Load Distribution of Analytical Query Workloads for Database Cluster Architectures

Thomas Phan *

Yahoo!, Inc.
701 First Ave.
Sunnyvale, CA 94089, USA
thomas.phan@acm.org

Wen-Syan Li

IBM Almaden Research Center
650 Harry Road
San Jose, CA, 95134, USA
wsl@us.ibm.com

ABSTRACT

Enterprises may have multiple database systems spread across the organization for redundancy or for serving different applications. In such systems, query workloads can be distributed across different servers for better performance. A materialized view, or Materialized Query Table (MQT), is an auxiliary table with pre-computed data that can be used to significantly improve the performance of a database query. In this paper, we propose a framework for coordinating execution of OLAP query workloads across a database cluster with shared nothing architecture. Such coordination is complex since we need to consider (1) the time to build the MQTs, (2) the query execution impact of the MQTs, (3) whether the MQTs can fit in the disk space limitation, (4) server computation power, and (5) the effectiveness of the scheduling and placement algorithms in deriving a combination of configurations so that the workload can be completed in the shortest time period. We frame the problem as a combinatorial problem with a solution space that is exponential in the number of queries, MQTs, and servers. We provide a stochastic search heuristic that finds a near-optimal mapping of queries-to-servers and MQTs-to-servers within an arbitrarily bounded time and compare our solution with an exhaustive search and three standard greedy algorithms. Our search implementation produced schedules within 9% of the optimal found through an exhaustive search and produced better solutions than typical greedy algorithms for both TPC-H and synthetic benchmarks under a variety of experiments. For a key trial where disk space is limited, it produced 15% better results than the next best competitor, corresponding to an absolute wall clock advantage of over 10 hours.

*The work described in this paper was performed while the author was with IBM Almaden Research Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

1. INTRODUCTION

Enterprises may have multiple database systems spread across the organization for redundancy or for serving different applications. For example, an insurance company may have two identical database systems, one as a production system for online transactions and one as a hot standby. The insurance company could also have additional database systems which have a subset of base tables for various applications, such as claim process, risk analysis, and cross sale analysis. In such systems, query workloads can be distributed across different servers for better performance.

These systems have their dedicated missions in the day time, but they could be left idle or under-utilized in the evening when batch workloads are processed. These systems may be located in the headquarters or data centers, or they may be distributed across multiple locations. In this paper, we propose to utilize and coordinate these available servers and use them as a database cluster to process OLAP-type query workloads.

A common approach to improving query running time is the use of materialized views, which we call Materialized Query Tables (MQTs). An MQT is an auxiliary table with pre-computed data that can be used to significantly improve the performance of a database query [18] [7]. To be consistent with work such as [2], we use the term MQT to refer to both the materialized view (with their indexes) as well as indexes on base tables.

Because MQTs are required in OLAP (Online Analytical Processing) applications in which the query workloads tend to have complex structures and syntax, a separate MQT Advisor product is often used to recommend MQTs [35] [1]. Several vendors have MQT Advisor products, including IBM DB2 Design Advisor [34] [35], RedBrick/Informix [28], Oracle 10g [24], and Microsoft SQL Server [21]. All of these MQT Advisors provide recommendations for only single server configurations.

In this paper, we propose a framework for coordinating and optimizing execution of OLAP query workloads across a cluster of database servers with shared-nothing architecture. For a database cluster, such an optimization is achieved when the maximum completion time of the workloads across all database servers is minimized. The completion time at each database server includes MQT and index building time and query workload execution time.

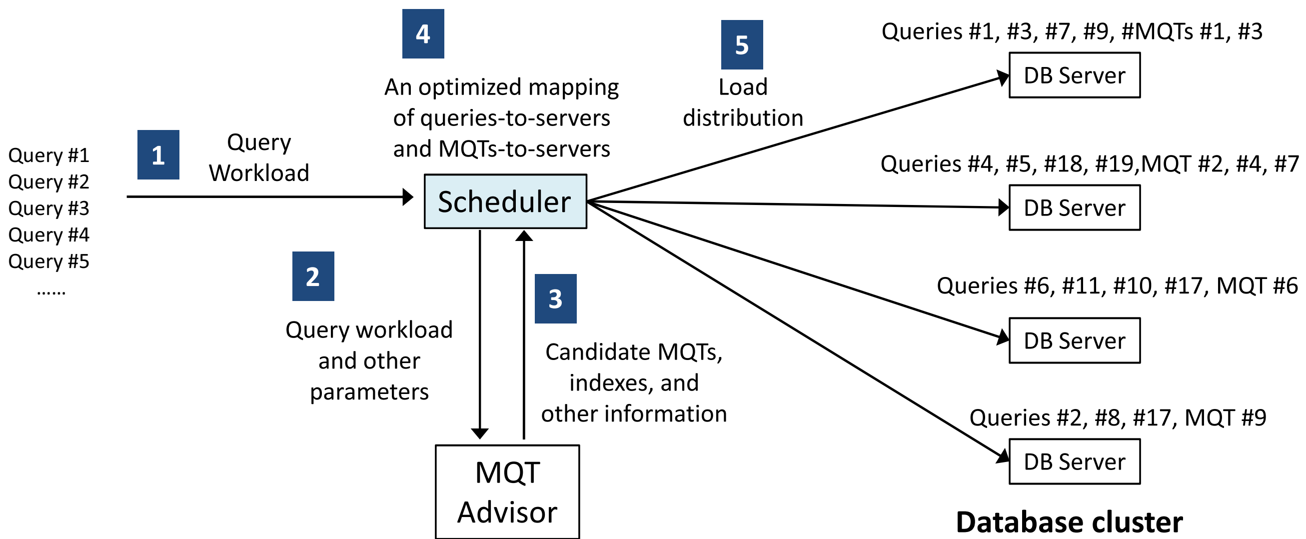

Figure 1: A high-level overview of how our scheduler system fits into a traditional DBMS. (1) A query workload is submitted to our scheduler. (2) The scheduler gives the queries to an MQT Advisor product, which (3) returns a set of candidate MQTs and associated indexes. (4) The scheduler runs a search heuristic against possible combinations and produces query-to-server and MQT-to-server mappings. (5) The mappings are used to distribute the queries and MQTs onto the servers.

A naïve manner of load distribution is to divide the workload into multiple sub-workloads and assign each sub-workload to a database server in some greedy manner, such as a round-robin distribution. This simple solution will not work for several reasons:

- queries routed to a database server may not be collocated with their needed MQTs;
- some MQTs may not fit in the data server that has a limited disk space;
- some sub-workloads may be more expensive to execute than others, so some server may be idle while other servers are still busy;
- some servers may be more powerful than others.

We assume the database cluster may consist of database servers. We would like to derive a load distribution solution such that the tasks of each server (building the MQTs and then running the sub-workload) are completed with minimum overall time. Such a solution is complex since we need to consider the time to build the MQTs, the query execution impact of the MQTs, whether the MQTs can fit in the disk space limitation, server computation power, and the solution space size of all combinations of configurations.

Furthermore, we would like a solution where no assumptions can be made about the presence of a query-routing optimizer, such as that found in [17], which assumes federation. Our approach only assumes that the servers in the cluster are independent and potentially heterogeneous with no other query placement optimizer than our own. We discuss these subtle differences more in Section 5 on related work.

