Resource Bricolage for Parallel DBMSs on Heterogeneous Clusters

Jiexing Li †, Jeffrey Naughton #, Rimma V. Nehme *

†Google Inc. jiexing@google.com

#University of Wisconsin, Madison naughton@cs.wisc.edu

*Microsoft Jim Gray Systems Lab rimman@microsoft.com

ABSTRACT

Running parallel database systems in an environment with heterogeneous resources has become increasingly common, due to cluster evolution and increasing interest in moving applications into public clouds or shared infrastructures. For database systems running in a heterogeneous cluster, the default uniform data partitioning strategy may overload some of the slow machines while at the same time it may underutilize the more powerful machines. Since the processing time of a parallel query is determined by the slowest machine, such an allocation strategy may result in a significant query performance degradation.

We take a first step to address this problem by introducing a technique we call *resource bricolage* that improves database performance in heterogeneous environments. Our approach quantifies the performance differences among machines with various resources as they process workloads with diverse resource requirements. We formalize the problem of minimizing workload execution time and view it as an optimization problem, and then we employ linear programming to obtain a recommended data partitioning scheme. We verify the effectiveness of our technique with an extensive experimental study on a commercial database system.

1. INTRODUCTION

With the growth of the Internet, our ability to generate extremely large amounts of data has dramatically increased. This sheer volume of data that needs to be managed and analyzed has led to the wide adoption of parallel database systems (DBMSs). Gamma [14] and Teradata [13] were some of the early parallel DBMSs to address the need to process this massive amount of data. Later on, many commercial parallel DBMSs were developed to provide this support. Examples include Aster nCluster [1], IBM DB2 Parallel Edition [7], Oracle nCUBE [8], Pivotal Greenplum [2], and Microsoft SQL Server Parallel Data Warehouse [3]. In order to get strategic insights to make business decisions, running online analytical processing (OLAP) workloads on parallel DBMSs has become increasingly important. To achieve high performance and scalability, these parallel DBMSs typically partition data across machines in a shared-nothing cluster to exploit data parallelism. A query running on such a system is typically broken up into subqueries, which are executed in parallel on the separate data chunks.

Nowadays, running parallel DBMSs in an environment with heterogeneous resources has become increasingly common, due to cluster evolution and increasing interest in moving applications into public clouds or shared infrastructures.

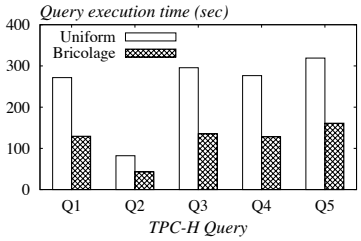

Figure 1: Query execution times with different data partitioning strategies.

Cluster evolution. When a cluster is first built, it typically begins with a set of identical machines. Over time, old machines may be reconfigured, upgraded, or replaced, and new machines may be added, thus resulting in a heterogeneous cluster.

Public clouds. With the proliferation of cloud computing, more and more parallel DBMSs are moving into public clouds. Previous research has revealed that the supposedly identical instances provided by public clouds often exhibit measurably different performance. Performance variations exist extensively in disk, CPU, memory, and network [16, 24, 32, 34].

Shared infrastructures. For systems that remain in private clusters, it is desirable to consolidate different workloads into a shared infrastructure to exploit data locality and multiplex physical resources. However, in a shared environment, it is hard to guarantee that resources available on different machines will always be the same for a given application. Since applications might request different amounts of resources on different machines, this may leave behind machines with vastly different available resources and capacities for new applications.

1.1 Motivation

Performance differences among machines (either physical or virtual) in the same cluster pose new challenges for parallel database systems. By default, parallel systems ignore differences among machines and try to assign the same amount of data to each. If these machines have different disk, CPU, memory, and network resources, they will take varying amounts of time to process the same amount of data. Unfortunately, the execution time of a query in a parallel DBMS is determined by its slowest machine. At worst, a

The original version of this article was published in VLDB 2015.

slow machine can substantially degrade the performance of the query.

On the other hand, a fast machine in such a system will be under-utilized, finishing its work early, sitting idle and waiting for the slower machines to finish. This suggests that we can reduce execution time by allocating more data to more powerful machines and less data to the overloaded slow machines, in order to reduce the execution time of the entire query. In Figure 1, we compare the execution times of the first 5 TPC-H queries running on a heterogeneous cluster with two different data partitioning strategies. One strategy partitions the data uniformly across all the machines, while the other partitions the data using our proposed technique, which we present in Section 4. The detailed cluster setup is described in Section 5. As can be seen from the graph, we can significantly reduce total query execution time by carefully partitioning the data.

