Memory Aware Query Routing in Interactive Web-based Information Systems

Florian Waas[1,2] Martin L. Kersten[2]

[1]Microsoft Corp., One Microsoft Way, Redmond, WA 98052, USA, *flw@mx4.org*
[2]CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, *mk@cwi.nl*

Abstract Query throughput is one of the primary optimization goals in interactive web-based information systems in order to achieve the performance necessary to serve large user communities. Queries in this application domain differ significantly from those in traditional database applications: they are of lower complexity and almost exclusively read-only. The architecture we propose here is specifically tailored to take advantage of the query characteristics. It is based on a large parallel shared-nothing database cluster where each node runs a separate server with a fully replicated copy of the database. A query is assigned and entirely executed on one single node avoiding network contention or synchronization effects. However, the actual key to enhanced throughput is a resource efficient scheduling of the arriving queries. We develop a simple and robust scheduling scheme that takes the currently memory resident data at each server into account and trades off memory re-use and execution time, reordering queries as necessary.
Our experimental evaluation demonstrates the effectiveness when scaling the system beyond hundreds of nodes showing super-linear speedup.

1 Introduction

A significant number of web-based information systems rely on database technology to serve large user communities which makes scalability a key issue for the design of web-enabled database systems. Parallel processing and data replication, are necessary to deal with the peak loads encountered. Likewise, an effective query dispatching scheme is needed to level the system load as well as to guarantee quality-of-service in terms of response time.

In this paper we are concerned with initial experiences with a multi-media portal under construction based on the Monet database system [BK99]. The system is intended to provide efficient access to a large collection of indexed multi-media objects. It is endemic to this kind of information system that user interaction is dominated by read accesses. A number of systems with similar requirements regarding the deployed database backend have been developed and many more are currently under construction.

With each user interaction, the interface emits a number of queries to the database that ideally lead to an answer set of a few tens of candidate results. Involving accesses to different multi-dimensional indexes, the evaluation of such queries is usually in the order of few seconds. Still, the queries are of distinctly low complexity compared

to queries in classical database applications. Moreover, the deviation of running time among the queries is limited, not least to ensure acceptable response times.

The primary challenge in this setting is to develop processing techniques to optimize the query throughput. Parallel processing is an essential element to achieve this, however, a straight forward recasting of methods developed for parallel databases does not apply here since most solutions devised in this area are almost exclusively geared to tackle highly complex and long running queries. There, queries are usually parallelized on a granularity of partial plans or even single operators, i.e. single operators like the join of two tables are executed in parallel on different nodes involving exchange of partial results among the single nodes. However, these techniques are ineffective for the kind of query we are considering since communication and coordination overheads would outweigh the actual benefits.

In this paper, we propose a parallel query processing architecture that can take advantage of the query characteristic by its physical design, suitable query scheduling, and the way queries are executed.

The platform of operation is a shared-nothing environment—i.e. a cluster of inexpensive PCs—where each node runs a Monet server with a fully replicated copy of the database. One machine is distinguished as coordinator node that dispatches the arriving queries to the servers according to a scheduling strategy. The scheduling schema we develop in this paper differs radically from previous work as we do not try to model various system parameters in order to exploit primarily idle system resources, but take into account what data is memory-resident at the servers, i.e. cached by the servers. The algorithm is based on a metric that determines the *distance* between a server and a query—the less this distance, the more similar the state of the memory at this server and what is required to process this query. Moreover, we investigate possibilities of re-ordering and deferred execution of queries to further reduce execution costs. Once a query is assigned to a server it is executed in isolation on this server, so no synchronization or communication within the cluster is needed freeing interconnection bandwidth for shipping of both queries and results.

Since we are dealing with read-only accesses, we do not have to consider transaction mechanisms to keep the replicated data consistent across the database cluster. Rather, the databases are periodically updated by mirroring a master database.

The experimental evaluation of the techniques proposed show substantial savings over conventional greedy scheduling that takes only the machines' workload into account. In a large number of experiments we investigate the impact of individual parameters closely. Our results confirm the architectural decisions showing excellent scaling behavior.

The remainder of this paper is organized as follows. We review related work in Section 2. In Section 3 we present the architecture and describe the query model in Section 4. Section 5 discusses the modeling of the server pool and the scheduling algorithm. In Section 6, we present a comprehensive performance analysis. Section 7 contains a discussion of the design decisions. We conclude the paper with Section 8.

