Multiple Query Optimization by Cache-Aware Middleware using Query Teamwork

K. O'Gorman D. Agrawal A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106
{kogorman,agrawal,amr}@cs.ucsb.edu

August 9, 2002

Abstract

The multiple-query optimization (MQO) problem has been well-studied in the research literature, usually by means of identifying and exploiting the occurence of common subexpressions, and has required implementation in the database engine. Observing that common subexpressions derive from common data, and that the amount of data is usually greatest at the source, we propose an optimization technique that exploits the presence of sharable access patterns to underlying data, especially scans of large portions of tables or indexes, in environments where query queueing or batching is an acceptable approach. We show that simultaneous queries with such sharable accesses have a tendency to form synchronous groups (teams) which benefit each other through the operation of the disk cache, in effect using it as an implicit pipeline. We propose a novel method of optimization, exploiting this tendency by scheduling the queries to enhance this tendency, and show that this can be accomplished even from outside the database engine with application server middleware. We present an algorithm for scheduling from a queue of similar queries, designed to promote such teamwork. This is implemented as middleware for use with a commercial database engine. Finally, we present tests using the query mix from the TPC-R benchmark, achieving a speedup of 2.34 over the default scheduling provided by the database.

1 Introduction

When a database system is presented with multiple queries, opportunities arise for optimizing the group of queries as a whole as well as the operations to evaluate the individual queries. This has been explored as the multiple query optimization (MQO) problem, and has generally been approached on the basis of batches of queries and detailed analysis of the query execution plans [Jar85, PS88, Sel88, CLS93, CD98, RSSB00]. A common theme

among most of these proposals is to identify common subexpressions among the queries and materialize them temporarily so that they can be shared among various queries.

Several variations on this idea have been recently proposed. Tan and Lu propose scheduling query fragments to optimize the use of data left behind in memory, thus reducing disk usage [TL95], or scheduling symmetric multiprocessors to access the shared data simultaneously [TL96]. Our method does not alter the query plans generated by the database engine, but instead makes use of properties of those plans as generated. Zhao et al. [ZDNS98] propose new join operators together with pipelining to optimize for particular similarities that arise in the context of OLAP queries. In contrast, our approach requires no new operators or other capabilities within the database engine. Dalvi et al. [DSRS01] propose using pipelining of data between processes accessing the same data or results. The proposed solution must deal with two deadlock problems associated with the use of pipes. First, considering the pipes as directed edges in a graph with queries as vertices, directed cycles represent a deadlock situation; an algorithmic solution is proposed to prevent this configuration. Even then it can happen that the corresponding undirected graph has a cycle, which may represent deadlock due to bounded buffering in the pipes where the configuration of pipes constrains consumers and producers of the data in the pipes to operate at different speeds. The paper assumes this possibility will be addressed by dynamically choosing to materialize the pipeline contents.

All of these approaches involve rewriting the query plans for the batch of queries, and thus require modifications to the optimizer or query plan generator of the database engine. Moreover, these approaches deal with the workload in batches rather than incrementally. Our method does not suffer from these drawbacks: it can be implemented as an incremental or batch method without modification to, or extensive knowledge of the internals of, the database engine. Further, our method involves no constraints between queries and cannot introduce deadlock; it operates by scheduling groups of queries to begin together, but enforces nothing beyond that initial synchronization. Finally, although our method requires information about the query plans to be used by the queries, the same is true of all of the MQO methods of which we are aware; we will demonstrate how the results can vary with the amount and kind of information known.

Our approach uses in-memory sharing of objects even though they may be much larger than the physical memory of the machine, enabled by self-synchronizing queries. We show that such query behavior arises naturally in queries that share disk access patterns such as scans, which is related to the characteristics exploited by previous MQO research. The approach exploits common access patterns to the underlying data, where traditional MQO approaches exploit common subexpressions; observing that common subexpressions derive from common data, we expect the approach to be useful in many if not most environments in which MQO provides benefits. In the experiments reported here, the optimizations are performed from outside of the database engine, in middleware, so that the method can be used with existing commercial database products; of course, the method could also be incorporated into the database engine and one would expect the results to be at least as good as reported here. Our results show that significant speedups are possible without rewriting queries, query plans or existing optimizers. Our approach is orthogonal to many of the existing optimization techniques, and may even be more effective with those that schedule query fragments.

The rest of the paper is organized as follows. Section 2 provides motivation and some initial results. Section 3 supports the claim that queries autonomously form cooperating teams under favorable conditions. Section 4 presents an algorithm to schedule queries from a queue, promoting team formation. Section 5 presents the performance comparison results. Section 6 discusses the results.

2 Motivation and Initial Experiments

In general queries are thought to compete for resources, specifically for use of the disk cache, and to thereby interfere with and impede each other's execution. This situation can be alleviated in various ways, as when caches are used to exploit the sharing of data between queries. For instance, small tables (relative to memory size) can be shared in a single-thread environment, mediated by scheduling fragments of the original query plans [TL95]. We explore the scheduling of concurrent processes to a similar end, but without size limits or query fragmentation. We include in this section the results of some initial

experiments, by way of motivation and showing the feasibility of our approach.

2.1 Experimental Setup

We explored the behavior of the 22 queries of the TCP-H benchmark [Tra]. These queries are parameterized templates, and are accompanied by a query generator program to supply randomly chosen parameter values. This program, QGEN, was used to generate a set of 40 query streams, some of which were used in the tests described here. The instances of a given query in the several query streams were parameterized differently by QGEN, with parameters which affected selection and join criteria in the query. As a result no two queries in any run were identical, because queries from the same template were always instantiated with different constants.