In this paper we model the scenario as a combinatorial problem with a solution space that is exponential in the number of queries, MQTs, and servers. We provide a genetic algorithm (GA) search heuristic that finds a near-optimal mapping of queries-to-servers and MQTs-to-servers within an arbitrarily bounded time and compare our solution with an exhaustive search and two standard greedy algorithms. We evaluated the proposed framework using an extended TPC-H workload and compared it with various approaches. The experimental results validate scalability and effectiveness of our approach. Our GA implementation produced schedules within 9% of the optimal found through an exhaustive search and produced better solutions than typical greedy algorithms for both TPC-H and synthetic benchmarks under a variety of experiments. For a key trial where disk space is limited, it produced 15% better results than the next best competitor, corresponding to an absolute wall clock advantage of over 10 hours.

This paper is organized in the following manner. In Section 2 we describe the problem space where queries and MQTs are distributed across servers and then propose a search heuristic to find near-optimal solutions. In Section 3 we describe how we implemented our genetic algorithm. In Section 4 we show our experiments and results from our implementation. We discuss related work in Section 5 and conclude our paper in Section 6.

2. QUERY AND MQT PLACEMENT

2.1 Problem space

Figure 1 shows how our system fits into an existing database framework. Our key component is a scheduler that operates on the queries and the MQTs produced by an existing MQT Advisor product.

Once the MQTs are recommended by the MQT Advisor, they and the queries may be spread across several servers in a distributed environment such as the clustered system we are considering. These servers may be placed across multiple branch sites, or they can be redundant nodes located at the

same site. Our scheduler distributes the workload's queries and MQTs according to query-to-server and MQT-to-server mappings. This paper is focused on finding the optimal mappings that produces the shortest workload running time.

Figure 2 is an abstract representation of the distribution problem. Suppose there are Q queries in the workload, M candidate MQTs recommended by the MQT Advisor, and S servers. We define the problem with the following assumptions.

1. Each of the Q queries can be assigned to exactly one of the S servers. There are thus S^Q total combinations of such assignments.

2. Each of the M MQTs can be replicated across the S servers. Since each MQT is either at or is not at each server (a boolean condition), there are $2^{(M \times S)}$ total combinations.

3. Every time a candidate MQT is replicated to a server, a materialization cost is incurred. In our experiments, we use the actual materialization cost from the MQTs produced from a TPC-H workload.

4. We assume that the servers have access to the base tables needed for the queries.

5. Each query may be dependent on one or more candidate MQTs, where this mapping is given by the MQT Advisor product. We further assume that each query's execution time is largely taken up by the execution of one or more long-running sub-queries, and when the MQT Advisor evaluated the query, it properly recommended one or more candidate MQTs created by these sub-queries. Thus, the execution time of a given query is largely determined by whether or not its required MQTs are collocated at the same server. We thus use: (a) a query-to-MQTs requirement mapping; (b) sub-query running times with collocated MQTs; and (c) sub-query running times without collocated MQTs. In our experiments, these parameters are again provided by the MQTs produced from a TPC-H workload.

6. In this paper we assume that the time to execute the queries and materialize the MQTs dominate any network delays (transmission, propagation, or queueing).

7. The execution time of the workload as a whole is defined to be the maximum execution of its queries across the servers. In the job-scheduling research community, this value is known as the *makespan*.

The objective of the system is to minimize item (7), the maximum workload execution time across the servers. (An alternative optimization objective would to minimize the variance between the execution times.) In other words,

- Let $t[s]$ be the execution time of server s
- $\forall s$, let $T \geq t[s]$
- The objective is to minimize T.

The resulting problem is to find optimal mappings of queries-to-servers and MQTs-to-servers that meets the objective. Intuitively, we would like a schedule of these two mappings

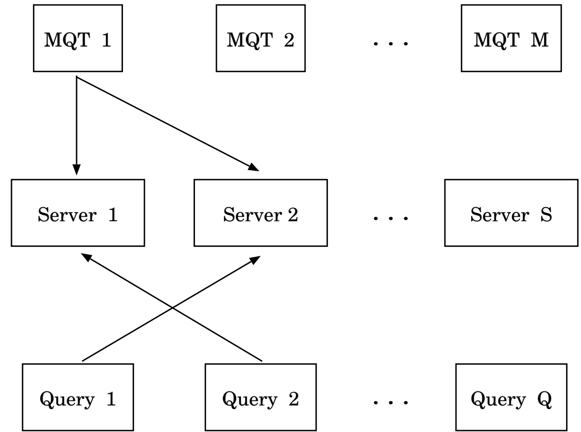

Figure 2: The allocation problem. There are Q queries, M MQTs, and S servers. Each query is assigned to one server. Each MQT can be replicated across multiple servers.

that produces a high number of collocations for a given number of MQT materializations. Such a workload would produce the best net effect between the positive factor of having a collocation (where the query can read the MQT) and the two negative factors of having to materialize an MQT and of not having a collocation (where the query must read the base tables).

Given (1) and (2) above, it can be seen that the exhaustive number of different distribution combinations is $S^Q \times 2^{(M \times S)}$. Even for small parameters, the solution space grows exponentially, making an exhaustive search infeasible.

2.2 Genetic algorithm search heuristic

Given the solution space of $S^Q \times 2^{(M \times S)}$, the goal is to find the best placements of queries and MQTs onto the servers to minimize the execution time. To search through the solution space, we use a genetic algorithm (GA) global search heuristic that allows us to explore portions of the space in a guided manner that converges towards the optimal solutions [14] [13]. We note that a GA is only one of many possible approaches for a search heuristic; others include tabu search, simulated annealing, and steepest-ascent hill climbing. We use a GA only as a tool.

GAs have been used to solve a variety of optimization problems. [11] and [20] provide good surveys. A GA is a computer simulation of Darwinian natural selection that iterates through various generations to converge toward the best solution in the problem space. A potential solution to the problem exists as a chromosome, and in our case, a chromosome is a specific mapping of queries-to-servers and MQTs-to-servers along with its associated workload execution time. Genetic algorithms are commonly used to find optimal exact solutions or near-optimal approximations in combinatorial search problems such as the one we address. It is known that a GA provides a very good tradeoff between exploration of the solution space and exploitation of discovered maxima [13].

Pseudo code for a genetic algorithm is shown in Algorithm 1. The GA executes as follows. The GA produces an initial random population of chromosomes. The chromosomes then recombine (simulating sexual reproduction)

Algorithm 1 Genetic search algorithm

1: **FUNCTION** Genetic algorithm
2: **BEGIN**
3: Time t
4: Population $P(t)$:= new random Population
5:
6: **while** ! done **do**
7: recombine and/or mutate P(t)
8: evaluate($P(t)$)
9: select the best $P(t+1)$ from $P(t)$
10: $t := t+1$
11: **end while**
12: **END**

to produce children using portions of both parents. Mutations in the children are produced with small probability to introduce traits that were not in either parent. The children with the best scores (in our case, the lowest workload execution times) are chosen for the next generation. The steps repeat for a fixed number of iterations, allowing the GA to converge toward the best chromosome. In the end it is hoped that the GA explores a large portion of the solution space. With each recombination, the most beneficial portion of a parent chromosome (that is, the best portions of the queries-to-servers and MQTs-to-servers mappings) is ideally retained and passed from parent to child, so the best child in the final generation has the best mappings. To improve the GA's convergence, we implemented elitism, where the best chromosome found so far is guaranteed to exist in each generation.

Note that the GA is not guaranteed to find the optimal solution since the recombination and mutation steps are stochastic. However, as we show later, our GA implementation consistently comes within 9% of the optimal solution without the exponential running time of an exhaustive search.