Our task is complicated by the fact that whether a machine should be considered powerful or not depends on the workload. For example, a machine with powerful CPUs is considered "fast" if we have a CPU-intensive workload. For an I/O-intensive workload, it is considered "slow" if it has limited disks. Furthermore, to partition the data in a better way, we also need to know how much data we should allocate per machine. Obviously, enough data should be assigned to machines to fully exploit their potential for the best performance, but at the same time, we do not want to push too far to turn things around by overloading the powerful machines. The problem gets more complicated when queries in a workload have different (mixed) resource requirements, as usually happens in practice. Thus, for a workload with a mix of I/O, CPU, and network-intensive queries, the partitioning of data with the goal of reducing overall execution time is a non-trivial task.

1.2 Our Contributions

To improve performance of parallel DBMSs running in heterogeneous environments, we propose a technique we call ***resource bricolage***. The term bricolage refers to construction or creation of a work from a diverse range of things that happen to be available, or a work created by such a process. The keys to the success of bricolage are knowing the characteristics of the available items, and knowing a way to utilize and get the most out of them during construction.

In the context of our problem, a set of heterogeneous machines are the available resources, and we want to use them to process a database workload as fast as possible. Thus, to implement resource bricolage, we must know the performance characteristics of the machines that execute database queries, and we must also know which machines to use and how to partition data across them to minimize the workload execution time. To do this, we quantify differences among machines by using query optimizer and a set of profiling queries that estimate the machines' performance parameters. We then formalize the problem of minimizing workload execution time and view it as an optimization problem that takes the performance parameters as input. We solve the problem using a standard linear program solver to obtain a recommended data partitioning scheme. In Section 4.4, we also discuss alternatives for handling nonlinear situations. We implemented our techniques and tested them in Microsoft SQL Server Parallel Data Warehouse (PDW) [3], and our experimental results show the effectiveness of our proposed solution.

The rest of the paper is organized as follows. Section 2 formalizes the resource bricolage problem. Section 3 describes our way of characterizing the performance of a machine. Section 4 presents our approach for finding an effective data partitioning scheme. Section 5 experimentally confirms the effectiveness of our proposed solution. Section 6 briefly reviews the related work. Finally, Section 7 concludes the paper with directions for future work.

2. THE PROBLEM

2.1 Formalization

To enable parallelism in a parallel database system, tables are typically horizontally partitioned across machines. The tuples of a table are assigned to a machine either by applying a partitioning function, such as a hash or a range partitioning function, or in a round-robin fashion. A partitioning function maps the tuples of a table to machines based on the values of specified column(s), which is (are) called the partitioning key of the table. As a result, a partitioning function determines the number of tuples that will be mapped to each machine.

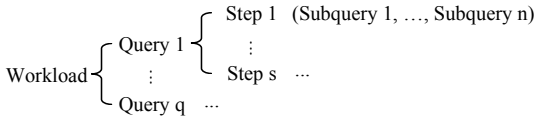

Figure 2: A query workload.

In our work, we target the problem of improving the performance of OLAP workloads in heterogeneous environments. In general, we may have a set of heterogeneous machines with different disk, CPU, network performance, and different amounts of memory. At the same time, we have a workload with a set of SQL queries as shown in Figure 2. A query can be further decomposed into a number of *steps* with different resource requirements. For each step, there will be a set of identical subqueries executing concurrently on different machines to exploit data parallelism. A step will not start until all steps upon which it depends on, if any, have finished. Thus, the running time of a step is determined by the longest-running subquery. The query result of a step will be repartitioned to be utilized by later steps, if needed.

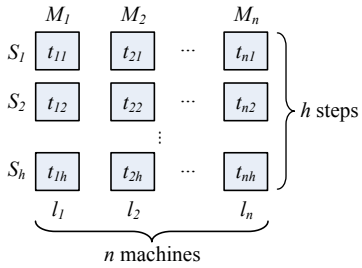

Figure 3: Problem setting.

We visually depict our problem setting in Figure 3. Let M_1, M_2, ..., M_n be a set of machines in the cluster, and let W be a workload consisting of multiple queries. Each query consists of a certain number of steps, and we concatenate all the steps in all of the queries to get a total of h steps: S_1, S_2, ..., S_h. Assume that t_{ij} would be the execution time for step S_j running on machine M_i if all the data were assigned to

M_i. Each column in the picture corresponds to a machine, and each row represents the set of subqueries running on the machines for a particular step. In addition, we assume that a machine M_i also has a storage limit l_i, which represents the maximum percentage of the entire data set that it can hold. The goal of resource bricolage is to find the best way to partition data across machines in order to minimize the total execution time of the entire workload.

2.2 Challenges

Whether it is worth allocating data to machines in a non-uniform fashion is dependent on the characteristics of the available computing resources. In [19], we compared the performance of three data partitioning strategies: *Uniform*, *Delete*, and *Optimal*. *Uniform* assigns the same amount of data to each machine. *Delete* is a simple heuristic that attempts to handle resource heterogeneity. It excludes some slow machines before it partitions the data uniformly to the remaining ones. Starting with the slowest set of machines, the machine exclusion attempt is repeated until no further improvement can be made. *Optimal* is the ideal data partitioning strategy that distributes data to machines in a way that minimized the overall execution time. We proved that in the worst case, the workload execution time of the *Delete* approach can be $\frac{n}{4}$ times as long as that of the *Optimal* approach [19].

