\delta}$ We will call this approximation algorithm as $\mathbf{A}_{et}$ (et represents adding intervals in ascending order of *ending time*).

5 Ordering of Queries

In this section we describe another algorithm for interval packing problem, which is similar to the algorithm described in Section 4.2, but removes the assumption that the order of queries is given. In this section we continue to use the term *interval* to refer to the cached result of a shared subexpression.

The algorithm uses a modified form of the interval packing algorithm. In algorithm $\mathbf{A}_{et}$ for the interval packing problem, described in Section 4.2, intervals are added to the existing set of schedules in ascending order of their finishing time. Consider a different approach for the interval packing problem, in which intervals are added to the existing set of schedules in descending order of their starting time; we call this algorithm $\mathbf{A}_{st}$. Intuitively $\mathbf{A}_{st}$ is the same as $\mathbf{A}_{et}$ (described in Section 4.2), except that instead of using $maxSize_1(s, t)$, $\mathbf{A}_{st}$ uses $maxSize_2(s, t)$ which is defined as the maximum size occupied by schedule s before time t. The rest of the procedure for grouping and trimming the set of schedules remains same as in $\mathbf{A}_{et}$. Both $\mathbf{A}_{et}$ and $\mathbf{A}_{st}$ are polynomial time algorithms, and behave in a similar manner.

5.1 A Heuristic For Query Ordering

This section describes a heuristic ordering algorithm $\mathbf{OA}_{et}$ for deciding the order in which queries should be evaluated.

The ordering heuristic is as follows:

1. First choose a pair of queries that has the maximum sharing. That is, consider each pair of queries, compute the total benefit of all common sub-expressions shared by the pair, and choose the pair whose common sub-expressions have the maximum total benefit. Let Q_{i1} and Q_{i2} be the chosen pair.

2. Initialize the sequence $Q = Q_{i1} \ . \ Q_{i2}$; this sequence represents the order in which queries should be executed.

3. At each step choose the query $Q_i \notin Q$, such that executing Q_i after all the queries in Q (in the order specified by Q) gives the maximum benefit; that is, choose the Q_i that maximizes
   $(\Sigma_{q \in Q}$ cost of $q)$ + cost of Q_i − cost of $Q \cdot Q_i$
   The cost of a sequence is found using algorithm $\mathbf{A}_{et}$. Append Q_i to the sequence Q: $Q = Q \cdot \{Q_i\}$.

Algorithm $\mathbf{OA}_{et}$ is similar to $\mathbf{A}_{et}$ in the following sense. In $\mathbf{A}_{et}$, intervals are added in ascending order of their finishing time. In $\mathbf{OA}_{et}$ queries are appended to Q. Appending a query Q_i to Q is just like adding a set of intervals to the existing set of schedules, one for each common subexpression between Q_i and earlier queries in Q; since all the intervals are ended by Q_i intervals are added, in effect, in ascending order of their finishing time. Thus, in $\mathbf{OA}_{et}$ a set of schedule L_i is tracked, and each time a query Q_i is added to Q, a set of intervals is added to L_i.

For example, let Q_1, Q_2 and Q_4 share the result of subexpression E. Let $Q = Q_1 \cdot Q_2 \cdot Q_3$ represent the order decided by the algorithm in previous steps. When Q_4 is considered to be added to the existing schedule which contains E as an interval ending at Q_2, the interval corresponding to E gets extended from Q_2 to Q_4. Extending an interval is equivalent to adding an interval E' from Q_2 to Q_4, such that $size_{E'}$ is equal to size of result of expression E, and $benefit_{E'}$ is equal to the benefit of caching E from Q_2 to Q_4.

The time complexity of $\mathbf{OA}_{et}$, like that of $\mathbf{A}_{et}$ is $O(n^2 q)$, where q is the number of queries.

We define another algorithm $\mathbf{OA}_{st}$, which is just like $\mathbf{OA}_{et}$, but it adds the new query at the beginning of the existing schedule. Thus, algorithm $\mathbf{OA}_{st}$ parallels algorithm $\mathbf{A}_{st}$.

Now we combine algorithms $\mathbf{OA}_{st}$ and $\mathbf{OA}_{et}$ to form algorithm $\mathbf{OA}$ as described below. The algorithm takes as parameter a value δ.

1. Choose the pair of the queries that, if executed together, have the maximum benefit. This procedure is similar to the first step of algorithm $\mathbf{OA}_{et}$.

2. Choose a query Q_p, which when executed at the end or beginning of the existing schedule gives the largest benefit.

3. Add the chosen query to all schedules present in L_i, where L_i is the list of all the schedule. New schedules (created after adding a query) are inserted to the list L_i, and L_i is then trimmed by following procedure:

   (a) Evaluate the benefit (described in 4.1) of every schedule. Divide the schedules into

groups such that, ratio of the minimum and maximum benefit of the schedule in a group is less than $(1 - \delta/n)$, n is the number of shared sub-expression.

