SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:361–391
Published online 21 December 2004 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.640

Multiple query optimization in middleware using query teamwork

K. O'Gorman[1,*,†], A. El Abbadi[2] and D. Agrawal[2]

[1] *Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407, U.S.A.*
[2] *Computer Science Department, University of California, Santa Barbara, CA 93109, U.S.A.*

SUMMARY

Multiple concurrent queries occur in many database settings. This paper describes the use of middleware as an optimization tool for such queries. Since common subexpressions derive from common data and the data is usually greatest at the source, the middleware exploits the presence of sharable access patterns to underlying data, especially scans of large portions of tables or indexes, in environments where query queuing or batching is an acceptable approach. The results 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. The middleware exploits this tendency by queuing and scheduling the queries to promote this interaction, using an algorithm designed to promote such teamwork. This is implemented as middleware for use with a commercial database engine. The results include tests using the query mix from the TPC Benchmark™R, achieving a speed-up of 2.34 over the default scheduling provided by one database. Other results show that the success depends on the details of the computing environment. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: query teamwork; multiple query optimization; middleware

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 [1–7]. A common theme among

*Correspondence to: Kevin O'Gorman, Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407, U.S.A.
†E-mail: kogorman@csc.calpoly.edu

Contract/grant sponsor: NSF; contract/grant numbers: EIA-0080134 and IIS-0223022

Copyright © 2004 John Wiley & Sons, Ltd.

Received 16 January 2003
Revised 1 July 2004
Accepted 1 July 2004

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 [8], or scheduling symmetric multiprocessors to access the shared data simultaneously [9]. The middleware proposed here 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*. [10] propose new join operators together with pipelining to optimize for particular similarities that arise in the context of OLAP queries. In contrast, the middleware requires no new operators or other capabilities within the database engine. Dalvi *et al*. [11] 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. The middleware proposed here 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. Furthermore, the middleware imposes 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 teamwork promotion requires information about the query plans to be used by the queries, the same is true of all of the MQO methods of which the authors are aware; the test results demonstrate how the outcome can vary with the amount and kind of information known.

The proposed 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. The test results and discussion 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; since common subexpressions derive from common data, the approach should be useful in many, if not most, environments in which MQO provides benefits. Most queries consume the most resources in processing at the leaves of the execution tree, because of the benefits of performing data-reducing selections and projections at that level; accordingly an optimization directed at that level may have significant effects on the overall execution. The teamwork approach aims to share the results of data retrieval when queries require the same leaf-level data, and thus are operating where the bottleneck may exist.

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. Conversely, when changes to the database engine are not possible, this affords a possible optimization through middleware, at least for some databases, that has

Copyright © 2004 John Wiley & Sons, Ltd.

not been explored previously. The test results show that significant speed-ups are possible without rewriting queries, query plans, or existing optimizers. The teamwork approach is orthogonal to many of the existing optimization techniques, and may even be more effective with those that schedule query fragments.

1.1. Overview

The rest of this 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 an environment in which teamwork performs poorly. Section 7 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 [8]. Teamwork schedules concurrent processes to a similar end, but without size limits or query fragmentation. This section includes the results of some initial experiments, by way of motivation and showing the feasibility of the approach.

2.1. Experimental set-up

The experiments explored the behavior of the 22 queries of the TPC Benchmark(TM) R [12]. 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, further testing used modified queries, so as to improve performance of each one by at least a factor of two [13], with the justification 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 the more pressing reason was that, as it stood, these three queries dominated the time of the runs, and made initial verification of the experimental 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; the modified query is shown in Figure 1. 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; the modified query is shown in Figure 2. Query 20 was rewritten to replace the coordinated IN subquery with a join to a subquery; the modified query is shown in Figure 3.

The initial experiments, described in the next section, were run on a system with a single IDE hard drive. All other experiments were run on four identical systems with dual Pentium-III processors,

Copyright © 2004 John Wiley & Sons, Ltd.

```
-- Q04p - order priority checking query (parameterized)
-- THIS IS AN UNAPPROVED VARIANT RECODING OF THE QUERY

define stream=&1
define date=&2

select
        o_orderpriority,
        count(*) as order_count
from (
        select distinct
                o_orderpriority,
                o_orderkey
        from
                orders,
                lineitem
        where
                o_orderdate >= date '&date'
                and o_orderdate < date '&date' + interval '3' month
                and l_orderkey = o_orderkey
                and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority;
```

Figure 1. Query 4 as recoded for speed on Oracle.

1 GB RAM, an IDE hard drive, and four SCSI hard drives, and the main results presented are the mean value of the runs on all four systems. For these main results, the Linux 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 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. This ratio seemed a reasonable representative of buffer sizes for large databases. The systems have 1 GB of RAM each, so that using the Linux buffer pool could have buffered over 25% of the database, had this been allowed.

2.2. Initial experiments

We tested the tendency of the TPC-R [12] queries to cooperate by running five instances of each query starting almost simultaneously, i.e. started 2 s apart over a period of 8 s. The instances were

Copyright © 2004 John Wiley & Sons, Ltd.

```
-- Q08p - national market share query (parameterized)
-- THIS IS AN UNAPPROVED VARIANT RECODING OF THE QUERY

define stream=&1
define nation='&2'
define region='&3'
define type='&4'

select
        o_year,
        sum(case
                when nation = '&nation'
                then volume
                else 0
        end) / sum(volume) as mkt_share
from (
        select /*+ ALL_ROWS FULL(lineitem) */
                extract(year from o_orderdate) as o_year,
                l_extendedprice * (1-l_discount) as volume,
                n2.n_name as nation
        from
                part,
                supplier,
                lineitem,
                orders,
                customer,
                nation n1,
                nation n2,
                region
        where
                p_partkey = l_partkey
                and s_suppkey = l_suppkey
                and l_orderkey = o_orderkey
                and o_custkey = c_custkey
                and c_nationkey = n1.n_nationkey
                and n1.n_regionkey = r_regionkey
                and r_name = '&region'
                and s_nationkey = n2.n_nationkey
                and o_orderdate between date '1995-01-01' and date '1996-12-31'
                and p_type = '&type'
        ) all_nations
group by
        o_year
order by
        o_year;
```

Figure 2. Query 8 as recoded for speed on Oracle.