Because the initial results showed unreasonably bad performance for three queries even when run in isolation, we modified these queries, so as to improve performance of each one by at least a factor of two [OAA00]. We reasoned that such bad query plans would be fixed in a similar way in any installation interested enough in performance to use the methods presented here, but our own reason was that as it stood, these three queries dominated the time of the runs, and made initial verification of our setup difficult because of the huge timeouts required. The queries changed were as follows: Query 4 was rewritten to replace a coordinated EXISTS subquery with a join. Query 8 was rewritten by adding an optimizer hint to its subquery, resulting in a full table scan, which was much faster than the lookup through an index used by the original query. Query 20 was rewritten to replace the coordinated IN subquery with a join to a subquery.

Our initial experiments, described in the next section, were run on a system with a single IDE hard drive. All other experiments were run on 4 identical systems with dual Pentium-III processors, 1GB RAM, an IDE hard drive, and 4 SCSI hard drives, and the main results presented are the mean value of the runs on all 4 systems. For these main results, the operating system and Oracle were stored on the IDE drive, and the Oracle system and user schemas were stored on the SCSI drives; the LINEITEM table had one SCSI drive, the other tables shared a drive, and all indexes shared a third. All other database objects were stored on a fourth SCSI drive. Rather than using the normal Linux ext2 filesystem format, the

Query Number	Teaming Speedup
1	3.78
2	3.07
3	4.91
4	4.95
5	2.20
6	1.24
7	4.77
8	4.96
9	3.11
10	3.41
11	2.96

Query Number	Teaming Speedup
12	4.96
13	1.17
14	5.01
15	4.95
16	3.06
17	3.43
18	3.87
19	1.48
20	3.43
21	4.89
22	4.68

Figure 1: Team Formation of Individual Query Types. Five instances of each query were started 2 seconds apart, and again sequentially to determine the speedup.

user schema and temporary files were stored in raw partitions on the SCSI drives, so that all buffering occurred in the Oracle buffer pool, and the Linux buffer pool was bypassed. The Oracle buffer pool was set at 10,000 8K blocks, or 80 megabytes, or about 2.5% of the size of the database and indexes. We reasoned that this ratio was reasonably representative of buffer sizes for large databases. The systems have 1GB of RAM each, so that using the Linux buffer pool could have buffered over 25% of the database, had we allowed this.

2.2 Initial Experiments

We tested the tendency of the TPC queries to cooperate by running five instances of each query running almost simultaneously, i.e. started 2 seconds apart over a period of 8 seconds. The instances were parameterized as indicated for query streams 1 through 5 in the benchmark, so that queries were instantiated with different constants. The running times for the set of five queries was compared to the running time for the five instances running sequentially. Figure 1 shows the results of the test. Since the CPU was observed to be less than 50% utilized throughout the tests, and the system had a single hard disk, the speed improvements reflect the change in the number of physical I/O requests. For example, the entry for Query 1 is 3.78, indicating that the 5 queries running sequentially took 3.78 times as long as it took to run them concurrently. The speedup ranged from a low of 1.17 for Query 13, which did not benefit much from this sharing, to a high of 5.01 for Query 14. One would expect a theoretical maximum of 5, so we ascribe the additional 0.01 to measurement error. This *team advantage* appeared even though most queries access more data from disk than will fit in the disk cache. In effect, the teams were synchronizing their operations so that the limited disk cache could be used as a data pipeline shared within the team.

Initial Separation	Stream 1 Time	Inter-Stream Ending Separations 1-2	2-3	3-4	4-5	Total Time
0	689	0	0	0	0	689
10	736	0	0	0	0	737
20	752	0	0	0	0	752
30	748	0	0	0	0	748
40	750	0	0	0	0	750
50	1683	188	74	8	0	1953
60	1607	230	88	23	0	1948
70	1547	273	105	38	0	1963
80	1486	304	120	51	3	1964

Figure 2: Teaming behavior with different initial time separations. The execution time of the first query is shown, then the time separations to the completion of the rest of the queries, and the total time for the run. Nonzero values of +1 or -1, which represent the measurement granularity, have been set to zero.

The team advantage persists in the presence of the 2-second delays between starting the queries, indicating that this behavior is reasonably robust in the presence of scheduling disturbances. Figure 2 illustrates the change in teaming behavior of one of the queries[1] with different time separations between the instances; happily this was the first query template we investigated – a few of the others would actually have shown interference between instances from the same template. The figure may be best explained by example; consider the line with an "initial separation" of 50. This line describes the behavior of five instances of the query, started at 0, 50, 100, 150 and 200 seconds into the test run. The first query finished 1683 seconds into the run, the second query finished 188 seconds later, the third query finished 74 seconds after that, and the fourth query finished in another 8 seconds. The final query finished with no delay after the fourth. Accordingly, the run lasted for 1953 seconds. We interpret these numbers to say that the 5^{th} query instance had overcome the 50 second head start and had "caught up" to the 4^{th} query instance; moreover, this pair had nearly caught up to the 3^{rd} query instance, so that this line shows team formation in operation. The team advantage persists even when the teams are not that obvious; even at the 80 second separation, the total time is some 40% less than for a sequential execution. In this example, the team advantage was present even in queries started 80 seconds apart; however, in general we have found the advantage degrades with delays of 10 seconds or more.