2 Related Work

Parallel query processing has been studied in a large variety of facets, see e.g. [PMC+90,DG92,HS93,WFA95,Gra95]. Most of related work in this field concentrated on possibilities to speedup highly complex queries with long running times. Approaches as taken in [HM94] and [GI96] suggest a decomposition of the query plans into sub-plans which are then executed in parallel on different nodes of the parallel processing environment. The granularity of this decomposition varies and can be as fine as parallelizing single operator as studied for example in [SD89,SD90,WFA95] but is often chosen coarser [HM95,CHM95,GI96,GI97].

These approaches have in common that they require communication between single nodes for shipping or exchanging partial results. This causes network contention and synchronization effects where nodes have to wait for others to complete their tasks first. As a result, a parallelization along these lines only pays if the query is of sufficiently high complexity. Otherwise communication overhead and synchronization effects outweigh performance gains.

Moreover, parallel processing as outlined above scales only for small numbers of nodes effectively. In shared-nothing architectures, network contention becomes increasingly a bottleneck; in the case of shared-everything, the high degree of resource sharing limits the scaling [Sto86,NZT96].

What makes most of these approaches, however, questionable is the fact that during the optimization a number of highly sensitive parameter—which are hard to model mathematically and impossible to maintain accurately during running time—have to be taken into account. Parameters like network load, page faults etc. are assumed constant during the query execution and estimate errors, which may have severe impact on the performance, can not be corrected (see e.g. [GI97]).

General memory allocation issues have been explored extensively with respect to various aspects of query processing. In [MD93], authors proposed dynamic memory allocation schemes for multi-query workload to level memory allocation without sharing resident data among different queries. By analyzing queries and their common sub-expressions the re-use of memory resident data is often a by-effect [SSN94]. In [MSD93], authors consider batch scheduling for parallel processing. However, main-memory is in both cases transparently viewed as a central resource and data location within distributed memory has not been considered.

In the context of transaction processing, several query routing schemas for database clusters have been investigated (see e.g. [Tho87,FGND93,RBS00]). This field of application differs from the problem at hand in its preliminaries: queries are usually of complex nature and updates to the database need to be propagated over the complete cluster. One of the major goals is for instance to schedule queries so that locking conflicts are avoided. For a comprehensive overview on this subject see e.g. [Rah92].

3 Architecture

Figure 1 shows the architecture of our system. It consists of a cluster of database servers managed by a central scheduler node. All machines have separate main-memory, CPU,

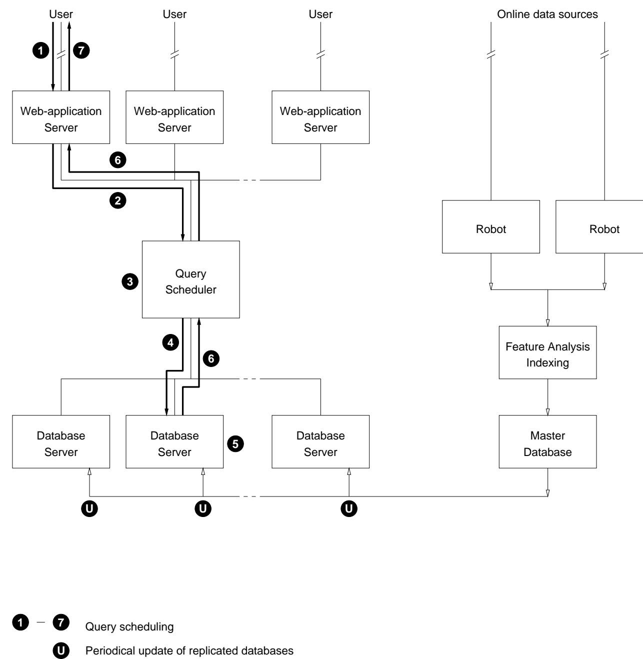

Figure 1. Query processing architecture

and disks not sharing any resources other than network bandwidth. We use Monet, the main-memory database system developed at CWI, as database server [BK95,BK99]. Besides its vertical fragmented data model, Monet is distinguished by its memory awareness, i.e. it solely uses operating system primitives for its memory management, mapping database tables directly to virtual memory, to avoid the overhead of a proprietary buffer manager simulating a virtual memory layer. Moreover it provides specifically cache aware operator implementations [BMK99] to maximize system utilization on running queries.

Additionally, a pool of web-servers forms the front-end of the system which clients interact with. The web-servers receive either parameterized queries using text-based forms, or they interact with visual query formulation tools.

Processing of a client query is done in 7 steps: Users formulate their query using the web-interface (Fig. 1,(1)). At the web-server, the query is re-formulated using the internal procedural query representation of the database system—in our case MIL, the query language of Monet—and submitted to the query scheduler (2). The scheduler maintains a queue of queries that are to be executed. By analyzing the data requirements for the execution of a query and the data resident in main-memory at the servers, the scheduler determines a favorable assignment of queries to servers (4). The query is

executed on the assigned node (5) and the result is returned to the front-end host (6).[1] After formatting the result, it is shipped to the user (7).

