Resource Bricolage for Parallel Database Systems

Jiexing Li [#], Jeffrey Naughton [#], Rimma V. Nehme [*]
[#]University of Wisconsin, Madison [*]Microsoft Jim Gray Systems Lab
{jxli, naughton}@cs.wisc.edu 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. 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 under-utilize 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. To exploit data parallelism, these systems typically partition data among multiple machines. A query running on the systems is then broken up into subqueries, which are executed in parallel on the separate data chunks.

Nowadays, 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. For example, 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. At the same time, more and more parallel database systems 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 [12, 20, 29, 30].

Proceedings of the VLDB Endowment, Vol. 8, No. 1
Copyright 2014 VLDB Endowment 2150-8097/14/09.

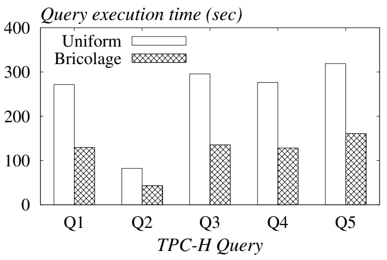


Figure 1: Query execution times with different data partitioning strategies.

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 database system is determined by its slowest machine. At worst, a 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 times of the slow ones. 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. 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.

Automated partitioning design for parallel databases is a fairly well-researched problem [8, 25, 26, 27]. The proposed approaches improve system performance by selecting the most suitable partitioning keys for base tables or minimizing the number of distributed transactions for OLTP workloads. Somewhat surprisingly, despite the apparent importance of this problem, no existing approach aims directly at minimizing decision support execution time for heterogeneous clusters. We will provide detailed explanations in Section 6.

1.2 Our Contributions

To improve performance of parallel database systems 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 workload execution time. To do this, we quantify differences among machines by using the 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) [1], 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.

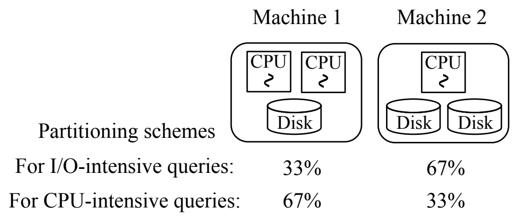


Figure 2: Different data partitioning schemes.

A uniform partitioning function may result in poor performance. Let us consider a simple example where we have two machines in a cluster as shown in Figure 2. Let the CPUs of the first machine be twice as fast as that of the second machine, and let the disks of the first machine be 50% slower than that of the second machine. We want to find the best data partitioning scheme to allocate the data to these two machines. Suppose that we have only one query in our workload, and it is I/O intensive. This query scans a table and counts the number of tuples in the table. The query completes when both machines finish their processing. To minimize the total execution time, it is easy for us to come up with the best partitioning scheme, which assigns 33% of the data to the first machine and 67% of the data to the second machine. In this case, both machines will have similar response times. Assume now that we add a CPU-intensive query to the workload. It scans and sorts the tuples in the table. Determining the best partitioning scheme in this case becomes a non-trivial task. Intuitively, if the CPU-intensive query takes longer to execute than the I/O-intensive query, we should assign more data to the first machine to take advantage of its more powerful CPUs, and vice versa.

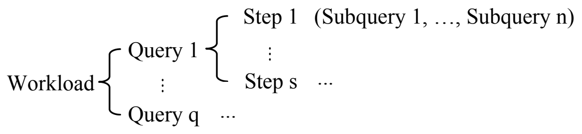


Figure 3: A query workload.

In general, we may have a set of heterogeneous machines with different disk, CPU, and network performance, and they may have different amounts of memory. At the same time, we have a workload with a set of SQL queries as shown in Figure 3. 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.

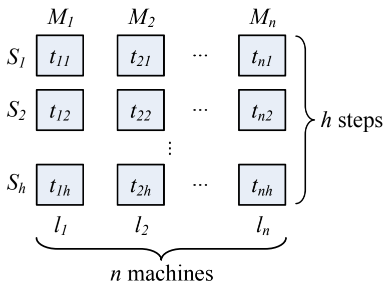


Figure 4: Problem setting.