We chose a GA for several reasons. From our own prior work, we are familiar with its operations and the factors that affect its performance and optimality convergence. Additionally, the queries-to-server and MQTs-to-server mappings are ideally suited to array and matrix representations, allowing us to use prior GA research that aid in chromosome recombination [9].

3. GENETIC ALGORITHM DESIGN

3.1 Chromosome representation

The chromosomes are the mappings and their associated scores. The GA recombines and evaluates these chromosomes during its execution.

Given the fact that we are modeling two mappings, namely the mapping of queries onto the servers and the MQTs onto the servers, it is intuitive that the mappings could be implemented as a one-dimensional array and a two-dimensional matrix, respectively. In the 1D array for the query placements, the ith element could contain the server identifier at which the ith query is placed. In the 2D matrix for the MQT placements with replication, position (i, j) could be a boolean indicator that MQT i is at server j.

However, from our previous work, we have learned that having two separate sets of chromosomes for a multi-objective

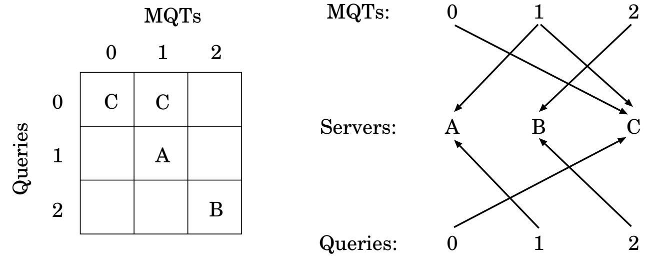

Figure 3: This figure shows a chromosome data structure on the left and the physical mapping that it represents on the right. A chromosome is a collocation mapping of queries and MQTs to the same server. Its data structure is a 2-dimensional $Q \times M$ matrix where row i, column j is the server at which query i and MQT j are collocated. For example, it can be seen in the matrix (left) that query 0 and MQT 1 are collocated at server C, which is also evident in the physical mapping (right).

optimization has a hard time converging. We instead use a single unified chromosome that represents collocations of queries and MQTs onto the same server.

The layout of the chromosome is shown in Figure 3. The chromosome is a 2D matrix shown on the left of the figure. Position (i, j) in the matrix contains the server identifier where query i is collocated with MQT j or *null* if they are not collocated. The physical collocations are shown on the right of the figure. Note that this representation is a very concise way of showing collocations. It omits unnecessary MQT-to-server placements that do not result in a collocation with a server.

3.2 Chromosome recombination and mutation

Two parent chromosomes recombine to produce a new child chromosome. The hope is that the child contains the best contiguous chromosome regions from its parents.

An example recombination is shown in Figure 4. A cut point in the range of $[0, Q-1]$ is chosen, and the rows above the cut point from one parent are combined with the rows below the cut point from the other parent. Since each row represents the collocations of a particular query with the MQTs, the child inherits the respective collocations from its parents. Note that new collocations may arise as shown in the figure.

To help the GA explore more of the solution space, we implemented mutations in new children. A mutation consists of randomly-introduced MQT-to-server links with low probability (we use 0.1 in accordance with GA best practices). Note that the mutation is introduced into the child before the chromosome is formed, meaning that the new MQT-to-server links will be ignored unless it results in a new collocation with a server.

3.3 Evaluation function

The evaluation function returns the resulting workload execution time given a chromosome and the requirement mapping between queries and MQTs (which is produced by the MQT Advisor). Note the function can be implemented to evaluate the workload in any way so long as it is consistently applied to all chromosomes across all generations.

Our evaluation function is shown in Algorithm 2. In lines

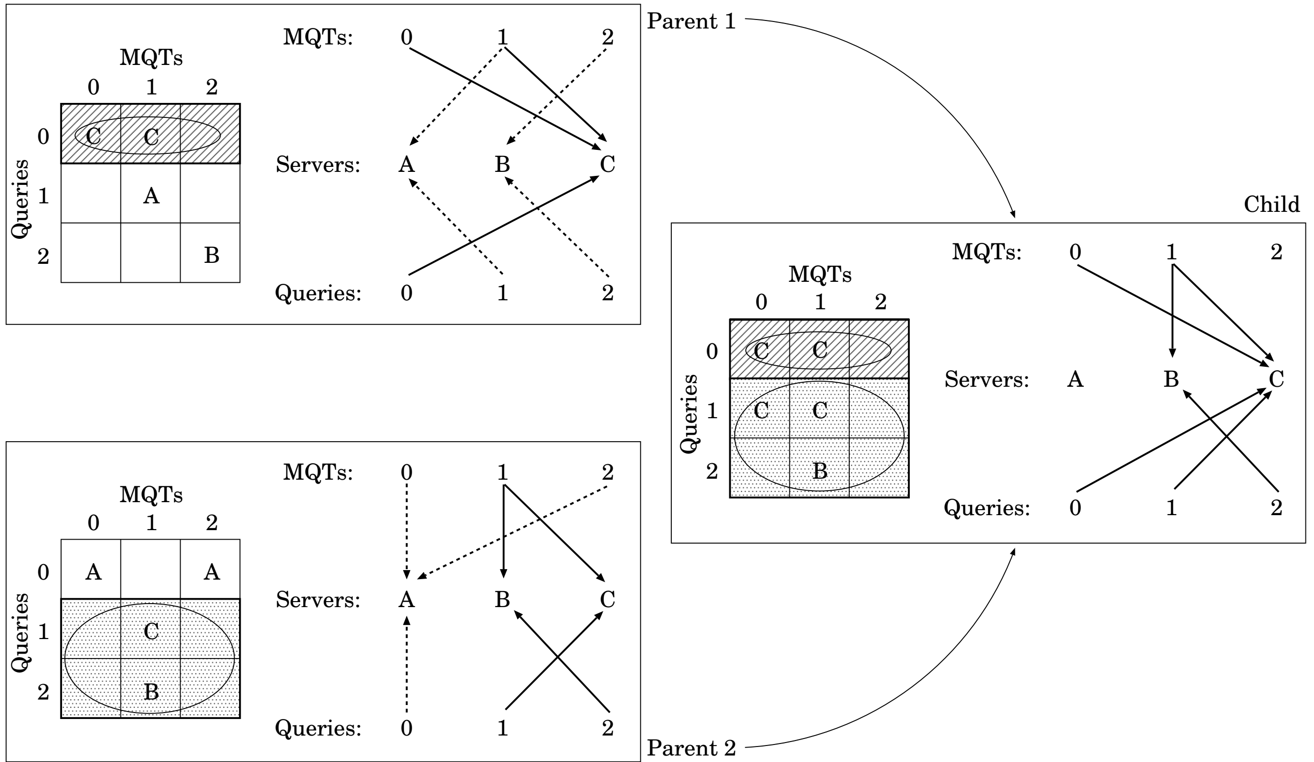

Figure 4: This figure shows an example recombination between two parents to produce a new child. Parent 1 is the same chromosome shown in Figure 3. Initially a cut point is randomly chosen in the range $[0, Q-1]$. The rows $[0, cutpoint-1]$ from parent 1 and the rows $[cutpoint, Q-1]$ from parent 2 are chosen. Since each row of the matrix represents a collocation of one query and several MQTs, the chosen rows represent sets of collocations. In this figure, a cut point of 1 was chosen. In parent 1 in the upper-left box, row range $[0, 0]$ was selected (shown shaded), and the resulting physical collocations are shown to the right of the matrix using dark solid arrows. The dashed arrows represent collocations which were not inherited by the child. In parent 2 in the lower-left box, row range $[1, 2]$ was selected. In the child in the right box, the respective collocations from parent 1 and parent 2 are combined into a new chromosome. Note that a new collocation was created (the collocation of query 1 and MQT 0 at server C).