Copyright © 2004 John Wiley & Sons, Ltd.

```
-- Q20p - potential part promotion query (Parameterized)
-- THIS IS AN UNAPPROVED VARIANT RECODING OF THE QUERY

define stream=&1
define color=&2
define date=&3
define nation=&4

select distinct
        s_name,
        s_address
from
        supplier, nation, partsupp, part,
        (
                select
                        l_partkey,
                        l_suppkey,
                        0.5 * sum(l_quantity) as limitval
                from
                        lineitem
                where
                            l_shipdate >= date '&date'
                        and l_shipdate  < date '&date'  + interval '1' year
                group by
                        l_partkey,
                        l_suppkey
        )
where
            p_partkey = ps_partkey
        and p_name like '&color%'
        and s_suppkey = ps_suppkey
        and l_partkey = ps_partkey
        and l_suppkey = ps_suppkey
        and ps_availqty > limitval
        and s_nationkey = n_nationkey
        and n_name = '&nation'
order by
        s_name;
```

Figure 3. Query 20 as recoded for speed on Oracle.

Copyright © 2004 John Wiley & Sons, Ltd.

Table I. Team formation of individual query types. Five instances of each query were started 2 s apart, and again sequentially to determine the speed-up.

Query number	Teaming speed-up	Query number	Teaming speed-up
1	3.78	12	4.96
2	3.07	13	1.17
3	4.91	14	5.01
4	4.95	15	4.95
5	2.20	16	3.06
6	1.24	17	3.43
7	4.77	18	3.87
8	4.96	19	1.48
9	3.11	20	3.43
10	3.41	21	4.89
11	2.96	22	4.68

parameterized as indicated for query streams 1–5 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. Table I 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 five query instances running sequentially took 3.78 times as long as it took to run them concurrently. The speed-up 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; presumably the additional 0.01 is 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.

The team advantage persists in the presence of the 2 s delays between starting the queries, indicating that this behavior is reasonably robust in the presence of scheduling disturbances. Table II illustrates the change in teaming behavior of one of the queries[‡] with different time separations between the instances; happily this was the first query template investigated—a few of the others would actually have shown interference between instances from the same template, and might have dissuaded pursuit of this line of inquiry. 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,

[‡]These results represent five instantiations of one of the TPC queries [12]; it was originally chosen because its running time seemed convenient for investigation. The query is anonymous because we are prohibited from publishing actual times for any particular query.

Copyright © 2004 John Wiley & Sons, Ltd.

Table II. 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. Non-zero values of $+1$ or -1, which represent the measurement granularity, have been set to zero.

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

150, and 200 s into the test run. The first query finished 1683 s into the run, the second query finished 188 s later, the third query finished 74 s after that, and the fourth query finished in another 8 s. The final query finished with no delay after the fourth. Accordingly, the run lasted for 1953 s. We interpret these numbers to say that the fifth query instance had overcome the 50 s head start and had 'caught up' to the fourth query instance; moreover, this pair had nearly caught up with the third query instance, so that this line shows team formation in progress. The team advantage persists even when the teams are not that obvious; even at the 80 s separation, the total time is some 40% less than that for a sequential execution. In this example, the team advantage was present even in queries started 80 s apart; however, in general the advantage degrades with delays of 10 s or more.

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, all pairs of queries were tested 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 31–40 in the benchmark, in five pairs, each pair being run once on each of the four test systems. The concurrent runs were started together rather than with the 2 s separation of the tests that generated Table I. Potential speed-ups 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 the experimental setup appeared to be accesses to the SCSI disk containing the LINEITEM table, the tests compared the number of accesses to that disk, as shown in Table III. 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

Copyright © 2004 John Wiley & Sons, Ltd.

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.

One might expect Table III to show its greatest improvements for pairs of queries that involve many references to the LINEITEM table, especially scans. However, of the five entries in the figure showing a speed-up over 1.50, only the {21,21} pair is of this type. Because the entries are focused on a single resource, quite a few of them are near unity, showing that this pair does not have much effect on that resource. Moreover, the non-unity entries do not take into account the level of support from the underlying data—the absolute value of the differences encountered—so that a change of 1 out of 10 is counted the same as a change of 100 out of 1000. In spite of efforts to completely evict TPC schema blocks from the Oracle buffer pool between tests, using a query involving a large index scan on a different schema, there was frequently a residual activity on the disk containing the LINEITEM table, even during queries which made no reference to that object at all. Nevertheless, the experiments show that these entries can be the basis of an effective optimization.

3. TEAMS AND TEAM FORMATION

The query behavior suggested that careful scheduling could improve overall system throughput for suitable workloads. This *team formation* 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 it 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, consider 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 a least recently used (LRU) 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

Copyright © 2004 John Wiley & Sons, Ltd.

Table III. Potential speed-up from reducing accesses to the most-used disk.

	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

Copyright © 2004 John Wiley & Sons, Ltd.

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, for these experiments the CACHE attribute was set for the largest table in the schema, the LINEITEM table, because earlier experiments has shown 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 considerable experimentation failed 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, consider Query 10 from the TPC Benchmark R [12] as an example. Judging from Table III, the overall run time improves (speed-up 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. Evidently this query may beneficially form teams with those other queries. The query itself has the following 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. The schema in use includes 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 set for the Oracle buffer pool, so that the first data accessed is 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

Copyright © 2004 John Wiley & Sons, Ltd.

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 the 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 Table III, 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.

3.3. Formal model

In analyzing database performance, it is usually assumed that the only significant factor is the amount of disk I/O that is performed. As a first approximation, this is usually a reasonable starting point, and is used here. It will turn out that under the assumption that the CPU costs for all operations can be ignored (i.e. are effectively instantaneous,) teamwork behavior can be described in simple closed-form formulas. The model is further embellished by also modeling the operation of a simple LRU buffer-cache, so that even some disk operations (the buffer cache hits) are instantaneous, and only the cache misses take time. As a result, the ratio of misses to total disk requests will be a crucial determinant of how the model performs. Of course, as optimization of disk I/O improves there is a strain on the underlying assumption that other things can be ignored, as in the extreme case where the entire database fits in the buffer pool, all operations are cache hits and nothing takes any time at all (in the idealized model, at least). Nevertheless, for reasonable values, the model will serve for the present purpose.