We visually depict our problem setting in Figure 4. 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 graph 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 Potential for Improvement

Whether it is worth allocating data to machines in a non-uniform fashion is dependent on the characteristics of the available computing resources. If all the machines in a cluster are identical or have similar performance, there is no need for us to consider the resource bricolage problem at all. At the other extreme, if all the machines are fast except for a few slow ones, we can improve performance and come close to the optimal solution by just deleting the slow machines. The time that we can save by dealing with performance variability depends on many factors, such as the hardware differences among machines, the percentage of fast/slow machines, and the workloads.

To gain preliminary insight as to when explicitly modeling resource heterogeneity can and cannot pay off, we consider three data partitioning strategies: *Uniform*, *Delete*, and *Optimal*. Uniform is the default data allocation strategy of a parallel database system. It ignores differences among machines and assigns the same amount of data to each machine. Since there is no commonly accepted approach for the problem we address in the paper, we propose Delete as a simple heuristic that attempts to handle resource heterogeneity. It deletes some slow machines before it partitions the data uniformly to the remaining ones. It tries to delete the slowest set of machines first, and then the second slowest next. This process is repeated until no further improvement can be made. Optimal is the ideal data partitioning strategy that we want to pursue. It distributes data to machines in a way that can minimize the overall workload execution time. The corresponding query execution times for these strategies are denoted as t_u, t_{del}, and t_{opt}, respectively. According to the definitions, we have $t_u \geq t_{del} \geq t_{opt}$.

We start with a simple case with n machines in total, where a fraction p of them are fast and $(1-p)$ are slow. Our workload contains just one single-step query. For simplicity, we assume that one fast machine can process all data in 1 unit of time (e.g., 1 hour, 1 day, etc.), and the slow machines need r units of time ($r \geq 1$). We also assume that, for each machine, the processing time of a step changes linearly with the amount of data. The value r can also be considered to be the ratio between execution times of a slow machine and a fast machine. We omit the underlying reasons that lead to the performance differences (e.g., due to a slow disk, CPU, or network connection), since they are not important for our discussion here. It is easy to see that $t_u = \frac{1}{n}r$, $t_{del} = \min\{\frac{1}{n}r, \frac{1}{np}\}$. In this limited specialized case that we are considering, calculating t_{opt} is easy and can be conducted in the following way. We denote the fractions of data we allocate to a fast machine as p_1 and to a slow machine as p_2, respectively. The optimal strategy assigns data to machines in such a way that the processing times are identical. This can be represented as $p_1 = rp_2$. Since the sum of p_1 and p_2 is 1, we can derive $t_{opt} = \frac{r}{n(rp+1-p)}$.

To see how much improvement we can make by going from a simple strategy to a more sophisticated one, we calculate the percentage of time we can reduce from t_1 to t_2 as $100(1 - t_2/t_1)$. We discuss the reduction that can be made by adopting the simple heuristic Delete first, and then we present the further reduction that can be achieved by trying to come up with Optimal.

From Uniform to Delete. When $r \leq \frac{1}{p}$, we have $t_{del} = \frac{1}{n}r = t_u$. The decision is to keep all machines, and no improvement can be made by deleting slow machines. When $r > \frac{1}{p}$, $t_{del} = \frac{1}{np}$. The percentage of reduction we can make is $100(1 - \frac{1}{rp})$. When rp is big, the percentage of reduction can get close to 100%. Delete is well-suited for clusters where there are only a few slow machines and the more powerful machines are much faster than the slow ones. Thus, given a heterogeneous cluster, the first thing we should do is try to find the slow outliers and delete them.

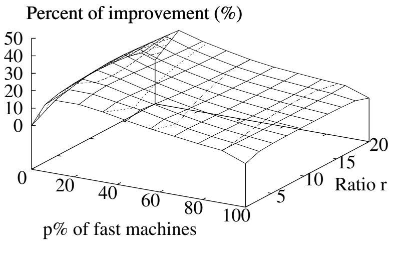


Figure 5: Potential for improvement.