Thus, it is important for us to come up with the optimal partitioning strategy to better utilize computing resources. To do this, there are a number of challenges that need to be tackled. First of all, we need to quantify performance differences among machines in order to assign the proper amounts of data to them. Second, we need to know which machines to use and how much data to assign to each of them for best performance. Intuitively, we should choose "fast" machines, and we should add more machines to a cluster to reduce query execution times. However, we found that this is not true for the worst-case example we discussed in [19]. In the worst-case example, the set of "fast" machines do not collaborate well with others in the same system, resulting in longer execution times.

3. QUANTIFYING PERFORMANCE DIFFERENCES

For each machine in the cluster, we use the runtimes of the queries that will be executed to quantify its performance. Since we do not know the actual query execution times before they finish, we need to estimate these values.

There has been a lot of work in the area of query execution time estimation [9, 10, 20, 22, 27]. Unlike previous work, we do not need to get perfect time estimates to make a good data partitioning recommendation. Instead, the ratios in time among machines are the key information that we need in order to assign the proper amounts of data to machines. Consider a very simple example with just two machines. Even if we can not estimate the query runtimes perfectly for these two machines, we can still assign the right amount of data to them, as long as we can correctly estimate how much faster or slower one machine is compared to the other. Thus, we adopt a less accurate but much simpler approach to estimate query execution times. Our approach can be summarized as follows. For a given database query, we retrieve its execution plan from optimizer and divide the plan into a set of pipelines. We then use the optimizer's cost model to estimate the CPU, I/O, and network "work" that needs to be done by each pipeline. To estimate the times to execute the pipelines on different machines, we run profiling queries to measure the speeds to process the estimated work for each machine.

3.1 Estimating the Cost of a Pipeline

Like previous work on execution time estimation [10, 22], we use the execution plan for a query to estimate its runtime. An execution plan is a tree of physical operators chosen by a query optimizer. In addition to the most commonly used operators in a single-node DBMS, such as Table Scan, Filter, Hash Join, etc., a parallel database system also employs data movement operators, which are used for transferring data between DBMS instances running on different machines.

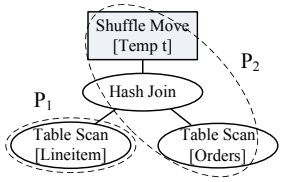

Figure 4: An execution plan with two pipelines.

An execution plan is divided into a set of pipelines delimited by blocking operators (e.g., Hash Join, Group-by, and data movement operators). The example plan in Figure 4 is divided into two different pipelines P_1 and P_2. Pipelines are executed one after another, and the total runtime of a query is the sum of the execution time(s) of its pipeline(s). To estimate a pipeline's execution time, we first predict what is the work of the pipeline and what is the speed to process the work. We then estimate the runtime of a pipeline as the estimated work divided by the processing speed.

For each pipeline, we use the optimizer's cost model to estimate the work (called *cost*) that needs to be done by CPUs, disks, and network, respectively. These costs are estimated based on the available memory size. We utilize the optimizer estimated cost units to define the work for an operator in a pipeline. We follow the idea presented in [20] to calculate the cost for a pipeline, and the interested reader is referred to that paper for details.

The default optimizer estimated cost is calculated using parameters with predefined values (e.g., the time to fetch a page sequentially), which are set by optimizer designers without taking into account the resources that will be available on the machine for running a query. Thus, it is not a good indicator of the actual query execution time for a specific machine. To obtain more accurate predictions, we keep the original estimates and treat them as estimated work if a query were to run on a "standard" machine with default parameters. Then, we test on a given machine to see how fast it can go through this estimated work with its resources (the speeds).

3.2 Measuring Speeds to Process the Cost

Measuring I/O speed. To get the speed to process the estimated I/O cost for a machine, we execute the following query with a cold buffer cache: *select count(*) from T*. This query simply scans a table T and returns the number of tuples in the table. It is an I/O-intensive query with negligible CPU cost. For this query, we use the query optimizer to get its estimated I/O cost, and then we run it to obtain

its execution time for the given machine. Then we calculate the I/O speed for this machine as the estimated I/O cost divided by the query execution time.

Measuring CPU speed. To measure the CPU speed, we test a CPU-intensive query: *select T.a from T group by T.a* from a warm buffer cache. We get its estimated CPU cost and runtime, and we calculate the CPU speed for this machine by dividing cost by runtime. Since small queries tend to have higher variation in the cost estimates and execution times, one practical suggestion is to use a sufficiently big table for the test. Meanwhile, since the time spent on transferring query results from a database engine to an external test program is not used to process the estimated CPU cost, we need to limit the number of tuples that will be returned. In our experiment, T contains $18M$ unsorted tuples, and only 4 distinct $T.a$ values are returned.