The model includes a simple LRU buffer-management policy, managing a buffer pool of B blocks, serving a workload consisting of some number N of work streams $W_i = \{b_{i,1}, b_{i,2}, \ldots, b_{i,n_i}\}$, where each $b_{i,j}$ represents a request for a block from secondary storage. The model further assumes that the time to complete the workload is proportional to the number IOS of the requests that result in actual disk I/O (i.e. that are not satisfied by a cache hit on the buffer cache).

Copyright © 2004 John Wiley & Sons, Ltd.

A schedule over this workload is an ordered sequence

$$S = (b_{i,j}, b_{k,l}, \ldots, b_{y,z})$$

such that each $b_{i,j}$ of the workload occurs exactly once in S, and for all m, p, q if $b_{m,p}$ occurs in the sequence before $b_{m,q}$ then $p < q$. Alternatively, schedules are sometimes denoted with single subscripts when the focus is on the order of requests rather than on the work streams they come from. An annotated schedule

$$AS = ([\star]b_i, [\star]b_j, \ldots, [\star]b_z)$$

is a schedule in which the star symbol ($\star$) may (or may not) appear before some of the block requests in the schedule. A block request with the star represents a *cache hit*, and a block request without the annotating star is a *cache miss*.

There are counting functions on the domain of single requests, or subsequences of annotated schedules as follows:

$$\begin{aligned} req(AS) &= \{\text{the number of block requests in } AS\} \\ hit(AS) &= \{\text{the number of } \star \text{ symbols in } AS\} \\ miss(AS) &= req(AS) - hit(AS) \qquad (1) \\ block(b_{k,l}) &= \{\text{the identity of the block requested by } b_{k,l}\} \end{aligned}$$

If $AS = (b_1, \ldots, b_j, \ldots, b_n)$ is an annotated schedule, and $SS = (b_i, \ldots, b_j)$ exists and is the shortest subsequence such that $block(b_i) = block(b_j)$, define

$$dwell(b_j) = req(SS)$$

Note that a cache miss always moves the previous contents of the buffer pool one step toward the least-recent end of the pool, evicting the extreme LRU block. It is also true that a cache hit reorganizes a part of the buffer pool, and while it does not evict any blocks it usually bumps some of them down one position while advancing the block which was the subject of the current request. Thus, if there are many cache hits, other blocks may be bumped down very quickly. This is modeled as cache dwell time for a block and a function to describe the conditions for bumping a block. If $AS = (b_1, [\star]b_2, \ldots, \star b_l)$ is an annotated schedule ending with a cache hit $\star b_l$, then

$$bump(AS, j) = \begin{cases} 1 & \text{if } j < l, block(b_j) = block(b_l), \\ & \forall m, j < m < l \Rightarrow block(b_m) \neq block(b_l), \\ & \exists i, k \text{ s.t. } 1 < i < j < k < l, \\ & block(b_i) = block(b_k), \\ & \forall m, i < m < k \Rightarrow block(b_m) \neq block(b_k) \text{ and} \\ & b_k \text{ is a cache hit} \\ 0 & \text{otherwise} \end{cases}$$

An annotated schedule $AS = (b_1, \ldots, b_n)$ is *correct* for a buffer pool B of size n if for every cache hit $\star b_p$ there is a subsequence $HS = ([\star]b_i, [\star]b_j, \ldots, \star b_p)$ of AS, in which $block(b_i) = block(b_p)$, and $\forall k, i < k < p \Rightarrow block(b_k) \neq block(b_p)$ and $n > miss([\star]b_i, \ldots, \star b_p) + \sum_{j=1}^{p} bump(AS, j)$.

Copyright © 2004 John Wiley & Sons, Ltd.

For subsequence SS of an annotated schedule there are corresponding measures of *hit ratio* and *miss ratio*:

$$HR(SS) = \frac{hit(SS)}{req(SS)}, \quad MR(SS) = \frac{miss(SS)}{req(SS)}$$

As a consequence of Equation (1),

$$HR(SS) + MR(SS) = 1$$

Because only cache misses take time, the duration of the schedule is

$$IOS(AS) = req(AS) * MR(AS)$$

It is useful to note that any sequence SS of block requests which completely replaces the contents of the buffer pool must have a duration of at least B disk I/O operations, and its length is bounded by $req(SS) \geq B/MR(SS)$; this defines a *time horizon* for finding a requested block still in the buffer pool. The time duration of a schedule AS is defined to be $dur(AS) = atime \times IOS(AS)$, where $atime$ is the access time for disk I/O. Reasonable numbers for the equipment used for the experiments are

$$atime = 0.003 \text{ s}, \quad B = 10\,000$$

With these values, if the buffer pool were completely ineffective (hit ratio zero, miss ratio 1) the time horizon would be $0.003 \times (10000/1) = 30$ s, meaning that team formation should be possible for suitable queries started up to a half a minute apart. In this idealized model, the time horizon depends directly on the miss ratio. For example, for the time horizon to be 1 s, the miss ratio must be 1/30, and the hit ratio is correspondingly 29/30 or 96.66%. This is reasonable; team formation can be viewed as a method to improve the utilization of the buffer pool as measured by the hit ratio, and it is hard to do that if it is already very good.

4. MIDDLEWARE DESIGN

With team behavior having been seen in preliminary experiments, more complete experiments used an application server middleware that is specifically designed to take advantage of affinities among queries being submitted to a database. The application server is shown in Figure 4. The experiments model the query sources as streams; each source submits one query at a time, and waits for the system's response before submitting the next query. The experimental setup provisions 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 queuing policy in force. The goal is to schedule client queries to execute simultaneously on different database server processes to take advantage of potential affinities among them, and this will be reflected in the choice of queuing policies. This section presents the scheduling algorithm and then proves some of its interesting properties.

Copyright © 2004 John Wiley & Sons, Ltd.