[1] The query is anonymous because we are prohibited from publishing actual times for any particular query. These results represent 5 instantiations of one of the TPC queries in which processing time was dominated by a full table scan.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	2.02																					
2	1.06	0.97																				
3	1.00	1.00	1.00																			
4	1.16	1.09	1.00	1.03																		
5	1.25	1.34	1.00	1.09	1.27																	
6	1.04	0.91	1.00	0.91	0.81	0.90																
7	1.00	1.00	1.00	1.00	1.00	1.00	1.01															
8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01														
9	1.28	1.44	1.00	0.87	0.96	1.01	1.00	1.00	1.27													
10	1.29	1.07	1.00	1.22	1.04	0.93	1.00	1.00	1.03	1.48												
11	1.04	0.90	1.00	1.20	0.98	0.73	1.00	1.00	1.37	1.27	0.93											
12	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.10										
13	1.54	1.04	1.00	1.06	1.18	1.08	1.00	1.00	1.14	1.39	1.19	1.00	0.67									
14	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01								
15	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.00	1.00	1.00	1.00	1.00	1.00							
16	1.12	1.02	1.00	1.21	1.39	0.94	1.00	1.00	1.25	1.54	0.99	1.00	1.05	1.00	1.00	0.89						
17	1.02	0.97	1.00	0.82	0.89	1.00	1.00	1.00	1.42	0.92	0.92	1.00	1.04	1.00	1.00	1.00	1.00					
18	1.37	0.99	1.00	0.75	0.89	0.87	1.00	1.00	1.08	1.01	1.07	1.00	1.68	1.00	1.00	1.17	0.92	0.88				
19	1.01	1.00	1.03	1.00	1.00	1.00	1.02	1.02	1.00	1.00	1.00	1.01	1.00	1.02	1.03	1.00	1.00	1.00	1.02			
20	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.02		
21	1.26	1.01	1.00	0.75	0.83	0.82	1.00	1.00	0.93	0.97	1.05	1.00	1.39	1.00	1.00	1.18	1.00	1.10	1.01	1.00	1.98	
22	1.05	0.96	1.00	0.76	0.95	0.82	1.00	1.00	0.96	0.87	1.00	1.00	1.05	1.00	1.00	0.92	0.91	1.00	1.01	1.00	1.10	0.97

Figure 3: Potential speedup from reducing accesses to the most-used disk. The entries are ratios of the accesses from queries run sequentially to queries run simultaneously, for queries instantiated from the given pair of query templates. The results shown are the mean of results from five runs, each using a different set of parameter values drawn from the specifications for runs 31-40. The best value (2.02) and worst value (.67) are both on the diagonal. Our later experiments required teams to have pairwise values of at least 1.00 for all pairs in the team.

2.3 Heterogeneous Teams

From the experiments so far, it appears that queries instantiated from the same template are capable of forming teams, but there may exist other possible groupings of queries that can exhibit this behavior. To test this hypothesis, we tested all pairs of queries for team formation; pairs of queries were run, first sequentially and then concurrently, and the running times were compared. The queries were instantiated by using parameters indicated for streams 31 through 40 in the benchmark, in 5 pairs, each pair being run once on each of the 4 test systems. The concurrent runs were started together rather than with the 2-second separation of the tests that generated Figure 1. Potential speedups can be derived from the logs of these tests in several ways, by comparing elapsed time, disk accesses, and so on. Because the critical resource for our setup appeared to be accesses to the SCSI disk containing the LINEITEM table, we used a comparison of number of accesses to that disk, as shown in Figure 3. An entry of 1.00 means there was no advantage or disadvantage, i.e. the concurrent run made the same number of accesses as running one query after the other. Entries under 1.00 indicate that the two queries actually interfere with each other. Entries over 1.00 represent the team advantage: the concurrent run made fewer accesses

to the critical disk than the sequential run. Because the entries are focused on a single resource, quite a few of them are 1.00, showing that this pair does not affect that resource; nevertheless, we will show that these entries can be the basis of an effective optimization.

3 Teams and Team Formation

We found query behavior that suggested that careful scheduling could improve overall system throughput where there were exploitable similarities among the queries. We call the behavior *team formation*. This behavior depends on the details of access to secondary storage, and particularly the disk cache, and of the queries involved.

Operating systems and database systems maintain caches of disk blocks from secondary storage, so that requests for frequently-used or recently-used information can be satisfied from the comparatively fast RAM rather than from the secondary storage device. As a result, operations that request information already present in the disk cache will execute much more quickly than ones that request information that must be retrieved from secondary storage. This is not strictly a property of the operations themselves, however, but is a dynamic property of the system as a whole. The contents of the disk cache will evolve over the course of time, in response to the activity experienced by the system, and this will affect team formation.

3.1 Disk Cache Management

For concreteness, we now discuss cache management in the specific context of an Oracle database. When a query is run and accesses tables stored on disk, it fills the disk cache as much as possible with the blocks that it needs, but eventually must replace some of the blocks it used early on in the process with blocks it needs later, because there is not enough room for all of them. This can happen in two ways in Oracle, depending on whether a full table scan is involved. The default for most operations is to keep blocks according to an LRU (least recently used) algorithm, so that blocks are evicted from the disk cache when they have been used less recently than any others. Full table scans are treated differently, and by default place blocks at the "least recent" end of the LRU list after they are used,

since it is considered unlikely that they will be used again before they have to be evicted from the cache anyway. It is possible, however, to set a per-table CACHE/NOCACHE attribute so that full table scans are not treated specially, to let such blocks be placed at the most-recent end of the LRU list. The Oracle documentation recommends doing this only for small, frequently-used tables, presumably to avoid the phenomenon of "cache wiping" which might cause a full table scan to fill the cache with blocks which will not be reused before they are evicted. In contrast, in our tests, this CACHE attribute was set for the largest table in the schema, the LINEITEM table, because we had determined that in the worst case it imposed a minimal cost (under 1%) on throughput, but in most experiments it yielded a major improvement to throughput. The Oracle documentation also states that there is a per-query hint to the optimizer that can control this behavior, but we have been unable to verify that it works in Oracle 8.1.7 on Linux, and it is not used in the experiments reported here. No matter how the blocks are handled, at the end of execution of a query unless the data accessed is small in relation to the size of the disk cache, it is unlikely that the data used at the beginning of its execution is still in the disk cache; such data will have been evicted in favor of the data needed later.