Measuring network speed. We use a separate program to test the network speed instead of a query running on an actual database system. The reason is that it is hard to find a query to test the network speed while isolating all other factors that can contribute to query execution times. For a query with data movement operators in a fully functional system, the query may need to read data from a local disk and store data in a destination table. If network is not the bottleneck resource, we can not observe the true network speed. Thus, we wrote a small program to simulate the actual system for transmitting data between machines. We run this program at its full speed to send (receive) data to (from) another machine that is known to have a fast network connection. At the end, we calculate the average bytes of data that can be transferred per second as the network speed for the tested machine.

Finally, for a pipeline P, we estimate its execution time as the maximum of $C_{Res}(P)/Speed_{Res}$, for any Res in {CPU, I/O, network}. The execution time of a plan is the sum of the execution times of all pipelines in the plan.

4. RESOURCE BRICOLAGE

After we estimate the performance differences among machines for running our workload, we now need to find a better way to utilize the machines to process a given workload as fast as possible. We model and solve this problem using linear programming, and we deploy special strategies to handle nonlinear scenarios.

4.1 Base and Intermediate Data Partitioning

Data partitioning can happen in two different places. One is base table partitioning when loading data into a system, and the other one is intermediate result reshuffling at the end of an intermediate step. For example, consider a subquery of a step that uses the execution plan shown in Figure 4. This plan scans two base tables: *Lineitem* and *Orders*, which may be partitioned across all machines. The result of this subquery, which can be viewed as a temporary table, is served as input to the next steps, if there are any. Thus, the output table may also be redistributed among the machines.

The execution time of a plan running on a given machine is usually determined by the input table sizes. For example, the runtime of the plan in Figure 4 depends on the number of *Lineitem* and *Orders* (L and O for short) tuples. The runtime of a plan that takes a temporary table as input is again determined by the size of the temporary table.

In some cases, the partitioning of an immediate table can be independent of the partitioning of any other tables. For example, if the output of $L \bowtie O$ is used to perform a local aggregate in the next step, we can use a partitioning function different from the one used to partition L and O to redistribute the join results. However, if the output of $L \bowtie O$ is used to join with other tables in a later step, we must partition all tables participating in the join in a distribution-compatible way. In other words, we have to use the same partitioning function to allocate the data for these tables.

In our work, we consider data partitioning for both base and intermediate tables. Note that our technique can also be applied to systems that do not partition base tables a priori or do not store data in local disks. For these systems, our approach can be used to decide the initial assignment of data to the set of parallel tasks running on machines with heterogeneous resources, and similarly, our approach can be used for intermediate result reshuffling. Instead of reading pre-partitioned data from local disks, these systems read data from distributed file systems or remote servers. In order to apply our technique, we need to replace the time estimates for reading data locally with the time estimates for accessing remote data. We omit the details here since it is not the focus of our paper.

4.2 The Linear Programming Model

Next, we will first give our solution to the situation where all tables must be partitioned using the same partitioning function, and then we extend it to cases where multiple partitioning functions are allowed at the same time.

Recall that in our problem setting, we have n machines, and the maximum percentage of the entire data set that machine M_i can hold is l_i. Our workload consists of h steps, and it would take time t_{ij} for machine M_i to process step S_j if all data were assigned to M_i. The actual t_{ij} values are unknown, and we use the technique proposed in Section 3 to estimate them. We want to find a data partitioning scheme that can minimize the overall workload execution time.

When all tables are partitioned in the same way, we can use just one variable to represent the percentage of data that goes to a particular machine for different tables. Let p_i be the percentage of the data that is allocated to M_i for each table. We assume that the time it takes for M_i to process step S_j is proportional to the percentage of data assigned to it. Based on this assumption, $p_i t_{ij}$ represents the total time to process p_i of the data for step S_j running on machine M_i. The execution time of S_j, which is determined by the slowest machine, is $\max_{i=1}^{n} p_i t_{ij}$. Then the total execution time of the workload can be calculated as $\sum_{j=1}^{h} \max_{i=1}^{n} p_i t_{ij}$. In order to use a linear program to model this problem, we introduce an additional variable x_j to represent the execution time of step S_j. Thus, the total execution time of the workload can also be represented as $\sum_{j=1}^{h} x_j$. The linear program that minimizes the total execution time of the workload can be formulated as below.