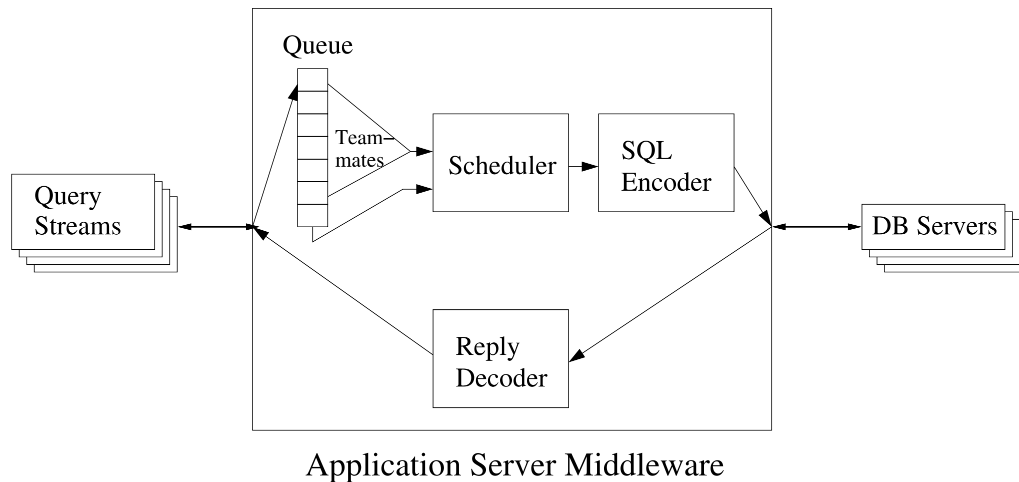

Figure 4. Middleware structure.

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 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 non-trivial 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.

Copyright © 2004 John Wiley & Sons, Ltd.

```
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. The scheduling algorithm, executed for each team.

For the TPC template set 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}}. In initial tests, it performed reasonably on a single-disk system, performing about half as well as the best rule that was found. However, on the multiple-disk systems we used for our main results, this rule gave no consistent advantage over non-team performance, and is not reported further here.

Affinity rules can also be developed from data on query behavior, such as that in Table III. 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, the greedy algorithm of Figure 6 generated rules from the data of Table III, taking that 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 Table III, 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 at 1.68 are very good. However, the 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 speed-ups better than the 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

Copyright © 2004 John Wiley & Sons, Ltd.

```
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. The greedy algorithm to find an affinity rule for a given threshold, and a given teamtable of pairwise team advantages.

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 the 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.

Copyright © 2004 John Wiley & Sons, Ltd.

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

Proof. 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 the 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 towards 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 the middleware 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 Table III), which is quadratic in the number of query templates, but is only calculated once (under the assumption of a static set of templates). The other is the search for teammates while serving the queue, whose complexity is bounded in the middleware algorithm (for each team start) by a linear function of the number of entries in the queue, and thus is a linear function of the number of clients, the number of servers, and the number of teams.

5. PERFORMANCE RESULTS

Of the query streams mentioned in Section 2, query streams 1–20 were the workload for the 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. The middleware connected to database server processes through 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 [14] and SQL*Plus scripts, with the scheduler being a single function of about 100 lines of code in one of the Expect scripts. The scripts perform the timing functions, and ensure logging of all results.

The various test runs differed in the number of teams and the number of database servers (SQL*Plus processes), as shown in Table IV. The schema and queries of the TPC Benchmark R [12] were used, scaled to about 1 GB of raw data (i.e. $SF = 1$), and about 2 GB of indexes were built, essentially those of [15] but without the clustered index reported there.

5.1. Affinity rule performance

The greedy algorithm of Figure 6 developed candidate affinity rules for the data in Table III, resulting in 31 different affinity rules. The results here were obtained using the affinity rule produced from the

Copyright © 2004 John Wiley & Sons, Ltd.

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

Condition	Values
No. of clients	20
No. of servers	1–15
No. of teams	1–8
Constraint	No. of servers $\geq$ No. of teams
Affinity rule	Derived, threshold = 1.00

Table V. 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 Table IV.

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

threshold value of 1.00, ensuring that all pairs of members in a team were scored as at least neutral in the pairwise tests of Table III. 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 Table V. The best result is 2.34 at four teams and 15 servers; the experiments were stopped at 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.

There were also tests of the effects of the CACHE attribute by running tests with simple queuing, varying the number of server processes. As shown in Figure 7, despite the intuitions and documentation to the contrary, for this workload at least, it is better to invoke the CACHE attribute for the LINEITEM

Copyright © 2004 John Wiley & Sons, Ltd.

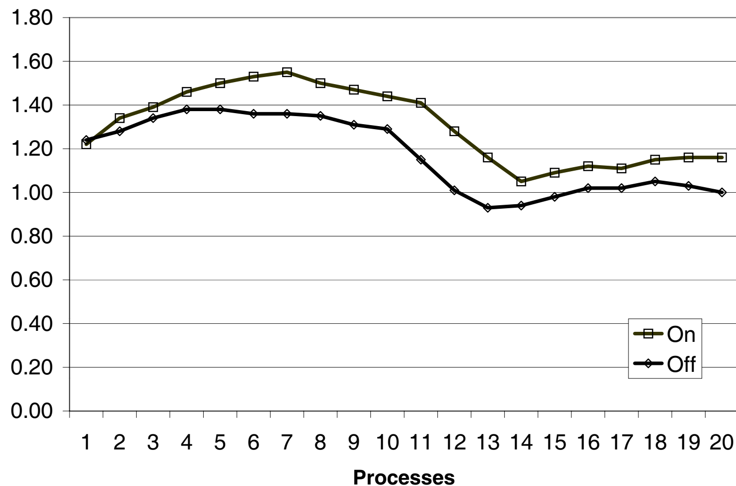

Figure 7. Effects of the cache attribute on performance. The total time to complete the workload is shown as a function of the number of server processes. The Y-axis is speed-up from naive setup and scheduling. The upper trace had the CACHE attribute on for the LINEITEM table, the lower (slower) trace had the attribute off.

table, producing a speed-up of about 1.09. This benefit is not seen for any of the other tables in these tests; perhaps this is because the LINEITEM table is the main fact table, and by far the single most active object in the schema under this workload.