7 to 9, it initializes the execution times for all the servers in the chromosome. In lines 11-16, it checks to see what MQTs have been materialized and at what server. It accordingly adds these materialization times to the respective servers. Lines 18 to 29 are the key loop that looks across all collocations. If a query needs a particular MQT, the query's execution with the MQT is added to the execution time if the query and MQT are collocated. If they are not collocated, then the query's execution time without the MQT is added (where this execution time is the time to run the sub-query against the base tables). The function returns the maximum execution time among the servers.

4. EXPERIMENTS

4.1 Setup

The goals of our experiments were to: (1) show that our genetic algorithm implementation does a good job in finding solutions and quantify how close it comes to the optimal solutions that are found via an exhaustive search; and (2) compare the performance of the GA against greedy algorithms that run faster than the GA but may not find as good a solution. In the experiments that follow, we use the following names for the scheduler algorithms:

- **GA**: our genetic algorithm implementation described in Section 3
- **Exhaustive**: an exhaustive searching algorithm that iterates through all $S^Q \times 2^{(M \times S)}$ combinations
- **Greedy1-MQTs-then-queries**: a greedy algorithm that deals out the MQTs across the servers in round-robin fashion and then assigns each query to the server that hosts the most number of its needed MQTs
- **Greedy2-queries-then-MQTs**: a greedy algorithm that assigns the queries across the servers in round-robin fashion and then places only the MQTs that are needed by each query onto the respective servers
- **Greedy3-no-MQTs**: a greedy algorithm that assigns the queries across the servers in round-robin fashion without any MQTs

In our experiments we used two types of data, namely a workload produced by the standard TPC Benchmark H (TPC-H) generator and a synthetic benchmark based on the

Algorithm 2 GA evaluation function

```
1: FUNCTION evaluate
2: IN: CHROMOSOME, a representation of the colloca-
   tion of queries and MQTs to servers
3: IN: REQUIRES, a mapping between queries and
   MQTs
4: OUT: runningtime, the running time of this workload
5: BEGIN
6:
7: for (each server ∈ CHROMOSOME) do
8:   set server's running time to 0
9: end for
10:
11: {Compute the materialization cost of each MQT}
12: for (each mqt ∈ CHROMOSOME) do
13:   if mqt was materialized at server then
14:     server's running time += mqt's materialization
        time
15:   end if
16: end for
17:
18: for (each query ∈ CHROMOSOME) do
19:   server := query's server
20:   for (each mqt ∈ CHROMOSOME) do
21:     if query and mqt ∈ REQUIRES then
22:       if query and mqt are collocated then
23:         server's running time += query's running
            time with mqt
24:       else
25:         server's running time += query's running
            time without mqt
26:       end if
27:     end if
28:   end for
29: end for
30:
31: runningtime := maximum running time of each server
32: return runningtime
33: END
```

queries and MQTs from the TPC-H benchmark. The initial TPC-H workload was a set that we extended from 22 queries to 133 queries with the added queries defined by a Business Intelligence performance team [1]. These additional queries were defined specifically to simulate BI applications with complex queries and result in an average query processing cost equivalent to that of the original 22 queries. The principal reason we used this extended set was to allow the IBM DB2 MQT Advisor to recommend more MQTs to be materialized, resulting in a bigger solution space. Specifically, for the 133-query workload, the MQT Advisor recommended 21 MQTs and 21 base table indexes, whereas for the 22-query workload the MQT Advisor recommended only 9 MQTs and 26 base table indexes. The total amount of TPC-H data was approximately 20GB.

The result of this step were (1) a mapping of queries to needed MQTs, (2) query execution times with and without each MQT, and (3) MQT sizes. We further wanted to study the algorithms' scalability with different query and MQT counts, but since running the TPC-H generator and the MQT Advisor product with different initial workloads would create different MQT characteristics (1-3 in this paragraph), we used an additional synthetic benchmark to normalize the results. The synthetic benchmarks produced query-to-MQT mappings, query executions times, and MQT sizes with the same statistical distributions as that produced by the TPC-H benchmark. These parameters are shown in Table 1.

We implemented our scheduler algorithms in standard C++ and ran our system on an off-the-shelf desktop computer running Red Hat Linux with a Pentium IV 2.8 Ghz CPU, 2GB of RAM, and 1MB of cache. In all the experiments that follow, the results shown are the results averaged over 20 trials per data point.

We ran our genetic algorithm for 100 generations with a population size of 100 chromosomes.

4.2 Comparison against exhaustive search

We initially wanted to quantify how well the GA does in finding an optimal solution. Figure 5 shows a comparison of the workload execution times produced by the exhaustive search and the GA using the synthetic workload. We vary the number of queries on the x-axis. It can be seen that the GA produces workload times that are very close, if not exactly the same as the exhaustive search. In Figure 6 we show the scheduled workload running times as a function of an increasing number of MQTs. Again, the GA produced running times very close to that of the exhaustive search.

For completeness we graph the percent difference between the GA and the exhaustive search for the two preceding experiments in Figure 7 and Figure 8, respectively. The difference between the two does not reach beyond 8.60%.

It is not surprising at all that the primary difference between the two scheduling algorithms is their running times. Figure 9 shows the running times *of the scheduler itself* for the exhaustive search and the GA for an increasing number of MQTs. The GA consistently finished within two seconds. Since the exhaustive search must examine all combinations in the solution, its running time grows exponentially. Since the number of combinations is $S^Q \times 2^{(M \times S)}$, its running time is very sensitive to M, the number of MQTs, which is reflected in the figure. We suspect that the exhaustive search's algorithm could be improved by a constant factor with faster hardware and a tighter software implementation, but its growth is exponential nonetheless.

Note that we would have shown data points for more MQTs had the running time for the exhaustive search not been so high. Experiments with more than 8 MQTs would have reached into durations of several hours and then days.

We conclude from this trial that our GA is able to provide a schedule very close to the optimal solution provided by the exhaustive search but without its high running time.

4.3 TPC-H workload

In these experiments we compared the running times resulting from the schedules produced by the GA and the greedy algorithms. We used a TPC-H workload that resulted in 133 queries and 42 combined MQTs and indexes on them, as mentioned earlier in section 4.1.

Figure 10 shows the running time for the TPC-H benchmark workload after having been scheduled by the GA and the three greedy algorithms. The x-axis shows the increasing number of servers. In this trial we configured the schedulers

[1]This workload is available for other researchers. Please email us if interested.