3.2 Why Teams Form and Persist

To see how this operates in team formation, we will use Query 10 from the TPC benchmark as an example. Judging from Figure 3, the overall run time improves (speedup of over 1.00) when it runs concurrently with other instances of itself, or with instances of queries 1, 2, 4, 5, 9, 11, 13, 16, and 18. We conclude this query may beneficially form teams with those other queries. The query itself has this form:

```
select <some attributes and aggregates>
from customer, orders, lineitem, nation
where
   c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate >= date '&date'
   and o_orderdate < date '&date' +
```

```
      interval '3' month
   and l_returnflag = 'R'
   and c_nationkey = n_nationkey
group by <some attributes>
order by revenue desc;
```

This query, like all the TPC queries, is actually a query template. In this case, the single parameter `&date` is required to instantiate the query. In our schema there are indexes that contain most of the data required from the tables, so that the ORDERS and LINEITEM indexes are scanned rather than the full table. The query is evaluated by three hash joins, involving a range scan on an index to ORDERS and full scans of index or table data for the other three relations. The index on LINEITEM alone is over twice the size we set for the Oracle buffer pool, so that we expect the first data accessed to be evicted from the buffer pool before the query is finished.

Any two instances of this query will access many of the same disk blocks in the same order, although the calculations on them may be different because different subsets of the data will be selected for inclusion in the results. If they are executed sequentially, as noted above, the second query to execute will encounter a disk cache that is empty of the data required for its execution; this data has been previously used but later evicted. Accordingly, the data will be read again. However, if the queries are started together and run concurrently, they have an opportunity to use the disk cache as a kind of pipeline to avoid the second reading of the secondary store.

To see how this works, let us suppose that two instances of the query are started a few seconds apart, as in some of our experiments. The first query to begin will read disk blocks into the disk cache and perform calculations, but let us further suppose that before this can proceed to the point where its inputs are evicted from the disk cache, the second instance of the query begins execution. At this point, the second instance encounters a disk cache that already contains the information that it requires to begin its calculations. Accordingly, the second instance of the query can execute more quickly because it does not have to wait for the comparatively slow secondary storage operations. It will make progress more quickly than the first instance did at the same point, until the second instance has

"caught up" in its progress to the same point, as concerns access to secondary storage, as the first instance of the query. At this point, the two instances will be requesting the same blocks from secondary storage. Because the database engine itself has safeguards to handle such situations, only one copy of each requested block will be read, and it will be used by both instances of the query.

To the extent that this general description applies to a particular case, one can expect the teams to be stable, self-organizing and self-maintaining so long as the two instances make the same requests of the secondary store. If one of the instances somehow makes more progress, and is reading blocks that the other instance has not yet used, the same team formation process as just described will cause the slower instance once again to catch up. This is a very strong tendency due to the orders of magnitude difference in speed between physical disc access and the other activities in the system. As shown in Figure 3 while this tendency is even present in some pairs of non-identical queries (e.g. 1.68 for Query 18 paired with Query 13), it may be weak in some pairs of queries from the same template (e.g. 0.67 for Query 13 teamed with itself). This presumably arises when the access patterns for queries from a given template are sensitive to the parameters, and can differ greatly between instances.

4 Middleware Design

We now describe an application server middleware that is specifically designed to take advantage of affinities among queries being submitted to a database. Our application server is shown in Figure 4. We assume the query sources act as streams; each source submits one query at a time, and waits for the system's response before submitting the next query. We provision the database with a fixed number of server connections for the submission of queries, each connection corresponding to a separate database server process. The number of servers will in general be less than the number of query streams, and therefore the middleware will queue incoming queries for submission to a server at an appropriate time, depending on the queueing policy in force. Our goal is to schedule client queries to execute simultaneously on different database server processes to take advantage of potential affinities