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 queuing 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 queuing delays. Overall the effect was positive, with six of the queries benefiting from queuing, and 16 being slowed, six of them by a factor of 10 or more. The latter six had such short running times that they did not contribute significantly to the speed-up results for teamwork, which suggests that to the extent that 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. Schedule visualization

Data on the experiments was extracted from log files generated by the driving programs. With 440 queries in each experiment in as many as 15 concurrent threads, it was quite difficult to get a clear idea of how the teams were forming and terminating just by reading these files. With five runs per experiment, and roughly one thousand experiments in the investigation, it was impractical to read all of the log files even in a cursory fashion. There were, after all, over 5 GB of raw log files just from the

Copyright © 2004 John Wiley & Sons, Ltd.

experiments performed on the final machine configurations. Nevertheless, to verify that the scheduler was performing as expected and to examine the results for anomalies, some analysis of the log files was required. Judging a human to be better than any program at detecting anomalies, a way was needed to make the information in the log files accessible to the experimenter.

The logs were visualized by transforming them into HTML documents as shown in Figure 8, focusing on the schedules realized by each experiment. The output was a table with a number of columns:

- a running timestamp for events;
- an column for each team, with subcolumns for all its members side-by-side;
- an area to highlight events of interest: features and anomalies;
- an area to show the current contents of the queue.

In Figure 8, columns 2–6 show the schedule, consisting of five teams and their team members. Each vertical bar with text represents the execution of a single query, and contains repetitions of the query number and stream number, and ends with the duration of the query's execution. The color of these vertical bars indicated whether or not this cycle of the team comprised more than one query; red (here, very light gray) for multi-member team, blue (here, slightly darker than the background) for a singleton team. The fifth column shows that team 4 had six members at one point, and that the affinity rule has grouped queries of very different durations, only two of which seem to finish together. This disparity in durations can be seen in other teams as well. Care must be taken in interpreting these bars because the times are not to scale; each row represents an event, not a particular duration; nevertheless it may be that the chosen affinity groups are suboptimal, and the performance could be further improved with a better method of selecting the groups, or of forming the teams.

The column between the schedules and the queue contents is for annotation, which is done automatically. The annotations are of several kinds.

- Possible anomalies detected in the schedule, compared to what the analysis program thought the queues should have produced. The analysis program was reconstructing the queues, since the logs did not report them, and in each case it was determined that the actual schedule was reasonable.
- Possible autonomous team formation, as indicated by queries from different teams ending within 2 s of each other. This sometimes occurred between teams of similar constitution that were started not simultaneously, but within a few tens of seconds apart. It also occurred in experiments where the scheduler was not scheduling teams, validating the idea that teams form 'in the wild', but that this phenomenon had not been previously noticed.
- Scheduler limits being reached, when there were no processes available for scheduling a suitable teammate.

This arrangement on the screen was perhaps not elegant, and certainly required a very high-resolution screen to view properly, but it was of immense value in debugging, and in getting an intuitive feel for what was happening in the scheduler and the workload. It became evident, for instance, that the same workload would usually generate different schedules in each different experiment. Team formation turned out to be sensitive to the precise order of entries in the queue, since the first entry determines the nature of the next team to be formed. In turn, the order of items in the queue very quickly comes to depend on the order in which members of prior teams completed their work.

Copyright © 2004 John Wiley & Sons, Ltd.

Figure 8. Screen shot of the schedule Web page from one of the experiments. Vertical black bars delimit columns; column 1 has timestamps, columns 2–6 display the schedule, column 7 has notations, and column 8 displays the reconstructed queue contents.

Copyright © 2004 John Wiley & Sons, Ltd.

When teams worked well, several members would be finishing essentially simultaneously, and there was a race condition for which one would be placed on the queue next.

Another example of the value of this tool is that it detected substantial delays in scheduling a successor team after a prior team had finished. These were tracked down to a bug in the interface between the Expect toolkit and it's base language Tcl, and were fixed by the authors of those open-source packages on request.

Yet another example of the usefulness of the visualization is that early experiments were performed with 10 query streams, and the visualizations of these experiments made it clear that for many combinations of the test parameters team formation was being inhibited by lack of queries in the queue to consider as teammates when a team was started. Accordingly the results include additional experiments with 20 query streams and, although some teams still had only a single member, it was seldom because the queue was empty or nearly so.

This method of schedule visualization also had the virtue that it could be generated by a Perl script directly from the log files. The same script also generated an index and some other summary information, including HTML tables corresponding to some of the tables in this paper. These other pages were also very useful, particularly when the need arose to go back and locate the experiments that supported a particular result or table. The schedule HTML files were usually a bit over 1 MB in size, their verbosity being partly due to the need to recode the color attributes for each item of each row, and to insert filler when a column was blank. All of the tables in this paper were also generated by programs, directly from log files.

5.3. 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. This was tested in several ways, by forming groups of queries according to some alternate protocols.

5.3.1. Simple batching

The first approach attempted to promote team behavior by simple batching. Instead of forming teams on the basis of the template from which the queries derive, it 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 queuing strategy of the driver program.

The experiments used teams of varying sizes, either four or five, and with varying limits on the number of concurrent teams, from one team to the maximum allowed by the availability of queries. The resulting speed-ups are shown in Table VI, and are actually slowdowns from pure queuing (1.55). Accordingly, it appears that some information about the query is needed to effectively promote team formation.

5.3.2. Random affinity groups

The next approach attempted to randomize the affinity groups, called *random groups*. This is not a well-defined notion, so the experiments used three different protocols for forming these random groups.

Copyright © 2004 John Wiley & Sons, Ltd.

Table VI. Speed-ups from batching, comparing running time to the baseline at 20 database servers and 20 teams. Entries marked ⊥ are prohibited by the number of query streams.

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	⊥

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 the main results, for a total of 20 timing runs for each protocol. We report the mean result of the 20 experiments for each protocol.

Protocol A formed the same size affinity groups as in the earlier experiments, but used a random permutation of the query templates, producing an overall speed-up 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 Table V. This produced a speed-up 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 to one of the groups, or to be in no affinity group, with equal probability of 0.2 for each case, producing a speed-up 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}}.

Copyright © 2004 John Wiley & Sons, Ltd.

Table VII. The effects of the queue and of the CACHE attribute. The entries are speed-ups 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.