Experimental parameter	Comment
Query running time with MQT	Gaussian (λ = 137156 milliseconds, μ = 20420 milliseconds)
Query running time without MQT	Gaussian (λ = 1910363 milliseconds, μ = 171428 milliseconds)
MQT materialization time	Gaussian (λ = 4753468 milliseconds, μ = 292773 milliseconds)
MQTs required per query	Gaussian (λ = 2.1, μ = 0.1)
MQT size	Gaussian (λ = 523 MB, μ = 150 MB)

Table 1: Experimental parameters for the synthetic benchmark taken by analyzing a 133 query, 42 MQT TPC-H workload

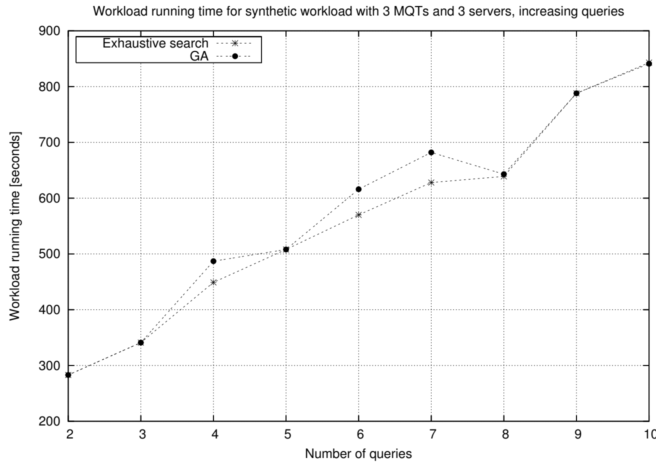

Figure 5: Comparison of the scheduled workload running times produced by the exhaustive search and the GA with an increasing number of queries.

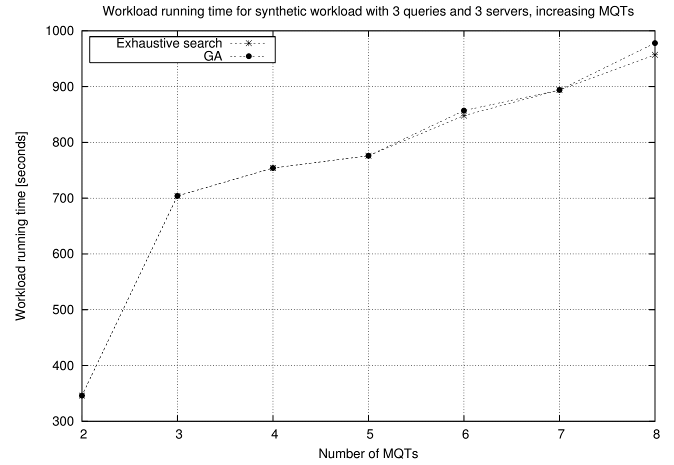

Figure 6: Comparison of the scheduled workload running times produced by the exhaustive search and the GA with an increasing number of MQTs.

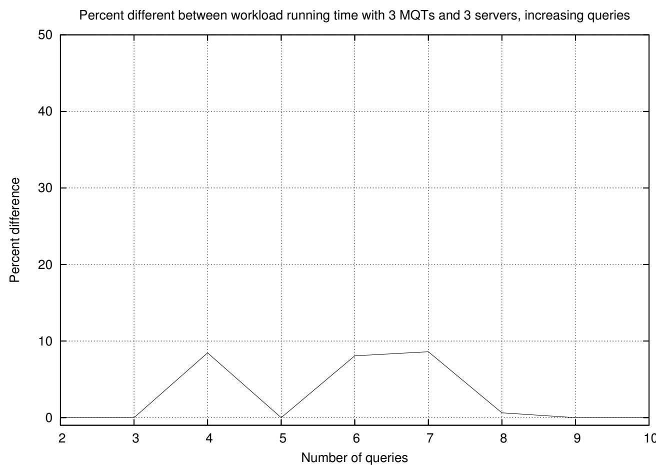

Figure 7: Percent difference between the exhaustive search and GA scheduled workload times shown in Figure 5.

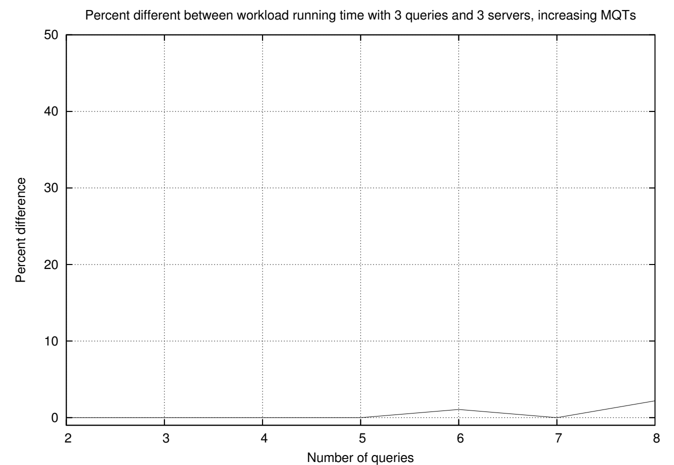

Figure 8: Percent difference between the exhaustive search and GA scheduled workload times shown in Figure 6.

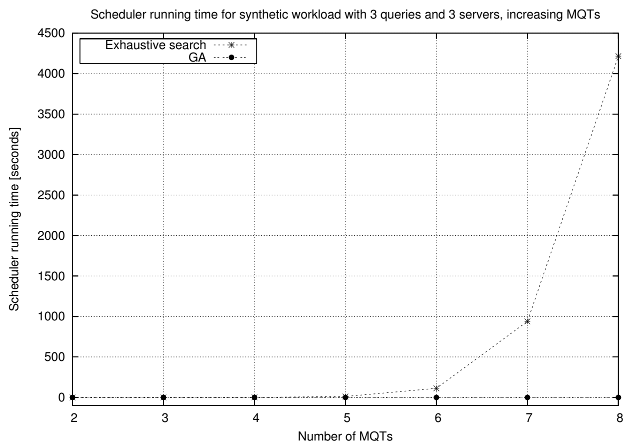

Figure 9: Comparison of the running time *of the scheduler itself* **for the exhaustive search and the GA.**

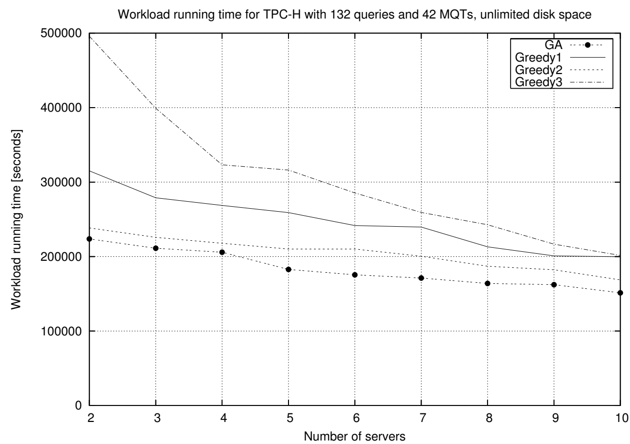

Figure 10: Running time of TPC-H workload with unlimited disk space for the MQTs.

to assume that each server had an infinite amount of available disk space to hold all 42 possible MQTs. (In reality at most 25GB of space was needed to hold all the MQTs.)

This figure shows several trends. First, as the number of servers increases, the workload running times decrease approximately linearly. This result is expected since the queries are being spread across the servers in the cluster, allowing for parallelism. Although the parallel execution is not a focus of this paper, we do note that the performance gain does not increase exactly as the number of servers increases. The reason is that the MQT materialization time is not fully parallelized: even though the queries are distributed to run in parallel, the MQTs may be replicated across many servers, and each server accumulates its MQTs' materialization costs. Second, the GA's schedule results in a better workload running time than the greedy algorithms. As we will discuss in the next subsection, this performance is due to the fact that the GA converges toward a schedule that produces a high number of MQT materializations but with a high number of query and MQT collocations. Third, the relative performance of the greedy algorithms shows that Greedy2 (which places queries first and then MQTs) produces a faster workload running time than Greedy1 (which places MQTs first and then queries) and Greedy3 (which places queries but does not use any MQTs). We will more closely examine the reasons for this behavior later.