From Delete to Optimal. In this case, the improvement we can make is not so obvious. In Figure 5, we plot the percentage of time that can be reduced from t_{del} to t_{opt}. We vary p from 0 to 100% and r from 0 to 20. As we can see from the graph, when r is fixed, the percentage of reduction increases at first and then decreases as p gets bigger. Similarly, when p is fixed, the percentage of reduction also increases at first and then decreases as we vary r from 0 to 20. More precisely, when $r \leq \frac{1}{p}$, $t_{del} = \frac{1}{n}r$. The percentage of reduction can be calculated as $100(1 - t_{opt}/t_{del}) = 100(1 - \frac{1}{rp+1-p})$. Since $rp \leq 1$, we have $rp + 1 - p < 2$. As a result, the reduction $100(1 - \frac{1}{rp+1-p})$ is less than 50%. When $r > \frac{1}{p}$, we have $t_{del} = \frac{1}{np}$, and the reduction is $100(1 - \frac{1}{1+\frac{1}{rp}-\frac{1}{r}})$. Since $rp > 1$, the denominator is no larger than 2. Therefore, the percent of reduction is also less than 50%.

Now, let us consider a more complicated example with n machines and $n/2 + 1$ steps. In this example, we will show that in the worst case, the performance gap between Delete and Optimal can be arbitrarily large. The detailed t_{ij} values are indicated in Figure 6, where a is large constant and ε is a very small positive number. If we use each machine individually to process the data, the workload execution time for a machine in the first half on the left is $a + (\frac{n}{2}+1)\varepsilon$. This is longer than the workload execution time $a + (\frac{n}{2} - 1)\varepsilon$ for a machine in the second half. When we look at these machines individually, the first $n/2$ of them are considered to be relatively slow.

	"slow" machines		*"fast" machines*		
	M_1	$M_{n/2}$	$M_{n/2+1}$	$M_{n/2+2}$	M_n
S_1	$a+\varepsilon$	$a+\varepsilon$	ε	ε	ε
S_2	ε	ε	$a-\varepsilon$	ε	ε
S_3	ε	ε	ε	$a-\varepsilon$	ε
	...			$\ddots$	
$S_{n/2+1}$	ε	ε	ε	ε	$a-\varepsilon$

Figure 6: A worst-case example.

Given these machines, Delete works as follows. First, it calculates the execution time of the workload when data is partitioned uniformly across all machines. The runtime for the first step S_1 is $\frac{1}{n}(a+\varepsilon)$. The runtime for a later step S_j ($j \geq 2$) is $\frac{1}{n}(a-\varepsilon)$, which is the processing time of machine $M_{n/2+j-1}$. In total, we have $n/2$ number of such steps. As a result, the execution time of all steps is $\frac{1}{n}(a+\varepsilon) + \frac{1}{2}(a-\varepsilon)$. Then Delete tries to reduce the execution time by deleting slow machines, thus it will try to delete $\{M_1, M_2, ..., M_{n/2}\}$ first. We can prove that the best choice for Delete is to use all machines. On the other hand, the optimal strategy is to use just the "slow" machines and assign $\frac{2}{n}$ of the data to each of them, and we have $t_{opt} = \frac{2}{n}(a + \varepsilon)$. Although Delete uses more machines than Optimal, it is easy to get that $\frac{t_{del}}{t_{opt}} \approx \frac{n}{4}$.

2.3 Challenges

Although the worst-case situation may not happen very often in the real world, our main point here is that when there are many different machines in a cluster and we have queries with various resource demands, the heuristic (Delete) that works well for simple cases may generate results far from optimal. In addition, the heuristic works by deleting the set of obviously slow machines. However, simple cases where we can divide machines in the same cluster into a fast group and a slow group may not happen very often. According to Moore's law, computers' capabilities double approximately every two years. If cluster administrators perform hardware upgrades every one or two years, it is reasonable to assume that we may see 2x, 4x, or maybe 8x differences in machine performance in the same cluster. This assumption is also consistent with what has been observed in a very large Google cluster [28]. Normally, we would not add a machine to a cluster that is significantly different from the others to perform the same tasks. On the other hand, machines that are too slow and out of date will be eventually phased out. For systems running on a public cloud, requesting a set of VM instances of the same type to run an application is the most common situation. As we discussed in Section 1, the supposedly identical instances from public clouds may still have different performance. Previous studies, which used a number of benchmarks to test the performance of 40 Amazon EC2 m1.small instances, observed that the speedup of the best performance instance over the worst performance instance is usually in the range from 0% to 300% for different resources [12].