	LINEITEM CACHE			LINEITEM CACHE	
Processes	Off	On	Processes	Off	On
1	1.24	1.22	11	1.15	1.41
2	1.28	1.34	12	1.01	1.28
3	1.34	1.39	13	0.93	1.16
4	1.38	1.46	14	0.94	1.05
5	1.38	1.50	15	0.98	1.09
6	1.36	1.53	16	1.02	1.12
7	1.36	1.55	17	1.02	1.11
8	1.35	1.50	18	1.05	1.15
9	1.31	1.47	19	1.03	1.16
10	1.29	1.44	20	1.00	1.16

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 speed-ups for the three protocols were very similar, being A: 2.02, B: 1.97, and C: 2.06; these are intermediate between the speed-up for queuing alone (1.55) and the result for full teamwork scheduling (2.34).

5.4. Analysis of results

In order to obtain these results, the middleware has done more than introduce an affinity rule; it has 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 this approach. Two additional series of tests explores this question. In both series, the scheduler ran with the same number of processes as teams, effectively ensuring each team had a single member. The first series did not include 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 Table VII, and show that in the absence of team selection the CACHE attribute provides a slight advantage on this query workload. Without the CACHE attribute, the best speed-up is 1.38 at four and at five processes, and with the CACHE attribute this increases to 1.50 at seven processes.

A speed-up due to queuing is reasonable given that the hardware setup has seven primary resources that are used during the tests: four SCSI drives, an IDE drive (for the logs), and two CPUs. Scheduling more processes than seven may introduce more contention for resources than assistance with the computation. The speed-up 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.

Copyright © 2004 John Wiley & Sons, Ltd.

Based on these results, it seems that the queuing contributes a speed-up of up to 1.38, that caching the LINEITEM table contributes perhaps an additional speed-up of 1.09, and that, for this workload, the teamwork contributes a speed-up of at least 1.56, since $1.50 \times 1.09 \times 1.56 = 2.34$. Presumably, the queuing speed-up 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 the experiments reported here, the query template number is a proxy for this information, and is sufficient to produce the benefit seen on this workload.

6. WHEN THINGS GO WRONG

In order to determine if teamwork scheduling would have the same effects on other database engines, an effort was made to set up a similar computing environment using Microsoft SQLServer 2000 running under Windows 2000 Pro. Since the main Windows clients for SQLServer are all GUI-based, and although they have their own scripting languages, they are difficult to script from external programs. It would not be possible to use the same scripting tools that were used with Oracle. We accordingly decided to use a character-mode client from a separate machine running Linux by means of a Perl module capable of communicating with SQLServer [16]. This quickly became operational, and including a substitute for the Oracle SQL*Plus client, entirely in Perl, with just enough functionality to execute the scripts used with Oracle. The client and server machines were connected to each other through a switch with category-5 cable using 100 Mbit ethernet. The existing scripts needed very little modification other than changing the query texts to accommodate differences in SQL dialects. This seemed promising, but a number of issues arose, as described in the following sections.

6.1. Controlling the RAM footprint

Recall that the Oracle tests had used raw partitions to bypass the Linux buffer pool, and had severely restricted the size of the Oracle buffer pool so as to create a low ratio of buffer pool to database size, since this is typical of large database servers. While raw partitions can be used with SQLServer, there is no user-available control of the buffer pool size; one can usually only control the size of the machine's RAM, or the amount of RAM that SQLServer is allowed to use for all purposes. With Linux, one can instruct the kernel to use a specific amount of RAM, and it will do so without probing memory; this is not possible in Windows 2000 Pro, and the test machines require a specific memory type (Registered ECC DRAM DIMMs) that were not readily available in smaller sizes. The first attempt to address this tried to sequester some of the RAM by installing a Ramdisk, using as a starting point a driver source code provided by Microsoft [17]. The driver was modified so that it could be configured to request up to 2 GB of non-paged RAM, but this failed when Windows 2000 Pro would not honor requests for more than about 120 MB. This limit was found to hold essentially the same on machines with 256 MB RAM and our test machines with 1 GB of RAM; moreover, it held in the aggregate when using two independent Ramdisk drivers.

Since it appeared that the only way to control the memory available through Windows 2000 Pro was to run on a smaller machine, the experiments were tried on a completely different set of machines with

Copyright © 2004 John Wiley & Sons, Ltd.

256 MB RAM. Initial tests of the database functionality went well, but attempts to run the query mixes ran into difficulties. When ramdisk drivers were used to reduce the RAM from 256 MB down to about 150 MBs some queries would begin to time out after 6 or 7 minutes, with a message that indicated the query had reached a limit of how long it was allowed to wait for memory. Evidently, the restrictions on memory had created a deadlock on memory resources. Running without the ramdisks and with limited Windows swap files resulted in another series of messages about the inability of Windows to properly execute the workload in this environment. Since swapping had not occurred in our Oracle tests, it did not seem wise to allow it for SQLServer because it would be likely to congest the resource being tested the most: disk transfer efficiency. Even removing the ramdisk, and running with the machine's full 256 MB of RAM did not support the tests with 20 concurrent servers. This configuration did support 10 concurrent servers, and tests modified for that environment showed a speed-up of 1.07 for queueing, an additional 1.07 for teamwork, for a combined speed-up of 1.15.

It seemed that attempting to mimic a large database by adjusting memory usage was not going to work well in a Windows environment, and suggested a more direct approach of scaling the database to roughly 10 GB, so that the natural buffering available in a 1 GB RAM would have roughly the same proportion as had been obtained for the Oracle tests. With the number of query streams left at 20, perhaps this arrangement would have adequate resources for running the query streams, and support the tests. Since testing might still take 10 times as long, experiments were planned only for the key values that had produced the results claimed for Oracle. Settings for SQLServer were changed to a 'simple' recovery strategy in which transaction logs are not retained, were enlarged as partitions required for the larger database, and the schema tables were allocated as nearly as possible in the same way as had been done for Oracle. Database loading in SQLServer is logged, so that while the logs are not retained, log files are kept during each operation; this required allocating large log files and performing the loading in segments, truncating the log files between segments.