In Figure 11 we show the results of running the TPC-H workload with an increasing amount of disk space per server for the MQTs. We fixed the number of servers at 4. It can be seen for all schedulers that the workload running time decreases as the disk space increases. Given more disk space, more MQTs may be materialized, incurring a higher materialization cost, but jointly more collocations with queries will occur as well. In the next subsection we show how the increasing disk space affects materializations and collocations. The GA does particularly well in this key case where disk space is a limiting factor. For example, with 10000MB the GA's workload running time is 15% better than that of Greedy2, its closest competitor, which is an absolute advantage of approximately 36000 seconds, or 10 hours.

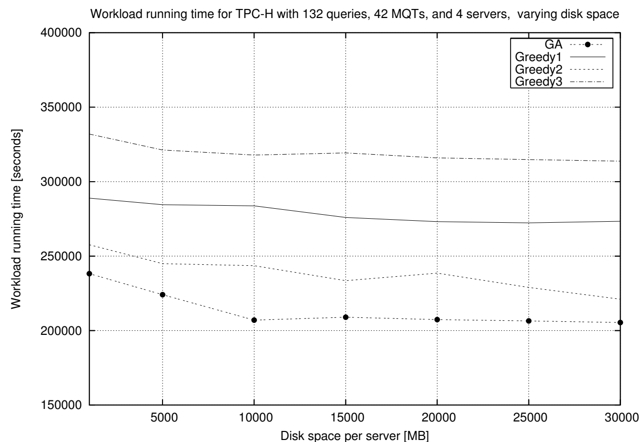

Figure 11: Running time of TPC-H workload with an increasing amount of disk space available for the MQTs. There were 4 servers.

4.4 MQT behavior

We examine in more detail the results from the previous subsection. As we described earlier, the scheduling algorithms differ in their treatment of the number of MQT materializations and the likelihood of collocations between the MQTs and the queries. These differences impact the running times of the workloads.

Greedy3 places queries across the servers but does not materialize any MQTs, and as a result it avoids materialization costs but instead accumulates more base table accesses. Greedy2 spreads queries first across the servers and then places each query's needed MQTs on the same servers, resulting in guaranteed collocations but a high number of materializations. Greedy1 spreads MQTs first across the servers and then places each query onto the server with the highest number of its needed MQTs, resulting in an intermediate number of materializations and an intermediate number of collocations. The GA aims to have a high number of collocations to offset the number of materializations and

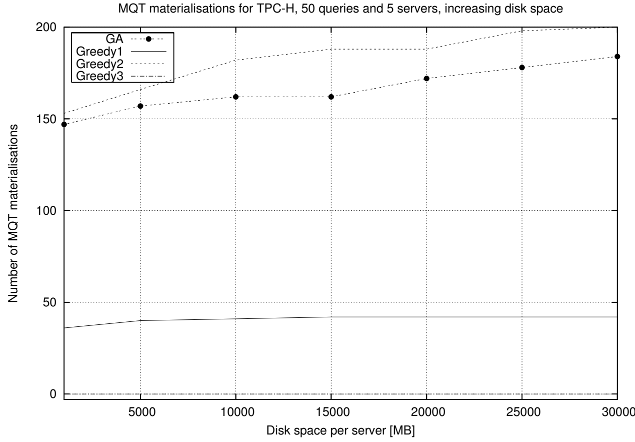

Figure 12: Number of MQT materializations for TPC-H workload.

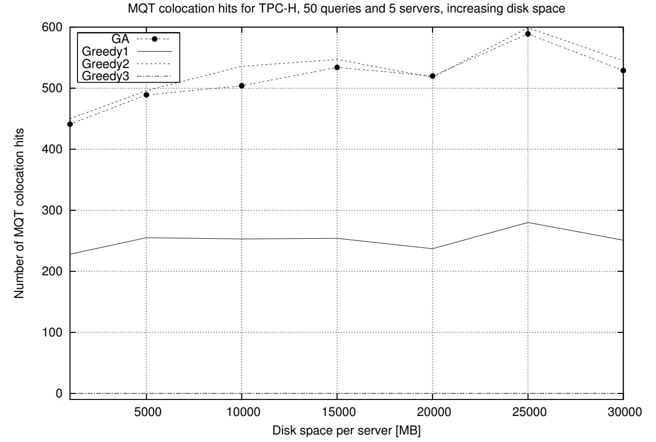

Figure 13: Number of MQT collocation hits for TPC-H workload.

query accesses to base tables (in the absence of collocations).

Greedy3 can also be interpreted as a baseline case against which the other algorithms are measured: since Greedy3 does not incur any MQT materialization cost but does have a high base table access cost, the other algorithms that do use MQTs should have enough collocation net gains to outweigh its materialization costs.

Figure 12 shows the number of MQT materializations for the TPC-H workload as a function of disk space available for MQTs. As can be see, Greedy2 algorithm produce the highest number of materializations as expected, Greedy3 produces no materializations as expected, while Greedy1 falls in the middle. The GA actually produces a number of materializations that is quite close to Greedy2, which is somewhat counterintuitive since the materialization cost contributes a fairly large portion to the workload execution time. However, this materialization cost is put to good use since it provides a high number of collocations.

Figure 13 shows the number of MQT collocation hits from the sample experiment. Since Greedy2 guarantees that all queries are collocated with their MQTs, its curve represents the upper bound. The GA comes very close to this upper bound; coupled with the fewer materializations vis-a-vis Greedy2, it can be seen that the GA produces a better net workload gain than Greedy2, which was what was seen in Figure 11. It can also be seen that Greedy3 had no collocation hits as expected, while Greedy1 produced an intermediate amount.

4.5 Scalability

Here we looked to see the behavior of the scheduling algorithms as we scale the number of queries and MQTs from a baseline value up to their respective numbers from the TPC-H configuration. As was mentioned in Section 4.1, we used a synthetic benchmark for this purpose and used the MQT characteristics as input into the random statistical distributions of the MQTs in this benchmark.

Figure 14 shows the workload running time for the synthetic workload as we scale the number of queries. As can be expected, the GA produces the best running times, followed closely by Greedy2 and then Greedy 3 and Greedy1. These results correspond to our findings from the previous subsection. The GA is able to produce a sufficiently high number of collocations that offset the costs of materializations and base table accesses (in lieu of collocations with MQTs). Greedy2 comes close since it guarantees collocations but has a higher cost of more materializations.

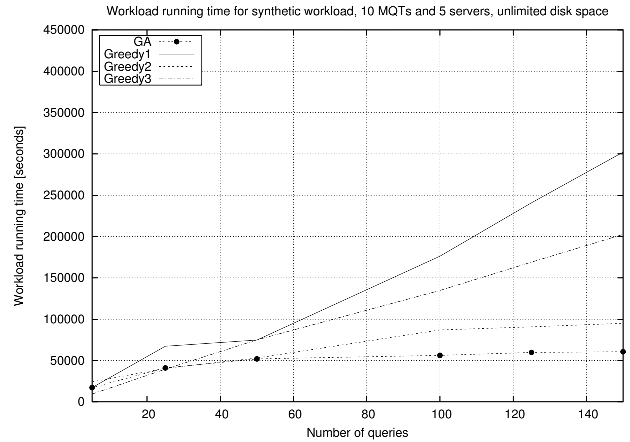

Figure 14: Impact of the number of queries on workload execution time.