This architecture directly aims at the throughput optimization of compact queries where an execution on a single node is the most efficient kind of processing. But this architecture also exploits the second query characteristic: the impreciseness of the data. For example, data gathered by robots from the Internet to build up a multi-media index is updated only at low frequency. Thus, the server cluster do not have to be kept in sync as would be the case if updates by users were allowed. Instead, the databases are updated periodically replicating the master database (U). The frequency of updates depends naturally on the application domain.

4 Query Model

The choice of Monet as database software for the back-end cluster implies a specific model for the query execution. The vertical fragmentation of the tables in Monet causes bulk-processing to be more efficient than pipelining techniques. As a consequence, there are no more than two tables processed on a single CPU at a time. Note, this execution model does not impose any restrictions on the shape of the execution plans—both linear and bushy plans are feasible. Access to a table can be either of type LOAD or SCAN. LOAD reads a table from disk into memory to make it part of the hot set, e.g. used with inner relations of nested-loop joins. SCAN reads the table, but does not keep the data in memory after the actual operation is performed, e.g. used in selections or for the probe relation of a hash join. The building of a hash table can be seen as special type of load as the result is only accessible via the hash attribute.

The costs for executing a relation algebra operator consists of the costs of loading/scanning of the table plus the actual costs for the operation. Fitting the pieces together, we can describe a query as a sequence

$$Q = \langle (t_1, a_1, l_1, w_1), \ldots, (t_n, a_n, l_n, w_n) \rangle$$

where each quadruple corresponds to an operator or—as in case of a join—to a partial operator. t_i specifies the table necessary for the operation. a_i denotes the hash attribute, i.e. the operator accesses t_i via this attribute; $a_i = \epsilon$ if not applicable.

The associated *loading costs*, if the table is not already resident in main-memory, are given by l_i. We denote the *total loading costs* of a query as

$$L(Q) = \sum_i l_i.$$

[1] For simplicity of presentation arrows (6) follow the routing of the query (2–4), however, the query result can be sent directly to the front-end and does not have to pass the query scheduler.

The costs to execute the operator, the *operator costs*, are denoted by w_i. We denote the *total operator costs* of a query as

$$W(Q) = \sum_i w_i.$$

Note, both l_i and w_i are expressed in terms of the same unit to achieve a proper comparison.

A queries execution time on a "cold" server, i.e. no data is loaded yet, is

$$\chi(Q) = L(Q) + W(Q).$$

If all tables needed by Q are already in memory—and in case of hash tables, are hashed by the proper attribute—the execution costs of the query amounts to $W(Q)$ only.

Examples

Here are some examples to illustrate this modeling based on Monet performance characteristics. The two tables A and B used in this example are of size 8.8MB and 4.95MB, respectively. Our test platform achieved a bandwidth of 5.5MB/s for disk access.

– Nested-Loop Join, $A \bowtie B$

$$Q = \langle (A, \epsilon, 1.6, 0), (B, \epsilon, 0.9, 2.6) \rangle$$

Table A takes 1.6s to load, no operator costs occur. Table B takes 0.9s to load; performing the join requires 2.6s.

– Hash Join, $A \bowtie B$

$$Q = \langle (A, a, 2.7, 0), (B, \epsilon, 0.9, 1.1) \rangle$$

Similar to previous but now, A must be a hash table with hash attribute a. Hence loading A is more expensive as it includes building the hash table.

5 Query Scheduling

The query scheduling comprises several elements. Besides a model for the servers we define a *server-query distance* which captures the potential re-use of memory resident data. Additionally, we also introduce deadlines.

5.1 Servers

For the scheduling, a server of the database cluster is modeled by its state of memory and the workload. The state of memory is the set of tables resident together with a replacement strategy. We are considering base tables only and discard intermediate results of the processing as soon as they are no longer used. As a replacement strategy we use LRU as it exhibits the best average performance. The loading and dropping of

tables is done via the memory mapping functionality of the operating system. To maintain sufficient control over the memory allocation throughout the complete cluster we load and drop only complete tables. This way, the scheduler can rely on the information which tables are memory-resident, i.e. accessing them will not cause additional costs for swapping. Swapping may only occur when all unused tables are already dropped but the memory requirements of the current operation are still not met.