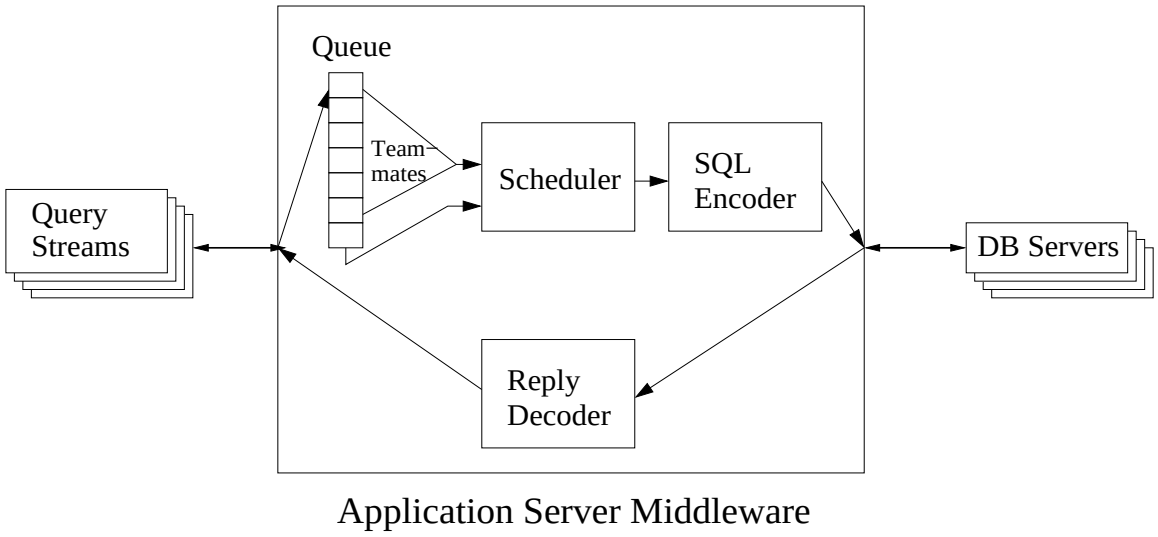

Figure 4: Middleware structure

among them, and we will choose queueing policies accordingly. In this section we present the scheduling algorithm and then prove some of its interesting properties.

4.1 Scheduling Algorithm

The unit of scheduling is the team. All queries in a team start at the same time, and the team is considered to be running until the last member finishes, at which point the scheduler schedules new work for a new team. This is referred to as the *team cycle*. Database server processes have no permanent connection to a team, but are assigned to the team for the duration of a single query. A server process that finishes before other members of its team is immediately available for joining any new team. However, each team always has at least one server process, even if the team is idle, so that when there is work, the team can proceed with at least that one server process, and perhaps more.

The scheduling algorithm is shown in Figure 5. It selects queries as team members at the beginning of each team cycle. The query at the head of the queue is always chosen to be one of the members, and is called the *team leader*, because of the way it affects the selection of the other members. If it happens that there are additional database server processes available, the team selection algorithm is invoked, to select another query to be started at the same time. As many additional queries as there are available database server processes

```
procedure TeamScheduler
  loop
    team ={query at the head of the queue}
    remove the query from the head of the queue
    while (there are idle database server processes)
          and (the queue contains a suitable
            teammate):
      reserve a database server process
      team = team∪{teammate}
      remove the teammate from the queue
    end while
    start all elements (members) of team
    wait for a teammate to finish
    while there is more than one teammate:
      release the database server process of the
        finished teammate
      wait for a teammate to finish
    end while
  end loop
end procedure
```

Figure 5: Scheduling algorithm, executed for each team

may be started, up to the number of queries in the queue that qualify as teammates of the team leader.

The additional queries added to the team are selected according to their *affinity* for the leader, according to an affinity rule. This rule is a set of *affinity sets*, which are sets of query numbers whose instances can be included in the same team. A team consists of queries instantiated from templates whose numbers are members of the same affinity set, or comprises a single query which is not in any affinity set.

Perhaps the simplest nontrivial rule is one that forms teams only from queries instantiated from the same template. It can be written formally as a set of singletons comprising the template numbers. For our template set, for instance it is written as {{1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22}}. When we initially tested this rule, it performed reasonably on a single-disk system, performing about half as well as the best rule we developed. However, on the multiple-disk systems we used for our main results, this rule gave no consistent advantage over non-team performance, and we do not report the details here.

Affinity rules can also be developed from data on query behavior, such as that in Figure 3.

```
procedure Affinity(threshold, teamtable)
  rule ={}
  for each querynumber:
    if a selfpair of querynumber meets threshold:
      rule = rule∪{{querynumber}}
    else:
      set all teamtable entries for querynumber to
        threshold
  end for
  while ((minimum value in teamtable) < threshold):
    find the minimum value in teamtable between
      members of different affinity sets, set1 and set2
    if these sets can be unified:
      rule = rule − {set1} − {set2}
      rule = rule ∪ {{set1 ∪ set2}}
    set teamtable entries connecting set1 and set2
      to threshold
  end while
  return rule
end procedure
```

Figure 6: Greedy algorithm to find an affinity rule for a given threshold, and a given teamtable of pairwise team advantages

By requiring teams to be made up of templates with pairwise team advantages over some given threshold, one can ensure that team members are paired with compatible queries, even though they may not share a template. For example, we used a greedy algorithm of Figure 6 to generate rules from the data of Figure 3. The algorithm takes as input the table of team advantages and the threshold value. For some threshold values, this algorithm would reject some of the teams formed by the first rule. For example, for our data a threshold value of 1.00 rejects queries 13 and 18 from participating in any multi-query teams including pairing instances from the same template, because they have (dis)advantage for pairs of their own instances of 0.67 and 0.88, and even though their team advantages together are a very good 1.68. However, this same 1.00 threshold would allow a rule containing the affinity set {1 4 5 8 10 15} because all of the pairs, including self-pairs, that can be formed from this set have team speedups better than 1.00 threshold.

The algorithm begins with singleton affinity sets, one for each query which met the threshold, i.e. corresponding to qualifying values on the diagonal of the figure. All values for non-qualifying queries (e.g. queries 13 or 18) were then set to the threshold value so

they would not be considered for team membership. Each subsequent step of the algorithm considered the pair of affinity sets represented by the minimum value remaining in the table corresponding to different sets. The sets are combined if the resulting set will contain no pair of queries that violates the threshold requirement. If the sets cannot be combined, their table entries are altered so that pair will not be the minimum again. The algorithm ends when the minimum entry itself no longer meets the threshold requirement. With different threshold values, different affinity sets are formed by this algorithm.