(b) In each group a schedules s is retained if it has least value of $maxSize_1(s,t)$ or $maxSize_2(s,t)$ at some point in time t. And rest of the schedules are discarded.

4. A new query can only be added at the end of a schedule $s \in L_i$ which has the least value of $maxSize_1(s,t)$, at some time t. On other hand, a new query can only be added at the beginning of a schedule s present in L_i that has least value of $maxSize_2(s,t)$, at some time t.

5. Repeat from Step 2 until all the queries are chosen.

5.2 Analysis of algorithm OA

At each iteration of **OA**, a query is picked and it is added at the beginning or at the end of the existing schedules. So now for a particular order of evaluation of queries there can be many schedules, and also there can be many schedules belonging to different evaluation orders of the queries. Let us assume that at the i^{th} step Q_i is chosen to be executed next, then the search space of **OA** will be as shown in Figure 4.

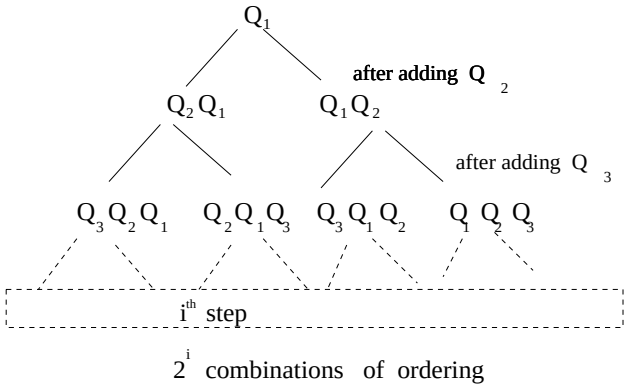

Figure 4: Tree representing the order of the queries

But note that **OA** is not an exponential algorithm, since the set of schedules is trimmed each time a query is added to it. The trimming procedure does not consider order of evaluation of queries (of the schedule) when trimming L_i. The basic reason for not considering order of queries in trimming procedure is that in the future queries will only be added to the start and end of the schedule, so all the schedules (corresponding to different query orders) can be treated as the same. The trimming procedure makes sure that the number of schedules tracked is polynomial.

5.3 Generalized Ordering Algorithm

One of the assumptions that we made in Section 3 was, if the result of the expression e is cached and later discarded from the cache then it will never be cached again. This assumption can be removed by keeping many intervals for one shared sub-expression as a candidate for getting admitted in cache. Intervals belonging to the same sub-expression should have different start and end points and be non-overlapping. As our algorithm is a cost based algorithm, it will ensure that two intervals belonging to the same shared sub-expression are not present in schedule in the same time.

In algorithm $\mathbf{OA}_{et}$, each time a query Q_k is added at the end of *schedule*, an interval e (which represents the shared expression E, which is used by Q_k) is extended to the point where Q_k is added. But now, to add a query to the schedule, the following steps are performed :

1. Let s be the *schedule* in which Q_k is to be added and let $(Q_{e1}, Q_{e2}, Q_{e3}, \ldots, Q_{en}, Q_k)$ be the queries that use result of E, as shown in Figure 5.

   Let Q_{ep} be the last query in the schedule s that uses E. Adding Q_k to s implies adding following intervals to s

   - Interval from Q_{ep} to Q_k
   - Interval from $Q_{e(p+1)}$ to Q_k

     $\vdots$

   - Interval from $Q_{e(n)}$ to Q_k

2. After adding a query to all the schedules we trim the list of schedules by the procedure as described in Section 4.2.

Thus when we add a query to a set of schedules, in the worst case the number of schedules in the set becomes q times what it was earlier, where q is the number of the queries. But the trimming procedure again makes sure that the size of set of schedules is polynomial. Given a particular order of the queries, it becomes a constant ratio approximation algorithm. In the similar manner we can modify algorithm **OA** so that it can decide to cache shared results that were discarded from the cache earlier.

6 Related Work

There had been work on multiple-query optimization in past. Early work had concentrated on exhaustive algorithms[11, 4]. The multi-query optimization problem has been addressed in [2, 4, 12]. These papers discuss the problem of searching an efficient execution plan for evaluation of a batch of queries. In this report