For the workload, we distinguish the two states *idle* and *busy*, i.e. we assign one query to one server at a time. This is not just a simplification to facilitate the scheduling but a necessity in main-memory databases where cache awareness and concurrent memory access are of distinctly higher importance than in I/O-dominated database models [MBK00]. We model the workload as function $J(S)$ which returns the expected time of job completion at server S, given its current workload, i.e. the expected time from now when S will become idle. If the server is idle, $J(S)$ evaluates to 0. J will be used in the scheduler to find the node that will finish its job next. J is computed by conventional cost formulae known from sequential query processing: Given the time x_0 the currently running query has been assigned to server S, x the time J is evaluated, and e the expected running time of the query, the time of job completion computes to $J(S) = x_0 + e - x$. See also Section 7 for a discussion on the accuracy of J.

5.2 Distance Metric

We define the *server-query distance* as the costs to load the tables for a given query Q on a server S:

$$d(Q, S) = \sum_i R(t_i, a_i) \cdot l_i$$

where

$$R(t_i, a_i) = \begin{cases} 0, & \text{if } t_i \text{ is memory-resident at } S \text{ and hashed by attribute } a_i \\ 1, & \text{else} \end{cases}$$

indicates whether table t_i is resident in memory at S. In case t_i is required as hash table, R also checks whether the table is hashed by attribute a_i.

Scheduling a batch of queries optimally on k servers is finding a division into k batches $B_1, \ldots, B_k$, each of which are executed sequentially on one server, such that the running time of the batch with the longest completion time

$$\max_i \left\{ \sum_j \big(d(Q_j, S) + W(Q_j) \big), \quad Q_j \in B_i \right\}$$

is minimal.

```
Algorithm MAS

while queue not empty do
    c_min ← ∞
    foreach query Q in top_n(queue) do
        for i = 1 to number of servers do
            c = d(Q, S_i) + W(Q) + J(S_i)
            if c < c_min do
                Q̂ ← Q
                Ŝ ← S_i
                c_min ← c
            done
        done
        if expired(Q) then break
    done
    assign query Q̂ to server Ŝ
    remove Q̂ from queue
done
```

Figure 2. Scheduling Algorithm MAS

5.3 Scheduling Algorithm

We use the distance measure to develop a greedy scheduling algorithm that establishes an acceptable trade-off between workload- and memory-focused scheduling. Figure 2 shows an outline of our algorithm called *Memory Aware Scheduling* (MAS). It iterates over the queue of arriving queries, selecting one at a time, and determines the best ad-hoc assignment.

In detail, we examine the first n elements of the queue—or less if the queue does not contain n queries. We investigate the impact of n and suitable values for it in the next section. For each of the n queries, we compute c which consists of the distance to all servers S_i plus the operator cost of the query and the expected time at which server S_i becomes available, $J(S_i)$. We record the pair $(\widehat{Q}, \widehat{S})$ with the lowest value for c. After examination of all n queries, we assign $\widehat{Q}$ to $\widehat{S}$ which means that $\widehat{Q}$ will be executed on $\widehat{S}$ as soon as this server gets idle.

The algorithm is in $O(N \cdot n \cdot s)$ where N is the number of queries in total, n is the number of queries considered in each run, and s is the number of servers available, i.e. the algorithm is linear in the number of queries. To give any meaningful bounds on the performance is particularly difficult because of the LRU replacement of tables.

Size	$\leq$ 20MB	$\leq$ 30MB	$\leq$ 40MB	$\leq$ 50MB	$>$ 50MB
#Tables	233	87	24	15	17

Table 1. Sizes and numbers of base tables

5.4 Deadlines

In order to give the user a guarantee of service, we tag every query with a deadline. This deadline refers to the latest point in time the query has to be assigned to a server for execution, i.e. as soon as the deadline of a query expires, the scheduler has no other choice than assigning this very query to a server.

We tag all queries with a time stamp according to their arrival. In other words queries are not forced by deadlines to overtake others, though it is often beneficial. As a result, we need only check the first query of the current top n batch for deadline expiration. If the first query's deadline is expired, we do not need to examine any other query in the batch but have to assign the first immediately to a server. Otherwise, if the first query's deadline is *not yet* expired, no other deadline can be due. Testing for expiration after the first query has been checked against all servers ensures the best assignment in case the deadline expired.

6 Experimental Results

In this section, we describe experimental results obtained with a simulator. We chose to simulate the system in order to experiment with parameters that are strictly limited by an actual hardware configuration, such as size of the database cluster or main-memory available at the individual servers.