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, this is not true in the worst-case example we discussed. In our example, the performance of the set of "slow" machines used by Optimal are similar, and the bottlenecks of the subqueries are clustered on the same step (S_1). Delete uses some additional "fast" machines, but these machines do not collaborate well in the system. They introduce additional bottlenecks in other steps (S_2 to $S_{n/2+1}$), which result 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 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 [5, 6, 16, 18, 23]. Unlike previous work, we do not need to get perfect time estimates to make a good data partitioning recommendation. As we will see in the experimental section, the ratios in time between machines are the key information that we need to deal with heterogeneous resources. 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 the optimizer, and we 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 [6, 18], 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.

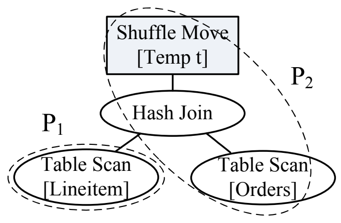


Figure 7: An execution plan with two pipelines.

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 7 is divided into two different pipelines P_1 and P_2. Pipelines are executed one after another. If we can estimate the execution time for each pipeline, the total runtime of a query is simply 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 [16] to calculate the cost for a pipeline, and the interested reader is referred to that paper for details.

However, 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 indication of 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 was 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 test 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. For this query, we can also get its estimated CPU cost and runtime, and we calculate the CPU speed for this machine in a similar way. 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 small and 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 when 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 resemble 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 7. 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 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 7 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 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 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 below.

For step S_j, since the execution time x_j is the longest execution time of all machines, we must have $p_i t_{ij} \le x_j$ 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 derive the values for p_is ($0 \le i \le n$) and x_js ($0 \le j \le h$), where the set of p_i values represents 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

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

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.

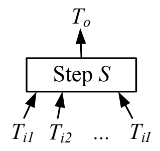


Figure 8: The input and output tables for a step.

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 8. 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 go to different machines[1].

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 to partition the tables in the 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 [2, 25, 27]. 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:

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.

[1]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.

(a) For the input tables to S, find the groups that they belong to. If more than one group is found, we ***merge*** them into a single group.

(b) ***Assign*** S to the group.

(c) ***Create*** a new group with the output table of S.

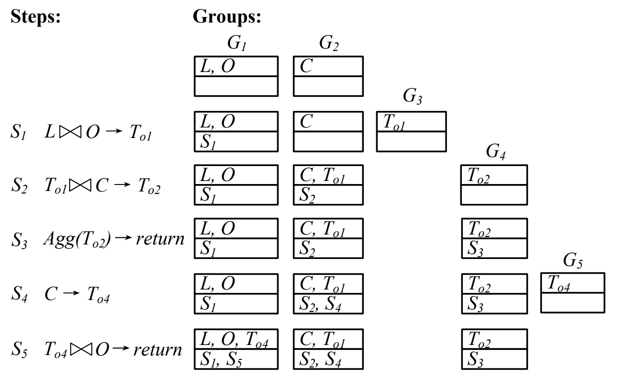


Figure 9: Example of distribution-compatible group generation.

We go through a small example shown in Figure 9 to demonstrate how it works. The example has only five steps and three base tables: L, O, and C, where L and O are distribution-compatible according to the physical design. First, we create two groups G_1 and G_2 for the base tables, and L and O belong to the same group G_1. Then for each step in the workload, we perform the three instructions (a) to (c) as described above. Step S_1 joins L and O from the group G_1. Since both of them belong to the same group, there is no need to merge. We assign step S_1 to group G_1 to indicate that the partitioning of the tables in G_1 has a significant impact on the runtime of S_1. A new group G_3 is then created for the output table T_{o1} of S_1. No query step has been assigned to the new group yet, since we do not know which step(s) will use T_{o1}. S_2 will then be processed. Since S_2 joins T_{o1} in G_3 with table C in G_2, we merge G_3 with G_2. We do this by inserting every element in G_3 into G_2. We then assign S_2 to the group that contains tables C and T_{o1}, and we create a new group G_4 for T_{o2}. At step S_3, a local aggregation on T_{o2} is performed, and the result is returned to the user. Thus we assign S_3 to group G_4. After all steps are processed, we get three groups for this workload.

For each distribution-compatible group generated, we can employ the linear model proposed above to obtain a partitioning scheme for the tables to minimize total runtime of the steps in the group.