For step S_j, since the execution time x_j is the longest execution time of all machines, we must have $p_i t_{ij} \leq x_j$ for machine M_i. We also know that the percentage of data that can be allocated to M_i must be at least 0 and at most l_i. The sum of all p_is is 1, since all data must be processed. We can solve this linear programming model using standard linear optimization techniques to obtain the values for p_is ($0 \leq i \leq n$) and x_js ($0 \leq j \leq h$), where the set of p_i values represents

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{h} x_j \\ \text{subject to} \quad & p_i t_{ij} \leq x_j \qquad 1 \leq i \leq n, 1 \leq j \leq h \\ & \sum_{i=1}^{n} p_i = 1 \\ & 0 \leq p_i \leq l_i \qquad 1 \leq i \leq n \end{aligned}$$

a data partitioning scheme that minimizes $\sum_{j=1}^{h} x_j$. Note that we may use only a subset of the machines, since we do not need to run queries on a machine with 0% of the data. Thus, the data partitioning scheme suggests a way to select the most suitable set of machines and a way to utilize them to process the database workload efficiently.

4.3 Allowing Multiple Partitioning Functions

When different partitioning functions are allowed to be used by different tables, we are given more flexibility for making improvements. Thus, we want to apply different partitioning functions whenever possible. In order to do this, we need to identify sets of tables that must be partitioned in the same way to produce join-compatible distributions, and we apply different partition functions to tables in different sets.

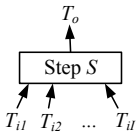

Figure 5: The input and output tables for a step.

For step S in workload W, let $\{T_{i1}, T_{i2}, ..., T_{iI}\}$ be the set of its input tables and T_o be its output table as we show in Figure 5. An input table to S could be a base table or an output table of another step, and all input tables will be joined together in step S. In order to perform joins, tuples in these tables must be mapped to machines using the same partitioning function, otherwise tuples that can be joined together may end up on different machines.

We define a ***distribution-compatible group*** as the set of input and output tables for W that must be partitioned using the same function, together with the set of steps in W that take these tables as input. Placing a step to a group implies that how the tables can be partitioned in a group has a significant impact on the execution time of the step. If we can find all distribution-compatible groups for W, we can apply different functions to tables in different groups for data allocation.

Given a database, we assume that the partitioning keys for base tables and whether two base tables should be partitioned in a distribution-compatible way or not are designed by a database administrator or an automated algorithm [4, 29, 31]. As a result, we know which base tables should belong to a distribution-compatible group. For intermediate tables, we need to figure this out. We generate the distribution-compatible groups for a workload W in the following way:

We omit replicated tables in our problem. Since a full copy of a replicated table will be kept on a machine, there is no need to worry about partitioning.

1. Create initial groups with corresponding distribution-compatible base tables according to the database design.
2. For each step S in W, perform the following three instructions.
   (a) For the input tables in S, find the groups that they belong to. If more than one group is found, ***merge*** them into a single group.
   (b) ***Assign*** S to the group.
   (c) ***Create*** a new group with the output table of S.

After the distribution-compatible groups are generated, we can employ the linear programming model proposed above to obtain a partitioning scheme for the tables to minimize total runtime of the steps in the group. For more details, we refer the reader to [19].

4.4 Handling Nonlinear Growth in Time

In our proposed linear programming model, we assume that query execution time changes linearly with the data size. Unfortunately, this assumption does not always hold true for database queries. This assumption is valid for the network cost of a query, where the transmission time increases in proportion to data size. It is also true for the CPU and I/O costs of many database operators, such as Table/Index Scan, Filter, and Compute Scalar. These operators take a large proportion of query execution times for analytical workloads.

The linear assumption may, however, be invalid for multi-phase operators such as Hash Join and Sort. We may introduce errors by choosing fixed linear functions for these operators in the following way. To estimate the t_{ij} value for step S_j running on machine M_i, we first assume that M_i gets $1/n$ of the data. We then use the query optimizer to generate the execution plan for S_j, and we estimate the runtime for the plan. Finally, the estimated value is magnified n times and returned as the t_{ij} value for S_j running on M_i. Based on all t_{ij}s we predict, a recommended partitioning is computed using the linear programming model, and the data we eventually allocate to M_i may be less or more than $1/n$.

If the plan is the same as the estimated plan and the operator costs increase linearly with the data size, everything will work as is. However, since the input table sizes could be different from our assumption, the plan may change, and some multi-phase operators may need more or fewer passes to perform their tasks. Thus, the estimated times we used to quantify the performance differences among machines may be wrong.

The impact of the changes in plans and operator executions is twofold. When a plan with lower cost is selected or fewer passes are needed for an operator, the actual query runtime should be shorter than our estimate, leaving more room for improvement. When things change in the opposite direction, query execution times may be longer than expected, and we may place too much data on a machine. The latter case is an unfavorable situation that we should watch out for. We use the following strategies to avoid making a bad recommendation.

- **Detection**: before we actually adopt a partitioning recommendation, we involve the query optimizer again

to generate execution plans. We re-estimate query execution times when assuming that each machine gets the fraction of data as suggested by our model. We return a warning to the user, if we find that the new estimated workload runtime is longer than the old estimate. This approach works for both plan and phase changes.

- **Safeguard**: to avoid overloading a machine M_i, we can add a new constraint $p_i \leq p_{isafe}$ to our model. By selecting a suitable value for p_{isafe} as a guarding point, we can force the problem to stay in the region, where query execution times grow linearly with data size.