6.1 Preliminaries

Since the full multi-media Acoi demonstrator database is still under construction, we had to confine the experiments to the part already operational. The part chosen is the index system of the ACM Anthology, which is included in the demonstrator to assess its capabilities in the area of XML-based database processing. We used statistical data available from an actual Monet database instance which contains the complete XML code of the Anthology decomposed into the vertically fragmented data model [SKWW00]. Including indexes, the database contains 376 tables with up to nearly 60000 rows. The numbers of tables according to their sizes are given in Table 1. Typical queries use some 5 tables or less, seldom up to 10 or more. We generated batches of 10,000 and 100,000 queries of exponentially distributed sizes accessing 5 tables on average.

We do not consider mechanisms to reject user queries due to overload since this can be done already at the front-end level. Modeling arrival rates probabilistically is not necessary as queries do not significantly differ in running time, thus information about

the queries does not need to be considered to make a choice which queries to accept and which to reject. For our experiments, we assume the maximal expected query arrival rate to model the worst case behavior with maximum load. Moreover, we assume all hosts within the database cluster are of identical configuration regarding the cost relevant parameters.

Since the replacement of tables along an LRU strategy makes analytical modeling of the algorithm hard, we implemented *Graham's list scheduling* (GLS), which is based on workload figures [Gra69], for comparison. We chose Graham's algorithm instead of seemingly more advanced techniques, like [GI96], as it is the only algorithm that does not make assumptions concerning parameters like network load etc., which are impossible to maintain accurately at reasonable costs (see also Sections 2 and 7).

We adapted the original algorithm to fit the online arrival of queries, i.e. to use the kind of look-ahead we introduced in MAS above. Graham's algorithm is known to be highly effective despite its simplicity overcoming some characteristic disturbances also known as *scheduling anomalies*. These anomalies occur with jobs that differ significantly in completion time. Please note, we run Graham's algorithm in exactly the same setting, i.e. with LRU replacement of tables as necessary. As a result, Graham's algorithm also profits from re-use of memory resident data.

The main parameters we want to investigate are, foremost, the number of servers, the look-ahead during the scheduling, and the amount of memory available at each server. Note, there are two principal ways of comparison: a one-on-one comparison of both algorithms run on identic configurations of the platform (relative performance), or a comparison of the scaling characteristics. Typical examples for the latter are speedup and scale-up. We will compare the two algorithms using both principles where adequate.

6.2 Warm/Cold Processing

In this first experiment we investigate the differences between warm and cold servers. We refer to a server as *cold* if no other tables than system tables have been loaded. If more than 80% of the available memory are allocated we call a server *warm*.[2] This is not only relevant for later experiments but also for the real application scenario when the system has to be shutdown and re-started for technical reasons like maintenance etc. We can expect different effects if the amount of memory per server is varied. For small server configurations the warm up is completed earlier than for large ones. However, more important is the amount of memory in total, i.e. the number of servers. Figure 3 shows the relative execution times of batches of 10,000 queries as function of the number servers. For example the leftmost data point for MAS reflects the ratio of execution times for MAS on a cold to that on a warm server. The left graph shows times for a server configuration of 64, the right for 256MB.

For GLS, cold and warm execution times are almost the same. Warm processing is only up to 5% quicker. For MAS the ratio differs significantly showing gains of more

[2] We determined the value 80 in preliminary experiments. Execution times on servers with more than 80% allocated memory did not differ significantly.

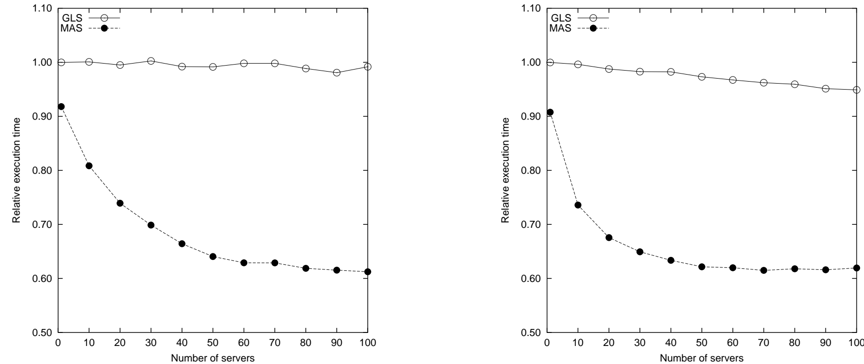

Figure 3. Relative performance of warm over cold servers

than 35%. Especially for the range up to 50 servers, the larger configuration (right) achieves better performance, i.e. the curve is steeper. For more servers, the impact of the larger amount of memory decreases: for 100 servers and more, results are virtually equal. We address the issue of memory sizes in more detail in 6.5.

All further results presented in this section are obtained from warm servers.

6.3 Reordering of Queries

The next experiment investigates the impact of the look-ahead during scheduling, i.e. the maximal number of queries that may be re-ordered between each assignment.