In Figure 15 we show the workload running time for an increasing number of MQTs. The same relative ordering takes place among the algorithms, but the absolute workload running times has increased. Since the number of unique MQTs needed per query is fairly constant, having a larger number of available MQTs means that the working set of active MQTs is necessarily larger.

4.6 Effect of heterogeneous servers

In the previous experiments we assumed the servers were homogeneous. Here we examine the impact of varying CPU performance. We normalized an “average” server to have a CPU performance factor of 1.0 and then assigned performance factors to all servers using a Gaussian distribution with a mean of 1.0 and a standard deviation of 0.25. The query processing time was increased or decreased ac-

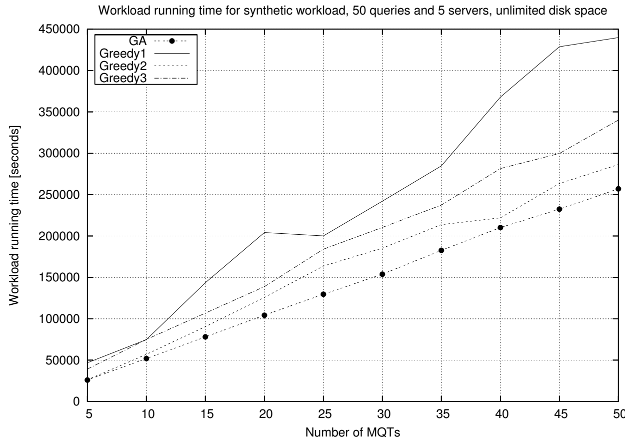

Figure 15: Impact of the number of MQTs on workload execution time.

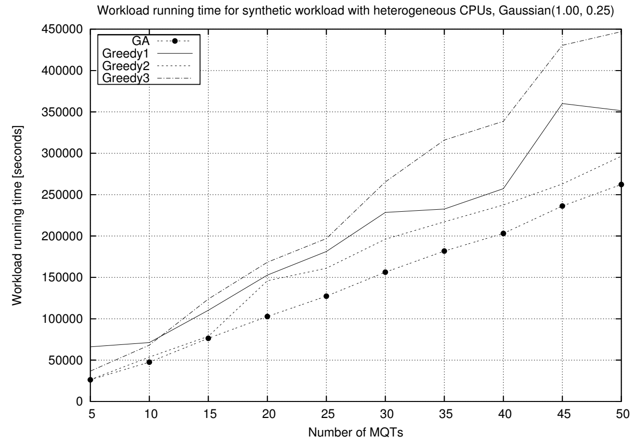

Figure 16: Workload running time with heterogeneous server CPUs. The servers' CPU powers were normalized with an average CPU power of 1.0, and each server's power was set to a random value with a Gaussian distribution with λ=1.0 and μ=0.25.

cordingly by this factor as an estimate of CPU performance variation. Of course, there are many factors which affect query processing, and CPU performance is but one among several such as RAM, cache speed, disk speed, and shared CPU load. As a first approximation, our approach makes a simple estimation. In future work we look to take measured server characteristics as inputs into our scheduler.

Figure 16 shows the impact of heterogeneous server power on the running time of the TPC-H workload. The same relative ordering of the algorithms can be seen as before. We observe that this degree of CPU heterogeneity does not impact the relative merits of each. In the future we will look to implement other algorithms that can better adapt to server heterogeneity and also explore other factors such as disk and network heterogeneity.

4.7 GA running time

For completeness, we examine the running time of the GA scheduler itself. From the experiment of Section 4.5 that produced Figure 15 showing the workload running time for an increasing number of MQTs, we observed that the GA took on average 61.74 seconds with a standard deviation of 4.59 seconds. Of course, this time is fairly arbitary since we had set up our GA to run for specifically 100 generations. As with most stochastic search techniques, the GA can spend more time in order to get closer to the optimal value.

We note that the other greedy algorithms took under a second to run under almost all cases. However, we feel that the faster workloads that the GA provides outweigh its execution time.

4.8 Summary

We summarize the results of this lengthy section. We have shown that the GA produces a workload execution time that is nearly as fast as the optimal time found through an exhaustive search. Since the comparisons between the GA and the exhaustive shown in subsection 4.2 scaled only up to 10 queries and 8 MQTs (since the exhaustive search took too long to run), we expect the GA's near-optimal performance to scale as well. In any event, the GA's convergence towards the optimal solution for larger numbers of queries and MQTs can be improved with more interations.

We also showed that the GA produces better workload execution times than the greedy algorithms. The GA consistently produces fewer MQT materializations than its closest competitor, Greedy2, but has nearly as many MQT collocation hits. The net result of these two factors is a better schedule with a shorter workload time.

5. RELATED WORK

The problem of *view maintenance* has been studied previously (e.g. [29]) in the context of materialized views in data warehouses. Since a data warehouse consists of a large view, the main focus of the database research has been the maintenance of materialized views *incrementally*. Numerous algorithms have been proposed for incremental view maintenance [30]. These techniques are complementary to our work in the sense that they can be used for the local refreshment of the MQTs recommended by our scheduler.

The work described in [19] presents a new framework to enable network and server load aware information integration. A component called the QCC (Query Cost Calibrator) is introduced to monitor network and remote server load conditions and provides feedback to the federation server. This work does not recommend MQTs or base tables for replication, but it operates on existing data placement settings while providing adaptability to select a proper set of servers that yield the fastest response time. It is limited to federated systems and cannot be applied to database clusters or enable parallelism.

Another closely related work is the IBM DB2 physical database design advisor [35] [27]. It can be used for a shared-nothing parallel database system in which data is horizontally partitioned among multiple independent nodes. The physical design advisor is used as a partition advisor in a shared-nothing parallel database system in [27]. Given a workload of SQL statements, the partition advisor seeks to automatically determine how to partition the base data across multiple nodes to achieve overall optimal (or close to optimal) performance for that workload. It uses the query

optimizer itself not only to recommend candidate partitions for each table that will benefit each query in the workload but also to evaluate various combinations of these candidates. The partition advisor can also provide advice for partition and placement of MQTs by evaluating combinations of MQTs (or their partitions) and base tables (or their partitions) in the same node. The evaluation is cost-based. The work in [35] assumes each node has identical computation power and availability of parallelism-aware optimizer. Our work does not make such assumptions.

The work described in [17] exploits correlation and parallelism for materialized-view recommendation in federated systems. This work proposed a federated MQT advisor to recommend MQTs at remote servers to improve parallelism. It does not assume all remote nodes are identical but does assume that all nodes are coordinated via a federation server. Our paper does not assume federation and can work on a cluster of database servers.

Scheduling is also used in the scientific computing community in the context of scheduling jobs onto compute nodes. For example, other researchers have looked at finding optimal schedules for data transfers across sites for grid computing [5] [3]. Other researchers have has also looked into global optimization algorithms for job scheduling [4] [32], but they do not consider simultaneous job and data placement, analogous to our work with simultaneous query and MQT placement.

The distribution of MQTs and queries across database servers is related to data placement problems also found in the domains of cluster computing and grid computing. In a typical cluster system, files can be placed at one of several compute nodes. Processes arrive at each of the compute nodes; if the needed file is already at that node, then processing occurs locally, but if the file is remote, then the request is transferred to the node containing the file for processing to proceed at that node. In a typical grid computing system, files (or more abstractly, data objects or data feeds) can be assigned to nodes in the grid, and arriving tasks can be placed at each of the nodes. In neither of these contexts has there been work in simultaneous scheduling of data and computation: data placement and computation placement are decoupled, so each placement of data or computation is in reaction to a prior placement of the other.