Even if additional passes are required for some operators, the data processing time of a powerful machine may still be shorter than that of a slow machine. One possible direction would be to use a mixed-integer program to fully exploit the potential of a powerful machine. Due to lack of space, we leave this as an interesting direction for future work.

5. EXPERIMENTAL EVALUATION

This section experimentally evaluates the effectiveness and efficiency of our proposed techniques.

5.1 Experimental Setup

We implemented and tested our techniques in SQL Server PDW. Our cluster consisted of 9 physical machines, and each machine had two quad-core processors, 16GB of main memory, and eight disks. On top of each physical machine, we created a virtual machine (VM) to run our database system. One VM served as a control node for our system, while the remaining eight were compute nodes. We artificially introduced heterogeneity by allowing VMs to use varying numbers of processors and disks, limiting the amount of main memory, and by "throttling" the network connection.

Table	Partition Key	Table	Partition Key
Customer	c_custkey	Part	p_partkey
Lineitem	l_orderkey	Partsupp	ps_partkey
Nation	(replicated)	Region	(replicated)
Orders	o_orderkey	Supplier	s_suppkey

Table 1: Partition keys for the TPC-H tables.

The parallel database system we ran consists of single-node DBMS instances connected by a distribution layer. We have eight instances of this single-node DBMS, each running in one of the VMs. The single-node DBMS is responsible for exploiting the resources within the node. We used a TPC-H 200GB database for our experiments. Each table was either hash partitioned or replicated across all compute nodes. Table 1 summarizes the partition keys used for the TPC-H tables. Replicated tables were stored at every compute node on a single disk.

5.2 Overall Performance

To test the performance of different data partitioning approaches, we used a workload of 22 TPC-H queries. By default, each VM used 4 disks, 8 CPUs, 1Gb/s network bandwidth, and 8GB memory. In our experiments, we created 6 different heterogeneous environments as summarized below to run the queries.

1. ***CPU-intensive configuration***: to make more queries CPU bound, we use as few CPUs as possible for the VMs. In this setting, we use just one CPU for half of the VMs, and two CPUs for the other half. As a result, CPU capacity of the fast machines is twice that of the slow machines.

2. ***Network-intensive configuration***: similarly, to make more queries network bound, we reduce network bandwidth for the VMs. For half of the VMs we set the bandwidth to 10 Mb/s and for the other half to 20 Mb/s.

3. ***I/O-intensive configuration (2)***: we reduce the number of disks that are used by the VMs. For half of the VMs we limit the number of disks to one and for the remainder to two.

4. ***I/O-intensive configuration (4)***: in this setting, we have 4 types of machines. We set the number of disks used by the VMs to 1, 1, 2, 2, 4, 4, 8, and 8, respectively. Note that the I/O speeds of the machines with 8 disks (the fastest machines) are roughly 4 times as fast as the I/O speeds of the machines with just 1 disk (the slowest machines), and the I/O speeds of the machines with 4 disks are roughly 3.2 times as fast as the I/O speeds of the slowest machines.

5. ***CPU and I/O-intensive configuration***: the number of disks used by the VMs is the same as in the above configuration, but we reduce their CPU capability. We set the number of CPUs that they use to 2, 4, 2, 4, 2, 4, 2, and 4, respectively. In this setting, all VMs are different. If we calculate a ratio to represent the number of CPUs to the number of disks for a VM, we can conclude that subqueries running on a VM with a small ratio tend to be CPU bound, while subqueries running on a VM with a large ratio tend to be I/O bound. We refer to this configuration as *Mix-2*.

6. ***CPU, I/O, and network-intensive configuration***: The CPU and I/O settings are the same as above. We also reduce network bandwidth to make some of the subqueries network bound. We set the bandwidth for the VMs in Mb/s to 30, 30, 30, 10, 10, 30, 30, and 30, respectively. We refer to this configuration as *Mix-3*.

For each heterogeneous cluster configuration, we evaluate the performance of the strategy proposed in this paper (we refer to it as Bricolage). We use Uniform and Delete as the competitors. In Table 2(a), we illustrate the predicted workload execution time for different approaches running with different cluster configurations. We also calculate the percentage of time that can be reduced compared to the Uniform approach. We load the data into our cluster using different data partitioning strategies to run the queries, and we measure the actual workload processing times and the improvements. In Table 2(b), we list the numbers we observe after running the workload. As we can see from the tables, although in some cases, our absolute time estimates are not very precise, the percentage improvement we achieve is close to our predictions. As a result, we can conclude that our model is reliable for making recommendations.