Figure 4 shows execution times for a single server (left) and a cluster of 100 servers (right). In both cases the server(s) had 64MB of memory. The execution time is shown as a function of the look-ahead, scaled to GLS's first data point, which corresponds to a first-come-first-serve (FCFS) scheduling. The size of the complete query batch was 10,000. In the case of one server, the look-ahead is of high importance for MAS and savings can amount up to 20% of the execution time. GLS, however, does not significantly profit from look-ahead.

In the case of 100 servers, the situation changes. GLS relatively improves more with increasing look-ahead but for MAS hardly any improvement is noticeable, though its execution time is substantially below the one of GLS. This is due to the fact that in a pool of 100 servers many server offer a very low server-query distance and differences are often very small. As a consequence re-ordering cannot help finding a significantly lower server-query distance as it did in the sequential case.

In all further experiments we use a look-ahead of 100 queries unless stated otherwise.

6.4 Speedup and Scale-up

The two fundamental measures when investigating the performance of parallel systems are speedup and scale-up. The previous quantifies the gains when scaling up the plat-

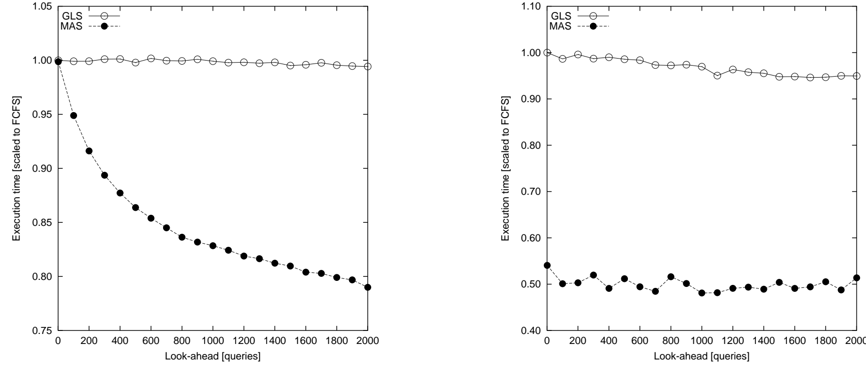

Figure 4. Impact of look-ahead; Single server (left) and cluster of 100 servers (right)

form but keeping the problem size constant, the latter describes the system's ability to cope with problem sizes growing proportionally with the platform (cf. e.g. [DG92]).

Figure 5 shows the speedup for a query batch of 100,000 queries evaluated on up to 4096 servers with 64MB memory each. As the plot shows, GLS achieves slightly sub-linear speedup whereas MAS achieves even super-linear speedup which translates to effective re-use of memory resident data. To better illustrate this phenomenon, consider a very basic example using 4 tables A,B,C and D of same size such that only two table fit into memory at the same time, and a batch of 4 queries:

$$Q_1 = \langle (A, 2, 2), (B, 2, 2) \rangle$$
$$Q_2 = \langle (C, 2, 2), (D, 2, 2) \rangle$$
$$Q_3 = \langle (D, 2, 2) \rangle$$
$$Q_4 = \langle (A, 2, 2) \rangle$$

On a single machine, the total costs amount to $8 + 8 + 4 + 4 = 24$. With linear speedup, we would expect 12 cost units for a parallelization on two hosts. MAS assigns queries Q_1 and Q_4 to one, Q_2 and Q_3 to the other machine which amounts to $8 + 2 = 10$ costs at each node. Hence, the total execution time is 10 compared to 24 on a single node which gives a speedup of 2.4.

Figure 6 shows the scale-up for the same server pool configuration. MAS maintains a scale-up of about 1.1 even for large configurations whereas GLS drops quickly to about 0.8.

6.5 Direct Comparison

In our last experiment, we compare the algorithms directly, i.e. we determine the ratio of MAS's execution time to the one of GLS for each individual parameter setting. Values below 1.0 indicate that MAS outperforms GLS. As parameter of the experiment we vary number of servers, memory available, and look-ahead.