The scheduler is concerned with starting the queries, but is not concerned with their ongoing execution other than to detect completion. Once started, the teammates may or may not cooperate, and may or may not remain synchronized according to the accuracy of the criteria chosen for query affinity. The team is considered to be running until all of the teammates have completed, at which point a new team is scheduled if there is work in the queue.

If there are just as many database server processes as teams then each team necessarily has exactly one database server process and the algorithm degenerates to normal round-robin FIFO scheduling with the given number of servers. More generally, at any given time a server can be a team leader, a teammate, or idle awaiting work. Servers can be idle because they finished first among their team, because the queue did not contain queries with suitable affinity for a team being started, or because the queue did not contain any queries.

4.2 Notes on Correctness and Complexity

It is easily shown that the scheduling algorithm has the useful properties of liveness and freedom from deadlock.

Theorem 1 (Liveness) *Every query entered into the queue is eventually served.*

Each team finishes because the number of queries in the team is fixed when the team begins, and each query is finite. The finishing of each team causes a scheduling event (i.e. a call to procedure schedule of Section 4.1). Each scheduling event either finds the queue empty, in which case the theorem is trivially true, or else serves the query at the head of the

queue. Thus the stream of scheduling events does not stop, and any queries in the queue advance toward the head or are served during that event. By induction on the size of the queue, every query is eventually served, proving the theorem. □

The synchronization of teams is not enforced by any mechanism in our algorithm or elsewhere. If the queries selected for a team fail to cooperate, they may become de-synchronized and make progress more slowly, but they will continue to make such progress as the database engine normally provides. This absence of locks or synchronization primitives for coordination leads trivially to Theorem 2.

Theorem 2 (Deadlock) *The scheduling algorithm of Figure 5 is deadlock-free.*

The complexity of most of the operations involved in this method is linear or constant, with two exceptions. One is the development of the array of team advantages (such as Figure 3), which is quadratic in the number of query templates, but is only calculated once (under our assumption of a static set of templates). The other is the search for teammates while serving the queue, which in the worst case is proportional to the product of the queue length and the number of available servers for the teammates.

5 Performance Results

Of the query streams mentioned in Section 2, we used query streams 1 through 20 as the workload for our tests, so that each test comprised the running of 440 queries, none of which was used in deriving the affinity rules. Considering each such query stream as simulating a client, there were 20 concurrent clients in each test. For database server processes, we actually ran SQL*Plus processes, the interactive client for Oracle. Each SQL*Plus client then made the connection to the database, which assigned the server process. The number of these servers varied from run to run. The middleware communicates with the SQL*Plus processes through I/O redirection and Linux pipes.

The clients and the middleware are implemented as a combination of Expect [Lib95] and SQL*Plus scripts, with the scheduler being a single function of about 100 lines of code

Condition	Values
# of Clients	20
# of Servers	1-15
# of Teams	1-8
Constraint	# of Servers $\geq$ # of Teams
Affinity Rule	Derived, threshold = 1.00

Figure 7: The setup conditions for testing. There was also a baseline of 20 teams and 20 servers, with an empty affinity rule.

Database Servers	Number of Teams						
	2	3	4	5	6	7	8
1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
2	1.34	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
3	1.54	1.39	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
4	1.72	1.60	1.46	$\perp$	$\perp$	$\perp$	$\perp$
5	1.79	1.81	1.64	1.50	$\perp$	$\perp$	$\perp$
6	1.94	1.84	1.79	1.65	1.53	$\perp$	$\perp$
7	2.01	1.91	1.89	1.82	1.63	1.55	$\perp$
8	1.99	2.09	2.02	1.92	1.73	1.64	1.50
9	2.06	2.14	2.10	2.08	1.89	1.75	1.69
10	2.12	2.13	2.12	2.10	1.96	1.88	1.76
11	2.17	2.21	2.18	2.15	2.07	1.93	1.79
12	2.17	2.22	2.21	2.18	2.12	2.04	1.86
13	2.11	2.23	2.31	2.22	2.16	2.06	1.97
14	2.15	2.22	2.29	2.28	2.20	2.08	1.94
15	2.09	2.25	2.34	2.25	2.17	2.24	2.00

Figure 8: Speedups from teamwork, comparing running time to the baseline at 20 database servers and 20 teams. Entries marked $\perp$ are prohibited by the constraint of Figure 7.

in one of the Expect scripts. The scripts perform the timing functions, and ensure logging of all results.

The various test runs differed as to the number of teams and the number of database servers (SQL*PLus processes), as shown in Figure 7. The schema and queries of the TPC-R benchmark [Tra] were used, scaled to about a gigabyte of raw data (i.e. SF=1), and about two gigabytes of indexes were built, essentially those of [Gra99] but without the clustered index reported there.

5.1 Affinity Rule Performance Analysis

We used the greedy algorithm of Figure 6 to develop candidate affinity rules for the data in Figure 3, and found that 31 different affinity rules could be derived from that data. We present results obtained using the affinity rule produced from the threhsold value of 1.00,

ensuring that all pairs of members in a team were scored as at least neutral in the pairwise tests of Figure 3. This resulted in the affinity rule {{9 12 14} {1 4 5 8 10 15} {3 19} {7 20 21}}. The results for this rule are shown in Figure 8. The best result is 2.34 at 4 teams and 15 servers; we did not test beyond 15 servers because with only 20 query streams the queue was becoming drained, and the improvement at 15 servers compared to 14 was not large. Some of the other thresholds produced slightly better results in preliminary tests, but the threshold value of 1.00 has the virtue of having a simple rationale *a priori*: limiting the presence of harmful pairs.