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. However, as we will see later in our experiments, the assumption is valid in many cases, and even when it does not strictly hold, it is a reasonable heuristic that yields good performance.

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 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 $1/n$.

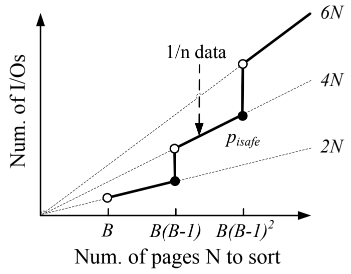


Figure 10: I/O cost for Sort.

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. We use the I/O cost for Sort as our running example, and the I/O cost for Hash Join is similar. To sort a table with N pages using B buffer pages, the number of passes for merging is $\lceil \log_{B-1}\lceil N/B\rceil\rceil$. In each pass, N pages of data will be written to disk and then brought back to memory. The number of I/Os for Sort[2] can be calculated as $2N\lceil \log_{B-1}\lceil N/B\rceil\rceil$, and we plot this nonlinear function in Figure 10. The axes are in log scale. As we can see from the graph, for a multi-phase operator like Sort, by making a linear assumption, we will stick with a particular linear function (e.g., $4N$ in the graph) for predicting the time. 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

[2]We assume that the I/Os for generating the sorted runs are done by a scan operator, and we omit the cost here.

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. For the example shown in Figure 10, we can use the value of the second dark point as p_{isafe}, to prevent data processing time from growing too fast.

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.

It is worth noting that a linear region spans a large range. For a sort operator with x passes, the range starts at $B(B-1)^{(x-1)}$ and ends at $B(B-1)^x$. The end point is $B-1$ times as large as the start point. B is typically a very large number. For example, if the page size is 8KB, an 8MB buffer pool consists of 1024 pages. Thus, introducing one more pass is easy if the assumed $1/n$ of the data happens to be close to an end point. To introduce two more passes, we need to assign at least 1000 times more data to a machine. Meanwhile, we typically will not assign so much more data to a machine, since the performance differences among machines in our problem are usually not very big (e.g., no more than 8x).

5. EXPERIMENTAL EVALUATION

This section experimentally evaluates the effectiveness and efficiency of our proposed techniques. Our experiments focus on whether we can accurately predict the performance differences among machines, and whether we are able to achieve the estimated improvements provided by our model. We also evaluate our technique's ability to handle situations where data processing times increase faster than linear.

5.1 Experimental Setup

We implemented and tested our techniques in SQL Server PDW. Our cluster consisted of 9 physical machines, which were connected by a 1Gbit HP Procurve Ethernet switch. Each machine had two 2.33GHz Intel E5410 quad-core processors, 16GB of main memory, and eight SAS 10K RPM 147GB 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 DBMSs connected by a distribution layer, and 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 (e.g., multiple cores and disks), however, this is transparent to the parallel distribution layer. 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 the first set of experiments, we created 6 different heterogeneous environments as summarized below to run the queries. In these cases, we vary only the number of disks, CPUs, and the network bandwidth for the VMs. We will study the impact of heterogeneous memory in a separate subsection later.

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. We set the bandwidth for half of them 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. We limit the number of disks used for half of them 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

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).

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, since to the best of our knowledge, there are no previously proposed solutions in the literature. The improvement in execution time due to our bricolage techniques depends on differences among machines. For each cluster configuration, we first measure the processing speeds for each machine using the profiling queries and the network test program described in Section 3. For a given machine, the data we use to measure its I/O speed is a 50MB Customer table, and the data we use to measure its CPU speed is a 2GB Lineitem table. We then generate execution plans for the queries in our workload assuming uniform partitioning, and we estimate the processing times for these plans running on different machines (the estimated t_{ij} values). These values are then used as input parameters for both Delete and Bricolage. For machine M_i, Delete sums up all its t_{ij} values and uses the summation as its score. Delete then tries to delete machines in descending order of their scores until no further improvements can be made. We then estimate the new query execution times for Delete where only the remaining machines are used. For our approach, we use the t_{ij} values together with the l_i values (determined by storage limits) as input to the model, and then we solve the linear program using a standard optimization technique called the simplex method [9]. The model returns a recommended data partitioning scheme together with the targeted workload execution time. 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 table, 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 Figure 1, we show the execution times of the first 5 TPC-H queries (Q_1 to Q_5) running with the I/O-intensive (4) configuration. The percentages of data that Bricolage