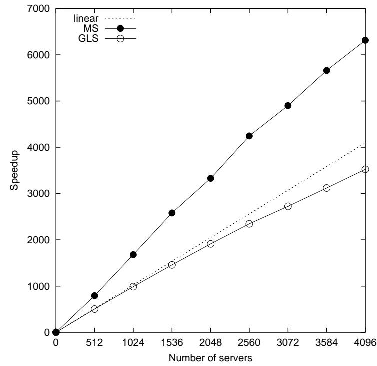

Figure 5. Speedup

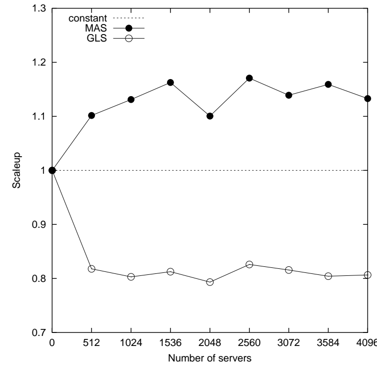

Figure 6. Scaleup

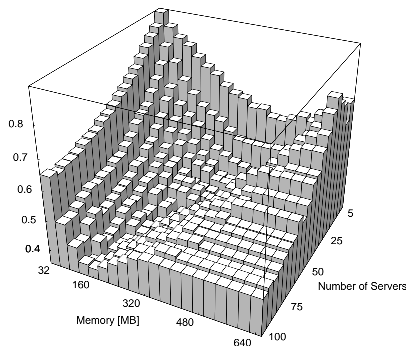

Figure 7. Relative performance; memory available and number of servers varied

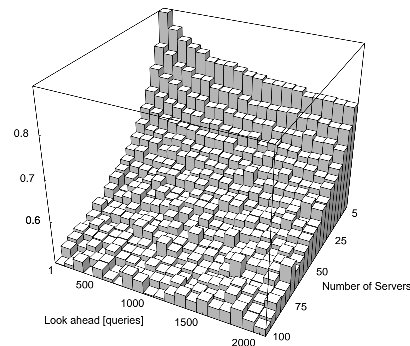

Figure 8. Relative performance; look-ahead and number of servers varied

Figure 7 shows the relative execution time as a function of the number of servers and the amount of memory per server. The number of servers varies between 5 and 100, memory between 32 and 640MB. As the diagram displays, MAS outperformed GLS in all 400 individual experiments achieving execution times as short as 40% of those of GLS. However, as the diagram reveals, this is no monotonic process, rather, with increasing memory sizes, GLS manages to "catch up", though only to a certain degree. See for example the front row where, after MAS increases its lead (0.4 at ca. 160MB) it cannot further improve on the running time whereas GLS becomes increasingly better. For more than ca. 320MB neither algorithm can achieve any improvement, thus the plateaux.

Figure 8 shows the relative performance as function of the number of servers and the look-ahead. All servers had 64MB of memory. The plot shows results that affirm the previously found ones now also in the direct comparison: For small number of servers look-ahead plays an important role (see last row), which fades as the number of server increases (see front row).

7 Discussion

While implementing and testing different versions of MAS we made several design decisions which are not self-evident at first sight and deserve to be discussed more elaborately in the following.

The most important question in the design process was whether to include more system information about the servers or not. Typically, cost models in parallel databases try to model the system—particularly the shared resources like network or shared disks—as detailed as possible. We chose not to incorporate this kind of information for three reasons: Firstly, the type of query we are dealing with cannot profit much from the kind of parallel processing these cost models have been developed for. Secondly, a cost computation that takes details at this fine granularity into account is computationally too expensive to deliver cost estimates for hundreds of servers when an online arrival of queries needs to be scheduled in real-time. Lastly, this detailed system information is hard to maintain. To keep it accurately up-to-date during the processing would require a substantial share of both network and processing resources and is thus unfeasible.

In contrast to that the cost values used by MAS are of simpler nature. The cost estimates for the query execution time (see Section 5.1) can be assembled from sequential cost estimates. Since the queries are emitted by a fixed interface, there is the possibility to pre-compile queries which are then instantiated with a few parameters. In that case highly accurate cost estimates can be pre-computed and—like the queries—instantiated with parameters.

The information necessary to describe the system for the scheduling is in so far easy to maintain as it consists only of a feedback indicating that processing of the last query has terminated, i.e. the result has been shipped, and what tables are now in main-memory. To avoid idleness between feedback and new assignment, servers can be extended to buffer *one* query while processing the previously assigned. The feedback together with the fact that servers do not maintain own query queues ensures that the impact of a few, inaccurate cost estimates, which can never be completely avoided, is kept down to a minimum.

Finally, the query *execution* we proposed does not allow for multi-programming, i.e. running several queries concurrently on one server. This choice was motivated by two facts. Firstly, multi-programming is not very beneficial in main-memory databases like Monet as concurrently processed queries often get into each other's way, unlike in I/O dominated database systems. Secondly, and more importantly as this holds for other database back-ends too, the potential gains of multi-programming where I/O of one query and CPU intensive computation of another one can be aligned is very limited as it is our foremost goal to avoid costly I/O operations at all.