[10] provides a survey of solutions to the file assignment problem where files must be assigned to compute nodes. Fourteen solutions are described where files are initially mapped to nodes, and arriving compute tasks are either executed at that node or are redirected elsewhere. Note that this approach assumes an initial distribution of the files, with a subsequent distribution of the compute tasks. This model is close to one of our greedy algorithms (Greedy1-MQTs-then-queries) where MQTs are placed first using some rules or heuristics, and then queries are directed to the best nodes using another set of rules or heuristics. Further, the fourteen solutions take shortcuts, including (a) finding non-optimal solutions (e.g. [12], (b) assuming that the files can be broken up into pieces (e.g. [8], and (c) assuming infinite storage at each compute node. None of the solutions there address all of these issues together (e.g. [15]). Our approach does handle these issues: (a) the GA finds near-optimal solutions, (b) we assumes that MQTs cannot be broken up, and (c) we place the MQTs using per-server storage limits as a first-class constraint.

[33] surveys solutions to the data assignment problem in the context of grid computing. In a typical system, the data placement is decoupled from the job placement by scheduling the job close to or at the source of the data or by accessing a replica, where closeness is refers to a site with minimum transfer time [26]. Other approaches follow similar strategies for reducing the response time of jobs by minimizing the input and and output data transfer time. [22] assumes that single-file input data has already been replicated across sites and then uses an exhaustive algorithm to search across all combinations to find the minimum cost. [31] treats files systems systems at each site as a passive cache. [6] does decouple job scheduling from data scheduling: at the end of the job scheduling, the popularity of needed files is calculated and the used by the data scheduler to replicate data for the *next* set of jobs, which may or may not share the same data requirements as the previous set of jobs.

6. CONCLUSION

Database systems may be spread across multiple DBMS servers to improve performance or redundancy. Since batch query workloads may rely on materialized query tables to improve their running time, the resulting problem is to distribute the queries and the MQTs across the servers to minimize the overall workload execution time. In this paper we framed the problem in the context of a combinatorial search across mappings of queries-to-servers and MQTs-to-servers, and we showed that the solution space was exponential. To explore this space, we implemented a search heuristic using a genetic algorithm. This approach produced schedules within 9% of the optimal found through an exhaustive search and produced better solutions than typical greedy algorithms for both TPC-H and synthetic benchmarks.

In the future we look to extend our work in several ways. In this paper we have assumed that the servers in the database cluster all have access to all the underlying base tables; in the future we look to investigate ways to optimize a schedule to take advantage of servers having only a subset of the base tables. We also look to compare our algorithms with other scheduling approaches from the job scheduling community, such as schedulers using dynamic programming or network flow optimization. Finally, we will look to explore other stochastic search heuristics.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. "Automated selection of materialized views and indexes for SQL database," In *Proceedings of VLDB.*

[2] S. Agrawal, E. Chu, and V. Narasayya. "Automatic physical design tuning: workload as a sequence," In *Proceedings of SIGMOD*, 2006.

[3] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. "Task scheduling strategies for workflow-based applications in grids," In *Proceedings of CCGrid*, 2005.

[4] T. Braun, et al. "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, June 2001.

[5] J. Casanova, A. Legrand, D. Zagorodnow, and F. Berman. "Heuristics for scheduling parameter sweep

applications in grid environments," In *Proceedings of the Heterogeneous Computing Workshop*, 2000.

[6] A. Chakrabarti, et al. "Integration of Scheduling and Replication in Data Grids." In Proceedings of the International Conference on High Performance Computing, 2004.

[7] S. Chaudhuri, R. Krishnamurthy, S. Potmianos, and K. Shim. "Optimizing Queries with Materialized Views," In *Proceedings of ICDE*, 1995.

[8] P. Chen. "Optimal file allocation in multilevel storage systems." In Proceedings of AFIPS, 1973.

[9] L. Davis. "Job Shop Scheduling with Genetic Algorithms," In *Proceedings of the International Conference on Genetic Algorithms*, 1985.

[10] L. Dowdy and D. Foster. "Comparative Models of the File Assignment Problem." ACM Computing Surveys, vol 14, no. 2, 1982.

[11] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*, Springer 2007.

[12] D. Foster, L. Dowdy, and J. Ames. "File assignment in a computer network." Computer Networks 5, Sept. 1981.

[13] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Kluwer Academic, 1989.

[14] J. Holland. *Adaptation in natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[15] P. Hughes and G. Moe. "A structural approach to computer performance analysis." In Proceedings of AFIPS, 1973.

[16] IBM DB2 9, `www.ibm.com/software/data/db2/9/`

[17] H. Jiang, D. Gao, W.-S. Li. "Exploiting Correlation and Parallelism for Materialized-View Recommendation in Distributed Data Warehouses," In *Proceedings of ICDE*, 2007.

[18] P. Larson and H. Yang. "Computing Queries From Derived Relations," *In Proceedings of VLDB*, 1985.

[19] W.-S. Li, V. Batra, V. Raman, W. Han, K. Candan, and I. Narang. "Load and Network Aware Query Routing for Information Integration," In *Proceedings of ICDE*, 2005.

[20] Z. Michaelwicz and D. Fogel. *How to Solve It: Modern Heuristics*, Springer 2004.

[21] Microsoft SQL Server, `www.microsoft.com/sql/default.mspx`

[22] H. Mohamed and D. Epema. "An evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusers." In Proceedings of the IEEE International Conference on Cluster Computing.

[23] MySQL Cluster `www.mysql.com/products/database/cluster/`.

[24] Oracle, `www.oracle.com`

[25] Oracle 11g Real Application Clusters, `www.oracle.com/technology/products/database/clustering/index.html`

[26] K. Ranganathan and I. Foster. "Computation Scheduling and Data Replication Algorithms for Data Grids." Grid Resource Management: State of the Art and Future Trends. Kluwer Academic Publishers, 2003.

[27] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. "Automating physical database design in a parallel database," In *Proceedings of SIGMOD*, 2002.

[28] Redbrick, `www.informix.com`

[29] E. Rudensteiner, A. Koeller, and X. Zhang. "Maintaining Data Warehouses over Changing Information Sources," *Communications of the ACM*, vol. 43, no. 6, 2000.

[30] K. Salem, K. Beyer, R. Cochrane, and B. Lindsay. "How to roll a join: Asynchronous Incremental View Maintenance," In *Proceedings of SIGMOD*, 2000.

[31] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. "Exploiting Replications and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids." In Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing, 2004.

[32] E. Schmueli and D. Feitelson. "Backfilling with lookahed to optimize the packing of parallel jobs," *Springer-Verlag Lecture Notes in Computer Science*, vol 2862, 2003.

[33] A. Venugopal, R. Buyya, and K. Ramamohanarao. "A taxonomy of Data Grids for distributed data sharing, management, and processing." ACM Computing Surveys, vol 31, no. 1, 2006.

[34] D. Zilio, et al. "Recommending Materialized Views and Indexes with IBM DB2 Design Advisor," In *Proceedings of ICAC*, 2004.

[35] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. "DB2 Design Advisor: Integrated Automatic Physical Database Design," In *Proceedings of VLDB*, 2004.