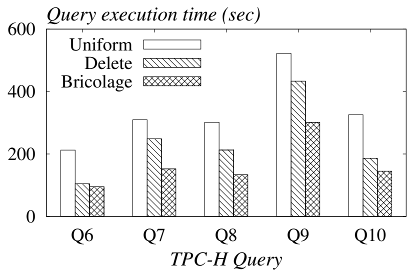


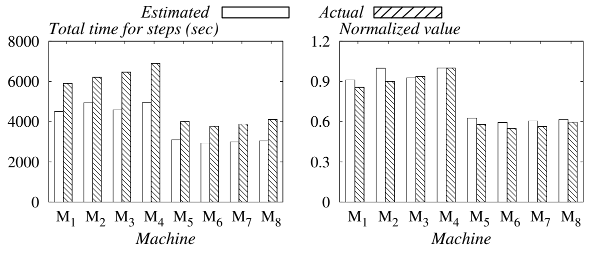
Figure 11: Query execution time comparison.

allocates to the 8 machines are 5.6%, 4.2%, 9.9%, 9.8%, 14.1%, 14.4%, 21.2%, and 20.8%, respectively. In Figure 11, we show the results for the next 5 TPC-H queries (Q_6 to Q_{10}) along with the results for Delete. Compared to Uniform, Delete reduces query execution times by removing the slowest machines (the bottleneck) with just one disk. For Q_6, Delete and Bricolage have similar performance, since this query moves a lot of data to the control node, which is the bottleneck when data is partitioned using these two strategies. For other queries, Bricolage can further reduce query execution times by fully utilizing all the computing resources.

5.3 Execution Time Estimation

In our work, we quantify differences among machines using data processing times (the $t_{ij}s$). Thus, we want to see whether our estimated times truly indicate the performance differences. For each machine in the cluster, we sum up its estimated and actual execution times for all steps. In Figure 12(a), we plot the results for the CPU-intensive configuration. In this case, the estimated workload execution time is 5346 seconds, which is shorter than the actual execution time of 7371 seconds. From the graph, we can see that the estimated times for all machines are consistently shorter than the corresponding actual execution times. If we pick the machine with the longest actual processing time (M_4 in the graph) and use the actual (estimated) time for it to normalize the actual (estimated) times for other machines, we get the normalized performance for all machines as shown in Figure 12(b). Ideally, we hope that for each machine its normalized estimated value is the same as the actual value. Although our estimates are not perfect, the errors we make when predicting relative performance differences are much smaller than when predicting absolute performance.

From Figure 12(b), we can also see that we underestimate performance for some machines (e.g., M_2) while overesti-



(a) Sum of time (b) Normalized time

Figure 12: Performance predictions for machines.

mate performance for some others (e.g., M_3). In this case, we will assign an inadequate amount of data to the underestimated machines and too much data to the overestimated ones, which leads to performance degradation. As a result, the actual improvement we obtained is usually smaller than the predicted improvement.

In our experiments, we found that the estimated CPU and network speeds tend to be slightly faster than the speeds we observed when running the workload. Since the queries in our workload are more complicated than the profiling queries we used to measure the speeds, we suspect that the actual processing speeds slow down a bit due to resource contention. But since we use the same approach (e.g., the same query/program) to measure the speeds for all machines, we introduce the same errors for them, consistently. As a result, we can still obtain reasonable estimates for relative performance.

5.4 Investigating Optimal Improvements

The experiments presented up until now demonstrate that the actual improvements we obtain are close to our predicted improvements. However, this does not tell us whether or not further improvements might be possible if we had better system performance predictions. In this section we explore this issue. Our goal is not to provide a better technique; rather, it is to evaluate the gap between our technique and the optimal, perhaps to shed light on remaining room for further improvement.