In the above cases, we varied only the number of disks, CPUs, and network bandwidth for the VMs. The impact of heterogeneous memory sizes was investigated in [19], and our results indicated that the two strategies proposed in Section

Strategy	CPU-intensive	Network-intensive	I/O-intensive (2)	I/O-intensive (4)	Mix-2	Mix-3
Uniform (sec)	5346	5628	5302	5583	6451	8709
Delete (sec)	5346 (0.0%)	5628 (0.0%)	5103 (3.7%)	3522 (36.9%)	4760 (26.2%)	8052 (7.5%)
Bricolage (sec)	4115 (23.0%)	4583 (18.6%)	3317 (37.4%)	2431 (56.5%)	3420 (47.0%)	5202 (40.3%)

(a) Estimated execution time and percentage of time reduction for different data partitioning strategies

Strategy	CPU-intensive	Network-intensive	I/O-intensive (2)	I/O-intensive (4)	Mix-2	Mix-3
Uniform (sec)	7371	8720	6037	6275	7680	11564
Delete (sec)	7371 (0.0%)	8720 (0.0%)	6581 (-9.0%)	4026 (35.8%)	6107 (20.5%)	9202 (20.4%)
Bricolage (sec)	6024 (18.3%)	7205 (17.4%)	4195 (30.5%)	3236 (48.4%)	5131 (33.2%)	5767 (50.1%)

(b) Actual execution time and percentage of time reduction for different data partitioning strategies

Table 2: Overall performance (22 TPC-H queries).

4.4 can effectively avoid overloading machines with scarce memory. In [19], we also presented (*i*) TPC-H query execution time comparison with different strategies, (*ii*) evaluations of the accuracy of query performance prediction, and (*iii*) studies on whether or not further improvements might be possible, if we had better system performance predictions. For (*i*), our results showed that Bricolage can provide the best performance compared to the other alternatives we have considered. For (*ii*) and (*iii*), we found that accurate relative performance prediction is critical for Resource Bricolage, and it may not be worth trying too hard to improve the absolute performance prediciton accuracy. Due to space constraints, the interested reader is referred to the paper for details.

6. RELATED WORK

Our work is related to query execution time estimation, which can be loosely classified into two categories. The first category includes the work on progress estimation for running queries [9, 18, 20, 22, 23, 26]. The key idea for this work is to collect runtime statistics from the actual execution of a query to dynamically predict the remaining work/time for the query. In general, no prediction can be made before the query starts. The debug run-based progress estimator for MapReduce jobs proposed in [28] is an exception. However, it cannot provide accurate estimates for queries running in parallel database systems [21]. On the other hand, the second category of work focuses on query running time prediction before a query starts [35, 36]. In [36], the authors proposed a technique to calibrate the cost units in the optimizer cost model to match the true performance of the hardware and software on which the query will be run, in order to estimate query execution time. This paper gave details on how to calibrate the five parameters used by PostgreSQL. However, different database optimizers may use different cost formulas and parameters. Additional work is required before we can apply the technique to other database systems.

Another related research direction is automated partitioning design for parallel databases. The work in [17] investigates different multi-attribute partitioning strategies, and it tries to place tuples that satisfy the same selection predicates on fewer machines. The work in [11, 25] studies three data placement issues: choosing the number of machines over which to partition base data, selecting the set of machines on which to place each relation, and deciding whether to place the data on disk or cache it permanently in memory. In [29, 31], the most suitable partitioning key for each table is automatically selected in order to minimize estimated costs, such as data movement costs. While these approaches can substantially improve system performance, they focus on base table partitioning and treat all machines in the cluster as identical. In our work, we aim at improving query performance in heterogeneous environments. Instead of always applying a uniform partitioning function to these keys, we vary the amount of data that will be assigned to each machine for the purpose of better resource utilization and faster query execution. The work in [12, 30] attempts to improve scalability of distributed databases by minimizing the number of distributed transactions for OLTP workloads. Our work targets resource-intensive analytical workloads where queries are typically distributed. An adaptive and query-workload-aware mechanism for partitioning large-scale spatial data is proposed in [6], while other systems for processing spatial data typically employ static data-partitioning structures that cannot adapt to data changes. An elastic partitioning framework is developed in [33] for distributed OLTP DBMSs, which automatically scales resources in response to demand spikes and gradual changes in an application's workload. Our work currently employs a static partitioning strategy, and dynamic data partitioning would be an interesting direction for future research.

Our work is also related to skew handling in parallel database systems [15, 37, 38]. Skew handling is in a sense the dual problem of the one that we deal with in the paper. It assumes that the hardware is homogeneous, but data skew can lead to load imbalances in the cluster. It then tries to level the imbalances that arise.

Finally, our paper is related to various approaches proposed for improving system performance in heterogeneous environments. The work in [40] proposed solutions to improve performance for latency-sensitive applications running on clusters with heterogeneous network connectivity. A suite of optimizations are proposed in [5] to improve MapReduce performance on heterogeneous clusters. Zaharia et al. [39] developed a scheduling algorithm to dispatch straggling tasks to reduce execution times of MapReduce jobs. Since a MapReduce system does not use knowledge of data distribution and location, our technique cannot be used to pre-partition the data in HDFS. However, we can apply our technique to partition intermediate data in MapReduce systems with streaming pipelines.