6.2. Teamwork in a hostile environment

This turned out to be a hostile environment for testing, and is reported here because it gives a picture of at least one way in which a system can be hostile to query teamwork. Because there did not seem to be a clear rationale for doing otherwise, the remainder of the tests on SQLServer ran with default settings other than the 'simple' recovery strategy and placement of files.

6.2.1. Feasibility tests

A re-run of the tests that had produced Table II tested whether this configuration also promoted the formation of teams. The results are shown in Table VIII, and display some unexpected features. The tests for Oracle had shown a tendency to form a single team of all five queries; the tests for SQLServer showed a tendency to form teams of two or three queries, limiting the potential benefits of team formation. This is most clearly seen in the first line of the figure; five queries instantiated from the same template are all started at once (zero separation), but they finish in two groups, with a separation equal to roughly the time a single instance of the query takes to finish when run alone. Team formation clearly occurs, in that there are three queries finishing in about the time a single query would take, but, at least for this query, large teams do not seem to form. At larger initial separations, the teams tend to have just two members, or to be indistinct.

Copyright © 2004 John Wiley & Sons, Ltd.

Table VIII. Test of teaming behavior on SQLServer 2000. The same query was used as in Table II, but the hardware was much faster. A single instance of the query takes 229 s on this hardware.

Initial separation	Stream 1 time	Inter-stream ending separations				Total time
		1–2	2–3	3–4	4–5	
0	237	0	0	245	0	482
10	315	0	82	0	157	554
20	295	0	101	0	141	537
30	291	0	252	13	87	643
40	270	304	7	31	78	690
50	290	213	79	78	0	660
60	236	198	164	0	117	715
70	298	242	103	114	34	791
80	305	274	78	110	10	777

In order to explore this phenomenon further, an additional experiment ran tests of 20 copies of identical instantiations of single query templates, in order to explore the differences in team formation between Oracle and SQLServer, as shown in Figure 9. Three queries are shown; one is the same as the query of Table VIII and the other two are respectively a long-running query and a short-running query. In the SQLServer panels on the left of the figure, the results show that for Query A teams definitely form, and perform well, so that the overall running time represents a speed-up of 3.53, which is somewhat better than the 2.13 achieved by Oracle. However, the queries do not form a single team, but appear to form four teams. This segregation into separate teams appeared in most, but not all, of the queries in SQLServer; it is not apparent in Query C, which may be related to the fact that this query runs very fast. One can see that Oracle did poorly with the fast query, but because it is fast this does not affect the overall performance much. SQLServer, however, did poorly with the long-running Query B which suggests poor performance will result; identical instances interfere and one can only expect worse behavior in a heterogeneous mix.

Even with 1 GB of main memory, tests with 20 concurrent processes continued to fail, and had to be replaced by tests with 10 concurrent processes. These produced substantially the same speed-ups as seen on the machine with 256 MB of RAM. In the light of the unpromising results seen in Figure 9, and the large amount of time projected for the effort, this effort stopped without tests to form new affinity groups, or to compute a full matrix of results such as seen in Table V.

It is natural in the face of such differences in results on the two database systems to inquire as to the reasons. This is difficult to determine because they are both closed-source commercial products, and the precise details of their internal workings are presumably trade secrets. Nevertheless, it is at least possible to point out some of the external differences between the systems and to present some possible explanations.

There is an evident difference in product philosophy between the two products. The Oracle product provides the system administrator with a number of configuration options, most of them embodied in

Copyright © 2004 John Wiley & Sons, Ltd.

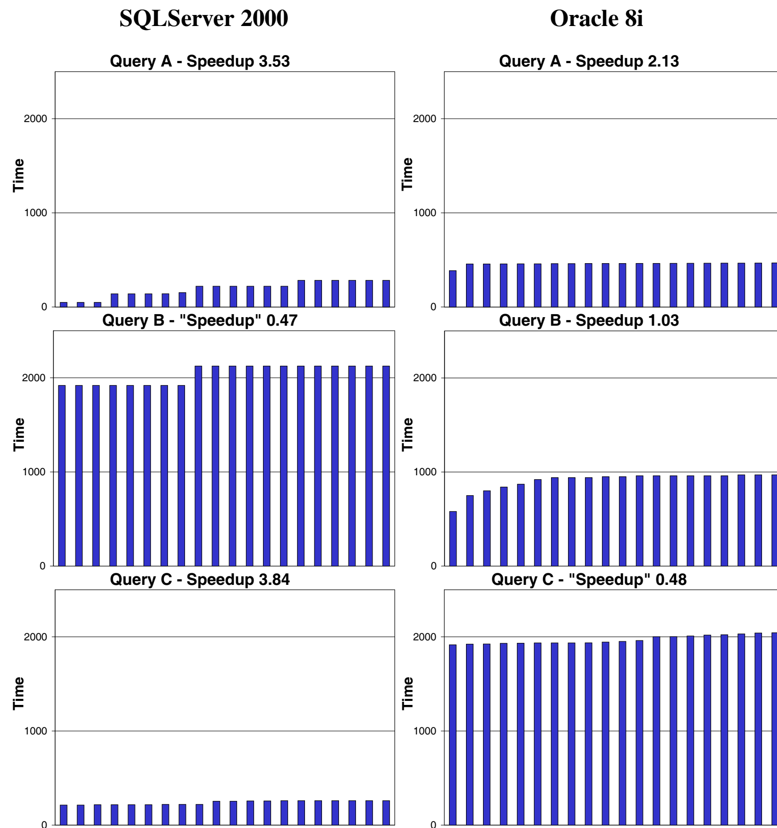

Figure 9. Team formation comparison. The running times of each of 20 identical instances of a single query are shown. Query A is the query of Table VIII; Query B is a long-running query in SQLServer; Query C is very short. Running times are normalized so that if the last query of each panel finishes at 1000, the speed-up is 1.00.

the so-called initORA.conf file, where one sees 15 or so numeric and a few Boolean configuration variables. As has been remarked, the Oracle experiments used two of these to control the size of the Oracle buffer pool and the sort area size. In contrast, while SQLServer provides a convenient GUI-based Enterprise Manager, there is not the degree of fine control over components that is found in Oracle. Moreover, it appears that SQLServer manages the associated resources actively, as compared to the static allocations of Oracle, at least with respect to resources that are user-configurable in Oracle. This may bear on the differences that were encountered.