8 Conclusion

Web-enabled databases and database back-end technology for large web-base information systems are one of the fastest growing segments of the database market. Those systems challenge the traditional repertoire of optimization techniques used in database technology. Powerful, user interfaces for multi-media object retrieval concurrently submit large numbers of queries to the database back-end which make throughput optimization a primary optimization goal.

In this paper, we propose a parallel query processing architecture and investigated the possibility to exploit data sharing by clever scheduling of the arriving queries. We have developed MAS, a scheduling strategy that tries to maximize the re-use of data resident in main memory across the database cluster. The algorithm is distinguished by its simplicity and robustness on one hand—the information needed to make MAS work is easy to obtain, accurate, and needs only little effort to be kept up-to-date—and its effectiveness on the other hand.

Our experiments show superior results compared to conventional list scheduling in terms of both query throughput and scaling behavior confirming our considerations. Moreover, by re-using the data available rather than assigning data actively, the scheduling algorithm adapts to changing hotspots.

Our future work is geared toward extending the scheduling schema to consider intermediate results and exploit similarities among queries.

References

[BK95] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proc. Basque International Workshop on Information Technology*, San Sebastian, Spain, 1995.

[BK99] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, 1999.

[BMK99] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the new Bottleneck: Memory Access. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 54–65, Edinburgh, UK, 1999.

[CHM95] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–265, San Jose, CA, USA, May 1995.

[DG92] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[FGND93] D. F. Ferguson, L. Georgiadis, C. Nikolaou, and K. Davies. Goal Oriented, Adaptive Transaction Routing for High Performance Transaction Processing Systems. In *Proc. of the Int'l. Conf. on Parallel and Distributed Information Systems*, pages 138–147, San Diego, CA, USA, January 1993.

[GI96] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 365–376, Montreal, Canada, June 1996.

[GI97] M. N. Garofalakis and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 296–305, Athens, Greece, September 1997.

[Gra69] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[Gra95] J. Gray. A Survey of Parallel Database Techniques and Systems. In *Tutorial Handouts of the 21st Int'l. Conf. on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[HM94] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelining Parallelism. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 36–47, Santiago, Chile, September 1994.

[HM95] W. Hasan and R. Motwani. Coloring Away Communication in Parallel Query Optimization. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 239–250, Zurich, Switzerland, September 1995.

[HS93] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[MBK00] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? - Dissecting CPU and Memory Optimization Effects. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, Cairo, Egypt, September 2000. Accepted for publication.

[MD93] M. Mehta and D. J. DeWitt. Dynamic Memory Allocation for Multiple-Query Workloads. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 354–367, Dublin, Ireland, September 1993.

[MSD93] M. Mehta, V. Soloviev, and D. J. DeWitt. Batch Scheduling in Parallel Database Systems. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 400–410, Vienna, Austria, April 1993.

[NZT96] M. G. Norman, T. Zurek, and P. Thanisch. Much Ado About Shared-Nothing. *ACM SIGMOD Record*, 25(3):16–21, September 1996.

[PMC$^+$90] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Proc. of the Int'l. Symp. on Databases in Parallel and Distr. Systems*, pages 4–29, Dublin, Ireland, July 1990.

[Rah92] E. Rahm. A Framewok for Workload Allocation in Distributed Transaction Processing Systems. *Systems Software Journal*, 18:171–190, 1992.

[RBS00] U. Röhm, K. Böhm, and H.-J. Schek. OLAP Query Routing and Physical Design in a Database Cluster. In *Proc. of the Int'l. Conf. on Extending Database Technology*, Lecture Notes in Computer Science, pages 254–268, Konstanz, Germany, March 2000.

[SD89] D. A. Schneider and D. J DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 110–121, May 1989.

[SD90] D. A. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 469–480, Brisbane, Australia, August 1990.

[SKWW00] A. R. Schmidt, M. L. Kersten, M. A. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In *International Workshop on the Web and Databases (In conjunction with ACM SIGMOD)*, pages 47–52, Dallas, TX, USA, May 2000.

[SSN94] K. Shim, T. Sellis, and D. Nau. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *IEEE Trans. on Knowledge and Data Engineering*, 12(2):197–222, March 1994.

[Sto86] M. Stonebraker. The Case for Shared-Nothing. *IEEE Data Engineering Bulletin*, 9(1):4–9, March 1986.

[Tho87] A. Thomasian. A Performance Study of Dynamic Load Balancing in Distributed Systems. In *Proc. of the IEEE Int'l. Conf. on Distributed Computing Systems*, Berlin, Germany, 1987.

[WFA95] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel Evaluation of Multi-Join Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 115–126, San Jose, CA, USA, May 1995.