Besides a beneficial effect on throughput, queuing has an effect on the latency and response times of individual queries. This is so because the various queries have very different execution times, spanning two orders of magnitude, but queueing introduces a delay which is constant across all queries, pushing the overall response times towards an average. For very fast queries in the mix, this delay increased overall running time by a factor of 66, and teamwork did not reduce this significantly because the actual running time was so small compared to queueing delays. Overall the effect was positive, with 6 of the queries benefittng from queueing, and 16 being slowed, 6 of them by a factor of 10 or more. The latter 6 had such short running times that they did not contribute significantly to the speedup results for teamwork, which suggests that to the extent such queries can be identified in advance, they would be better served by another means and this would not significantly dilute the advantages of teamwork for the rest of the queries.

5.2 Alternative Algorithms

It seemed possible that the results obtained in Section 5.1 were not due to the construction of the affinity groups, but might be obtained by merely grouping queries randomly. We tested this in several ways, by forming groups of queries according to some alternate protocols.

First, we attempted to promote team behavior by simple batching. Instead of forming teams on the basis of the template from which the queries derive, we simply ran queries in batches of a given size, formed according to the order of arrival of the queries. In other respects, the experiments were the same as the results reported above, and were accomplished by a minor modification to the queueing strategy of the driver program.

Team Size	Number of Teams				
	1	2	3	4	5
4	1.37	1.36	1.40	1.40	1.30
5	1.38	1.38	1.37	1.32	$\perp$

Figure 9: Speedups from batching, comparing running time to the baseline at 20 database servers and 20 teams. Entries marked $\perp$ are prohibited by the number of query streams.

We performed the experiments with teams of varying sizes, either 4 or 5, and with varying limits on the number of concurrent teams, from 1 team to the maximum allowed by the availability of queries. The resulting speedups are shown in Figure 9, and are actually slowdowns from pure queueing (1.55). Accordingly, it appears that some information about the query is needed to effectively promote team formation.

Next, we attempted to randomize the affinity groups, which we call *random groups*. This is not a well-defined notion, so we used three different protocols for forming these random groups. For each protocol, five different random seeds were used to derive five different experiments, and the corresponding test was run on the same four systems as were used in our main results, for a total of twenty timing runs for each protocol. We report the mean result of the twenty experiments for each protocol.

Protocol A formed the same size affinity groups as in our experiments, but used a random permutation of the query templates, producing an overall speedup of 2.02. The five random seeds produced five different random groups as follows:

- {{ 7 19 15} { 4 9 2 14 21 20} {12 8} { 5 1 17}}
- {{22 20 9} {17 10 14 18 16 7} {15 12} {19 8 6}}
- {{14 5 3} { 2 22 4 21 11 8} {18 20} { 7 13 19}}
- {{20 3 22} { 1 14 12 8 15 11} {21 18} { 5 7 19}}
- {{22 20 2} {17 10 12 7 21 1} {13 3} {16 4 14}}

Protocol B formed four affinity groups, but their sizes were not constant. In each trial, queries were assigned to each group with probability weighted by the size of the groups in the experiments reported in Figure 8. This produced a speedup of 1.97. The five runs produced the following groups:

- {{11 12 16 17 18 22} {1 3 5 6 10 13} {15} {2}}
- {{3 20 22} {1 8 12 14 15} {11 21} {5 18}}
- {{} {2 6 11 17 20 22} {} {3 5 8 9 13 15 16}}
- {{5 7 12 13 19} {6 8 22} {11} {14 15 20}}
- {{12 21} {6} {7 8 20} {2 16}}

Protocol C formed four affinity groups, and each query was assigned each, or to be in no affinity group, with equal probablility of 0.2 for each case, producing a speedup of 2.06. The five runs produced the following groups:

- {{3 8 9 11 14 20 21} {7 15} {5 10} {2 6 12 19}}
- {{1 2 3 7 16 22} {10 12 13 18} {5 11 14 15 17 19} {3 6 9 20}}
- {{15 20 21 22} {4 8 13} {} {1 2 3 10 11 16 19}}
- {{1 9 11 12} {4 7 8 14 22} {6 19 21} {2 5 10 15 18 20}}
- {{7 11 12 15 17} {3 4 8 20} {5 6 9 18 22} {1 10 14}}

Protocols B and C could and did occasionally produce an empty set, effectively reducing the number of affinity groups. This did not appear to affect the results. The mean speedups for the three protocols were very similar, being A: 2.02, B: 1.97, and C: 2.06; these are intermediate between the speedup for queueing alone (1.55) and the result for full teamwork scheduling (2.34).

5.3 Analysis of Results

In order to obtain these results, we have done more than introduce an affinity rule; we have altered the way that queries are scheduled. The question therefore arises of the degree to which the other changes may have contributed to the success of our approach. In order to explore this we ran another series of tests. In both series, we ran the scheduler with the same number of processes as teams, effectively ensuring each team had a single member.

Processes	LINEITEM Cache	
	Off	On
1	1.24	1.22
2	1.28	1.34
3	1.34	1.39
4	1.38	1.46
5	1.38	1.50
6	1.36	1.53
7	1.36	1.55
8	1.35	1.50
9	1.31	1.47
10	1.29	1.44

Processes	LINEITEM Cache	
	Off	On
11	1.15	1.41
12	1.01	1.28
13	0.93	1.16
14	0.94	1.05
15	0.98	1.09
16	1.02	1.12
17	1.02	1.11
18	1.05	1.15
19	1.03	1.16
20	1.00	1.16

Figure 10: The effects of the queue and of the CACHE attribute. The entries are speedups compared to the naive setup and scheduling (reported here for 20 processes and cache off). Runs were made with the indicated number of processes to serve 20 query streams, both with and without the CACHE attribute specified for the LINEITEM table.