We try to derive the best possible improvements by using information obtained from actual runs of the queries to get more accurate t_{ij} estimates. For the pipelines that do not transfer any data to other machines, their processing times are determined only by the performance of the machine on which they run, and we know their actual execution times, and we can replace our estimated values with the actual values. However, for a pipeline which transfers data to other machines, the execution time we observe in an actual run may also be determined by the processing speeds of other machines. For this kind of pipeline, it may be hard to get the processing time that is independent of the other machines, and we have to use our estimated value. However, we can still try to improve the estimates by using actual query plans and actual cardinalities. In our experiment, we found that for the 4 configurations without network-intensive pipelines, the other machines have negligible impact on the execution time of a pipeline running on a specific machine. Thus, we have very accurate t_{ij} values for these 4 cases. However, the impact of other machines on the execution time of a network-bound pipeline is very obvious.

We use these updated t_{ij} values as input to our model, and we calculate the percentage of time that can be reduced for the 6 cases (we refer to this method as Optimal-a later). The new estimated time reductions are shown in Table 3. If we compare these values with the actual improvements we made, we find that they are close. Based on this investigation, we suspect that it is not worth trying too hard to improve the t_{ij} estimates.

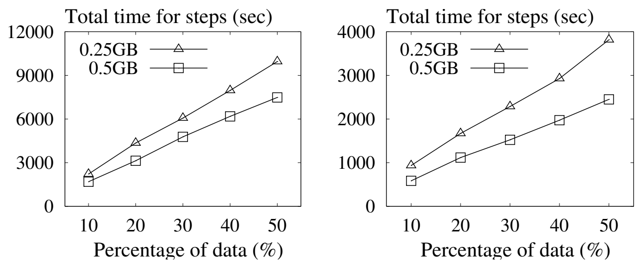
Configuration	Est. reduction	Act. reduction
CPU-intensive	20.6%	18.3%
Network-intensive	22.1%	17.4%
I/O-intensive (2)	32.3%	30.5%
I/O-intensive (4)	51.2%	48.4%
Mix-2	41.1%	33.2%
Mix-3	42.7%	50.1%

Table 3: Estimated time reductions using actual runs.

5.5 Handling Nonlinearity

The method we use to handle nonlinearity is based on the hypothesis that available memory changes processing time by changing the number of passes needed by multi-phase operators, and there are linear regions for these operators that are determined by the number of phases required.

To test whether linear regions exist along with the number of passes needed, we test how data processing time changes with data size. The cluster is configured with the I/O-intensive (4) setting. We set the memory size of the last machine to 0.25GB or 0.5GB, and we vary the amount of data assigned to it from 10% to 50%. The memory sizes of the other machines are set to 8GB, respectively, and they evenly share the remaining data. We sum up the time to process all steps for the last machine and plot the results in Figure 13(a). In both cases, the total time increases linearly with data size. When memory size is 0.5GB, all memory-consuming operators need no more than one pass, and when the memory size is 0.25GB, some operators need two passes. Since these operators do not change the number of passes required when we vary data size, they stay in regions where processing time grows linearly. Furthermore, when memory size is 0.25GB (2 passes are needed), the line should also have a steeper slope. To see this more clearly, we plot the results in Figure 13(b) for a only subset of the most memory-consuming queries.



(a) All TPC-H queries (b) Memory-consuming queries

Figure 13: Execution time vs. data size.

Based on our observations, to assign the proper amount of data to a machine, we need to estimate the execution time for a query accurately with different memory sizes, and we also need to use the corresponding estimate when the execution goes to a phase with a different number of passes.

For the system that we worked with, our technique is effective when no more than one pass is needed. Take the I/O-intensive configuration as an example. We set the DBMS memory size to 0.5GB (where no operator needs more than one pass) for the last machine and 8GB for other machines to repeat the experiment. The predicted and actual time reductions for our approach are 53.5% and 46.7%, respectively. The time estimates for the last machine correctly represent its performance differences compared to other machines, and thus less data is assigned to it compared to its original configuration with 8GB memory.

Strategy	Bricolage-d	Bricolage-g	Optimal-a
Est. reduction	53.1%	52.5%	46.7%
Act. reduction	35.2%	44.1%	44.9%

Table 4: Percentage of time reductions when memory size is 0.25GB for the last machine.