7. CONCLUSION AND FUTURE IDEAS

We studied the problem of improving database performance in heterogeneous environments. We developed a technique to quantify performance differences among machines with heterogeneous resources and to assign proper amounts of data to them. Extensive experiments confirm that our technique can provide good and reliable partition recommendations for given workloads with minimal overhead.

This paper lays down a foundation for several directions towards future studies to improve database performance running in the cloud. Our work aims at improving the performance of OLAP workload on heterogeneous clusters, where similar problems on performance prediction and performance optimization for OLTP workloads on heterogeneous environments remain open. While the focus of this work has been on static data partitioning strategies, a natural follow-up will be to study how to dynamically repartition the data at runtime, when our initial predictions were not accurate or system conditions have changed. The model proposed in the paper assumes that queries are running sequentially in the workload. Since queries might run concurrently in a system, the relatively short running ones might have negligible impact on the total execution time. One promising direction would be to take into account concurrent query execution and explicitly model how queries interact with each other to better utilize resources. As we mentioned in the introduction, previous research has revealed that the supposedly identical instances provided by a public cloud often exhibit measurable performance differences. In the cases where we are given a budget constraint or a time constraint, we will encounter a problem of selecting the most suitable computing resources for building a cluster. In addition to the performance prediction and data allocation challenges we are tackling in the paper, we also need to either (*i*) select a set of most suitable computing resources that minimize the total execution time for a given budget, or (*ii*) select a set of most suitable computing resources that minimize the cost for a given performance goal. We think both are very interesting directions for future research.

Acknowledgment

This research was supported by a grant from Microsoft Jim Gray Systems Lab, Madison, WI. We would like to thank everyone in the lab for valuable suggestions on this project.

8. REFERENCES

[1] Aster Data nCluster. http://download.101com.com/tdwi/ww29/Aster_Data_A_New_Architecture.pdf.

[2] Pivotal Greenplum. http://pivotal.io/big-data/pivotal-greenplum.

[3] SQL Server 2012 Parallel Data Warehouse. http://www.microsoft.com/en-ca/server-cloud/products/analytics-platform-system/.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. SIGMOD, 2004.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. ASPLOS, 2012.

[6] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: Adaptive query workload aware partitioning of big spatial data. *PVLDB*, 2015.

[7] C. Baru, G. Fecteau, A. Goyal, H.-I. Hsiao, A. Jhingran, S. Padmanabhan, W. Wilson, and A. G. H. i Hsiao. DB2 Parallel Edition. *IBM Systems Journal*, 1995.

[8] R. Buck. The Oracle Media Server for nCUBE Massively Parallel Systems. *Parallel Processing Symposium*, 1994.

[9] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for SQL queries? In *SIGMOD*, 2005.

[10] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD*, 2004.

[11] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. *SIGMOD Record*, 1988.

[12] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 2010.

[13] M. Dempsey. Monitoring active queries with Teradata manager 5.0. http://www.teradataforum.com/attachments/a030318c.doc, 2001.

[14] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *TKDE*, 1990.

[15] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. VLDB, 1992.

[16] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. SoCC, 2012.

[17] S. Ghandeharizadeh, D. J. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. SIGMOD, 1992.

[18] A. C. König, B. Ding, S. Chaudhuri, and V. Narasayya. A statistical approach towards robust progress estimation. *PVLDB*, 2012.

[19] J. Li, J. Naughton, and R. V. Nehme. Resource bricolage for parallel database systems. *PVLDB*, 2014.

[20] J. Li, R. V. Nehme, and J. F. Naughton. GSLPI: A cost-based query progress indicator. In *ICDE*, 2012.

[21] J. Li, R. V. Nehme, and J. F. Naughton. Toward progress indicators on steroids for big data systems. In *CIDR*, 2013.

[22] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a progress indicator for database queries. SIGMOD, 2004.

[23] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Increasing the accuracy and coverage of SQL progress indicators. In *ICDE*, 2005.

[24] D. Mangot. EC2 variability: The numbers revealed. http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed, 2009.

[25] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 1997.

[26] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. *ICDE*, 2007.

[27] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[28] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. ICDE, 2010.

[29] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. SIGMOD, 2011.

[30] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. SIGMOD, 2012.

[31] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. SIGMOD, 2002.

[32] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *PVLDB*, 2010.

[33] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB*, 2014.

[34] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon EC2 data center. INFOCOM, 2010.

[35] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 2013.

[36] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, 2013.

[37] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel DBMS. *PVLDB*, 2009.

[38] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. SIGMOD, 2008.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. OSDI, 2008.

[40] T. Zou, R. L. Bras, M. V. Salles, A. Demers, and J. Gehrke. ClouDiA: A deployment advisor for public clouds. PVLDB, 2013.