In the first series, we did not make the changes to the CACHE attribute of the LINEITEM table reported in Section 3.1. In the second series, these changes were made. The results are shown in Figure 10, and show that in the absense of team selection, the CACHE attribute provides a slight advantage on this query workload. Without the CACHE attribute, the best speedup is 1.38 at 4 and at 5 processes, and with the CACHE attribute this increases to 1.50 at 7 processes.

A speedup due to queueing is reasonable given that the hardware setup has 7 primary resources that are used during the tests: 4 SCSI drives, and IDE drive (for the logs), and 2 CPUs. Scheduling more processes than 7 may introduce more contention for resources than assistance with the computation. The speedup from the CACHE attribute on the LINEITEM table may be due to the workload imposing its heaviest load on the drive containing that table, and perhaps having more of the blocks of that table in the buffer pool is more of an advantage than the cache-wiping phenomenon is a disadvantage, at least for this schema and workload.

Based on these results, we conclude that the queueing contributes a speedup of up to 1.38, that cacheing the LINEITEM table contributes perhaps an additional speedup of 1.09, and that for this workload, the teamwork contributes a speedup of at least 1.56, since $1.50 \times 1.09 \times 1.56 = 2.34$. Presumably, the queueing speedup could also be obtained from Oracle's Multithreaded Server (MTS) option which does the same thing, but this would not be compatible with middleware scheduling for team formation.

The failure of pure batching to provide any improvement, and the modest success of random groups, suggest that some information about the queries' query plans is needed in

order to effectively promote team formation. In our case, the query template number is a proxy for this information, and is sufficient to produce the benefit seen on this workload.

6 Discussion

Tradtitionally, multi-query optimization is achieved at the level of the database engine. In general, this approach to multi-query optimization requires modifications to the underlying database management system. In this paper, we propose an alternative approach to multi-query optimization that obviates the need for such modification. However, if it should be convenient to make such changes, we expect that the benefits would be at least as great as when implemented as middleware. We discovered that significant speedups were available from middleware by substituting cooperation for contention by scheduling compatible queries together. Such improvement is important in itself inasmuch as some of the TPC-R queries can run for tens of minutes each on our setup. We have described the query behavior that leads to cooperating query teams and measured this behavior under a benchmark workload that represents a variety of business queries. We designed an incremental scheduling algorithm to promote team formation, with little ad-hoc information about the query workload. Our examples show improvements with an affinity rule that generates heterogeneous teams. We demonstrate that substantial throughput improvements are achievable with a lightweight scheduler operating as a middleware between the clients and an unmodified commercial database. Our experiments with alternate algorithms in Section 5.2 suggest that team formation can be promoted in a variety of ways, and expect that this will be possible in many if not most workloads which are amenable to the other techniques of Multiple Query Optimization. However, this will require at least some information to support the selection of candidate teams. Our future work will explore methods for dynamically determining team formation.

References

[CD98] Fa-Chung Fred Chen and Margaret H. Dunham. Common subexpression processing in multiple-query processing. *IEEE Transactions on Knowledge and Data*

Engineering, 10(3), May June 1998.

[CLS93] Ahmet Cosar, Ee-Peng Lim, and Jaideep Srivastava. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In Bharat K. Bhargava, Timothy W. Finin, and Yelena Yesha, editors, *CIKM 93, Proceedings of the Second International Conference on Information and Knowledge Management, Washington, DC, USA, November 1-5, 1993*, pages 433–438. ACM, 1993.

[DSRS01] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Principles of the Twentieth SIGMOD-SIGACT-SIGART Symposium of Principles of Database Systems (PODS)*, Santa Barbara, CA, May 2001.

[Gra99] Goetz Graefe. The value of merge-join and hash-join in SQL Server. In *Proceedings of the 25th International Conference Very Large Databases*, pages 250–253, Edinburgh, Scotland, 1999.

[Jar85] Matthias Jarke. Common subexpression isolation in multiple query optimization. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 191–205. Springer, Berlin, Heidelberg, 1985.

[Lib95] Don Libes. *Exploring Expect.* O'Reilly & Associates, Inc., 1995.

[OAA00] Kevin O'Gorman, Amr El Abbadi, and Divyakant Agrawal. On the importance of tuning in incremental view maintenance: An experience case study. In *Data Warehousing and Knowledge Discovery Second International Conference, DaWaK 2000*, London, UK, September 2000.

[PS88] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the 4th International Conference on Data Enginering*, pages 311–319, Los Angeles, CA, February 1988.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the SIGMOD International Conference on Management of Data*, Dallas, Texas, May 2000.

[Sel88] Timos K. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[TL95] Kian-Lee Tan and Hongjun Lu. Workload scheduling for multiple query processing. *Information Processing Letters*, 55(5):251–257, September 1995.

[TL96] Kian-Lee Tan and Hongjun Lu. Scheduling multiple queries in symmetric multiprocessors. *Information Sciences*, 95(1&2):125–153, 1996. Elsevier Science Publishing Inc, North-Holland.

[Tra] Transaction Processing Performance Council. *TPC-R Benchmark Specification.* URL http://www.tpc.org/rspec.html.

[ZDNS98] Y. Zhao, Prasad Deshpande, Jeffrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, in SIGMOD Record*, 27(2):271–282, June 1998.