However, when memory is really scarce and more than one pass is required, the I/O cost estimates provided by our system are no longer accurate. Our predicted times are usually smaller than the actual processing times. In the first column of Table 4, we show the estimated and actual reductions in time for our default approach without guarding points (we refer to it as Bricolage-d in the table), when the DBMS memory size is set to 0.25GB for the last machine. This is a really adversarial situation, since the last machine has the most powerful disks to accomplish more I/O work while at the same time, it does not have enough memory to accommodate the data. The actual performance we obtained is much worse than our prediction, since we assign too much data to the last machine.

We have proposed two strategies in Section 4.4 for handling this: issuing a warning or using guarding points. In the above case, after we use Bricolage-d to provide an allocation recommendation, we estimate the input size $|S|$ for each memory-consuming operator as if data were partitioned in the suggested way. We found that some operators need two passes based on the estimated input table sizes and available memory. Thus, we can issue a warning saying that we are not sure about our estimate this time. Another approach denoted as Bricolage-g is to use guarding points. For machine M_i, we calculated a p_{isafe} value, to ensure that as long as the data allocated to M_i is no more than p_{isafe}, no operator needs more than one pass. As we can see from the table, by using guarding points, our estimate is now more accurate. We also investigate the optimal improvement for this case by using information derived from actual runs as input parameters to the model. The results are shown in the last column of Table 4. Although the actual reductions for Bricolage-g and Optimal-a are similar here, in general, an approach that uses true performance for machines can better exploit their capabilities. As a result, we leave accurate time estimation for memory-consuming operators as our future work.

5.6 Overhead of Our Solution

Our approach needs to estimate the processing speeds for machines, estimate plans and their execution times, and solve the linear model. Here, we describe the overheads involved. In our experiments, we used 2 minutes each to test the I/O and the CPU speeds for a machine. This can be done on all machines concurrently. We used 30 seconds to test the network speed for a machine, but another fast machine is required for sending/receiving the data. In the worst case, where we use just one fast machine to do the test, we need $0.5n$ minutes to test all n machines. We think this overhead is sufficiently small. For example, we need only 50 minutes to test the network speeds for 100 machines. For the complex analytical TPC-H workload, the average time to generate plans and estimate processing times for a query is 2.3 seconds. Thus the expected total time to estimate the performance parameters for a workload is $2.3|W|$, where $|W|$ is the number of queries in the workload. After we get all the estimates, the linear program can be solved efficiently. For example, for a cluster with 100 machines of 10 different kinds, and a workload with 100 queries, the linear program solver returns the solution in less than 3 seconds.

6. RELATED WORK

Our work is related to query execution time estimation, which can be loosely classified into two categories. The first category includes work on progress estimation for running queries [5, 15, 16, 18, 19, 22]. 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 [24] is an exception. However, it cannot provide accurate estimates for queries running on database systems [17]. On the other hand, the second category of work focuses on query running time prediction before a query starts [4, 11, 13, 31, 32]. In [32] 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 about 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. Usage of machine-learning based techniques for the estimation of query runtime has been explored in [4, 11, 13]. One key limitation of these approaches is that they do not work well for new "ad-hoc" queries, since they usually use supervised machine learning techniques.

Another related research direction is automated partitioning design for parallel databases. The goal of a partitioning advisor is to automatically determine the optimal way of partitioning the data, so that the overall workload cost is minimized. The work in [14] investigates different multi-attribute partitioning strategies, and it tries to place tuples that satisfy the same selection predicates on fewer machines. The work in [7, 21] 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 [25, 27], 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 [8, 26] 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.

Our work is also related to skew handling in parallel database systems [10, 33, 34]. 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 [3, 35]. A suite of optimizations are proposed in [3] to improve MapReduce performance on heterogeneous clusters. Zaharia et al. [35] develop 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

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. Previous research has revealed that the supposedly identical instances provided by a public cloud often exhibit measurable performance differences. One interesting problem is to select the set of most cost-efficient instances to minimize the execution time of a workload. While the focus of this work has been on static data partitioning strategies, the natural follow-up will be to study how to dynamically repartition the data at runtime, when our initial prediction was not accurate or system conditions change.

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

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

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

[4] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. ICDE, 2012.

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

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

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

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

[9] G. B. Dantzig and M. N. Thapa. *Linear Programming 1: Introduction*. Springer-Verlag, 1997.

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

[11] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. SIGMOD, 2011.

[12] 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.

[13] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. ICDE, 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. SoCC, 2012.

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

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

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

[32] 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.

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

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

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