6.2.2. A conjecture

One particular control that occurs in Oracle that attracts attention in this regard is the sort area size, which was increased by a factor of 100 from the default setting to about 6.5 MB, which seemed

Copyright © 2004 John Wiley & Sons, Ltd.

reasonable for up to 20 concurrent processes in a node with 1 GB of RAM. Early experiments with Oracle had encountered what seemed like anomalous team formation until this change was made. No corresponding control is available to the system administrator in SQLServer, and it is tempting to think that there may be a connection with the team formation anomalies encountered during testing. Presumably the size of sort areas in SQLServer is managed by the database engine, presumably in a dynamic and adaptive fashion; it may be that the choices made in this way are good for the expected computation but create a situation in which team formation is inhibited. The sort area comes into play in all of the TPC query templates, because they all have an ORDER BY clause, a GROUP BY clause, or both, and these are implemented by sorting the intermediate results.

If the sort area is very large, as might happen when the material to be sorted is projected to be large, it might not be possible to have 20 or perhaps even five of the sort areas active simultaneously. In such a case, one might see smaller groups being allowed to proceed, such as is observed in Table VIII and Figure 9, possibly as the result of a deadlock-avoidance policy related to memory resources. While the results here cannot resolve the issue, it would be consistent with the excellent team formation for the short-running query, which presumably does not require large memory resources. In the absence of teamwork, such a memory management policy might be an optimum choice for overall throughput. The determination of the exact causes of the limits on team formation are beyond the scope of the current work, as is any comparison of the merits of the product strategies of commercial products. The difficulties and the symptoms are presented as an example of the characteristics of friendly and hostile environments, respectively.

7. DISCUSSION

Traditionally, 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, an algorithm is proposed as an alternative approach to multi-query optimization that obviates the need for such modification, at least in suitable databases. However, if it should be necessary or convenient to make such changes, it seems that the benefits would be at least as great as when implemented as middleware, and might apply to databases that are hostile to the middleware approach. The experiments showed that significant speed-ups were available from middleware by substituting cooperation for contention through scheduling compatible queries together. Such improvement is important in itself inasmuch as some of the TPC-R [12] queries can run for tens of minutes each on the test equipment used. A descriptive model and a formal model are presented for query behavior that leads to cooperating query teams and the behavior is measured under a benchmark workload that represents a variety of business queries. A scheduling algorithm is proposed to promote team formation, with little *ad-hoc* information about the query workload. The examples show improvements with an affinity rule that generates heterogeneous teams. The results show that substantial throughput improvements are achievable with a lightweight scheduler operating as a middleware between the clients and an unmodified commercial database. The experiments with alternative algorithms in Section 5.3 suggest that team formation can be promoted in a variety of ways, and suggest 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

Copyright © 2004 John Wiley & Sons, Ltd.

to support the selection of candidate teams. Future work will explore methods for dynamically determining team formation.

ACKNOWLEDGEMENTS

This work was performed while Dr O'Gorman was at the University of California, Santa Barbara.

REFERENCES

1. Jarke M. Common subexpression isolation in multiple query optimization. *Query Processing in Database Systems*, Kim W, Reiner DS, Batory DS (eds.). Springer: Berlin, 1985.
2. Park J, Segev A. Using common subexpressions to optimize multiple queries. *Proceedings of the 4th International Conference on Data Engineering*. IEEE Computer Society: Washington, DC, 1988; 311–319.
3. Sellis T. Multiple query optimization. *ACM Transactions on Database Systems* 1988; **13**(1):23–52.
4. Cosar A, Lim E, Srivastava J. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. *CIKM 93, Proceedings of the Second International Conference on Information and Knowledge Management*. ACM, 1993; 433–438.
5. Chen F, Dunham M. Common subexpression processing in multiple-query processing. *IEEE Transactions on Knowledge and Data Engineering* 1988; **10**(3):493–499.
6. Roy P *et al.* Efficient and extensible algorithms for multi query optimization. *Technical Report*, Indian Institute of Technology, Bombay, India, October/November 1998.
7. Roy P *et al.* Efficient and extensible algorithms for multi query optimization. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, 2000; 249–260.
8. Tan K, Lu H. Workload scheduling for multiple query processing. *Information Processing Letters* 1995; **55**(5):251–257.
9. Tan K, Lu H. Scheduling multiple queries in symmetric multiprocessors. *Information Sciences* 1996; **95**(1/2):125–153.
10. Zhao Y *et al.* Simultaneous optimization and evaluation of multiple dimensional queries. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data. SIGMOD Record* 1998; **27**(2):271–282.
11. Dalvi N *et al.* Pipelining in multi-query optimization. *J. Comput. Syst. Sci.* 2003; **66**(4):728–762.
12. Transaction Processing Performance Council (TPC). *TPC Benchmark R Standard Specification Version 1.4.0.* http://www.tcp.org/rspec.html [25 April 2002]. Used by permission of TPC. TPC Copyright Notice: TPC Benchmark is a trademark of the TPC. All parties are granted permission to copy and distribute to any party without fee all or part of public TPC copyrighted material provided that: (1) copying and distribution is done for the primary purpose of disseminating TPC material; (2) the TPC copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Transaction Processing Performance Council.
13. O'Gorman K, Agrawal D, El Abbadi A. On the importance of tuning in incremental view maintenance: An experience case study. *Data Warehousing and Knowledge Discovery Second International Conference (DaWaK)*. Springer: Berlin, 2000; 77–82.
14. Libes D. *Exploring Expect*. O'Reilly & Associates, 1995.
15. Graefe G. The value of merge-join and hash-join in SQLServer. *Proceedings of the 25th International Conference on Very Large Databases*. Morgan Kaufmann: San Francisco, CA, 1999; 250–253.
16. Trice A. Connect to Microsoft SQL 2000 with the Perl Sybase Module. *Linux Journal* 2002; April: 62–65.
17. Microsoft Corporation. RAMDISK.SYS sample driver for Windows 2000 (Q257405). http://support.microsoft.com/support/kb/articles/q257/4/05.asp [August 2002].

Copyright © 2004 John Wiley & Sons, Ltd.