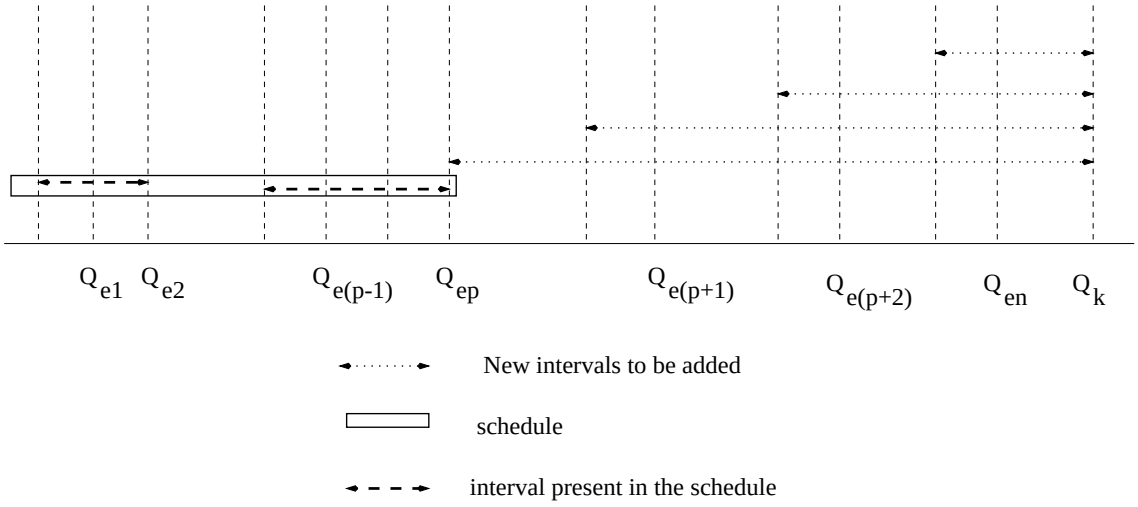

Figure 5: Adding a query to a schedule

we have considered the problem of multi-query optimization with space constraints. None of the related work addresses the issue of query scheduling with a fixed size cache.

The problem of materialized view and index selection given a fixed size cache is considered in [5, 6], but here the decision is static, based on a workload of queries, and does not change as queries are evaluated. These algorithms find the best set of sub-expressions to materialize (cache for the execution of all the queries), for optimizing a workload consisting of both queries and updates. The multi-query optimization problem differs from the above since it can assume absence of updates, and all queries are given as a batch.

The problem of dynamic caching of query results is considered in [10, 7, 9], where the issue is to dynamically choose what to cache, based on the current access pattern of the queries. Query optimization with efficient update of cached results is also discussed in [9]. All these address the case where the future queries are not known, and queries have to be evaluated as they are received. In contrast, we deal with a case where an entire batch of queries is available, and we can reorder the queries as required.

The problem of expression DAG scheduling has been discussed in [1]. But that paper only consider scheduling of a particular kind of DAG structure with some assumptions such as fixed result size, which is simplified case of the general problem of query scheduling.

7 Conclusion

In this paper we have addressed the problem of query scheduling in multiquery optimization, which has not been addressed earlier. We made several assumptions to simplify the problem, and gave approximation algorithms for the case when the order of queries is given. Later we presented two greedy heuristics for choosing the execution order of queries, and merged the two heuristics to form a combined heuristic.

We showed how to relax some of the assumptions. However, in this paper we continued to assume that the benefits of a subexpression do not depend on what else has been cached. Finding approximation algorithms without making this assumption is an interesting direction for future work. Finding an approximation algorithm, rather than just a heuristic, for the case when the query order is not given is another direction of future work. Another area of future work lies in implementing our algorithms and heuristics and comparing them with the optimal choice on some sample benchmarks.

References

[1] Sandip K. Biswas and Sampath Kannan. Minimizing space usage in evaluation of expression trees. In *Int'l Conf. on Foundations of Software Technology and Theoretical Computer Science*, 1995.

[2] C. M. Chen and N. Roussopolous. The implementation and performance evaluation of the adms query optimizer. In *Extending Database Technology*, March 1994.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[4] Ahmet Coser, Ee-Peng Lim, and Jaideep Srivastava. Multiple query optimization with depth-first

branch-and-bound and dynamic query optimization. In *Intl. Conf. on Information and Knowledge Management*, 1993.

[5] H Gupta. Selection of views to materialize in a data warehouse. In *Intl. Conf. on Database Theory*, 1997.

[6] V. Harinarayan, A. Rajaraman, and J Ullman. Implementing data cubes efficiently. In *SIGMOD Intl. Conf on Management of Data*, 1996.

[7] Arthur M. Keller and Julie Basu. A predicate based caching scheme for client-server database architecture. *VLDB Journal*, 5(1):35–47, 1996.

[8] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddesh Bhobe. Efficient and extensible algorithms for multi query. In *SIGMOD Intl. Conf. Management of Data*, 2000.

[9] Nicholas Russopoulos and Yannis Kotidis. Dynamat : A dynamic view management system for warehouse. In *SIGMOD Intl. Conf on Management of Data*, 1999.

[10] Peter Scheuermann, Junho Shim, and Radek Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of VLDB Conference*, pages 51–61, 1996.

[11] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering*, 12:197–222, 1994.

[12] Subbu N. Subramaniam and Shivkumar Venkataraman. Cost based optimization of decision support queries using transient views. In *SIGMOD Intl. Conf on Management of Data*, 1998.