The VLDB Journal (2017) 26:31–54
DOI 10.1007/s00778-016-0435-4

SPECIAL ISSUE PAPER

Resource bricolage and resource selection for parallel database systems

Jiexing Li[1] · **Jeffrey F. Naughton**[2] · **Rimma V. Nehme**[3]

Received: 15 December 2015 / Revised: 10 May 2016 / Accepted: 14 June 2016 / Published online: 25 June 2016

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. Performance differences among machines in the same cluster pose new challenges for parallel database systems. First, 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. Second, since machines might have varying resources or performance, different choices of machines may lead to different costs or performance for executing the same workload. By carefully selecting the most suitable machines for running a workload, we may achieve better performance with the same budget, or we may meet the same performance requirements with a lower cost. We address these challenges by introducing techniques we call *resource bricolage* and *resource selection* that improve database performance in heterogeneous environments. Our approaches quantify the performance differences among machines with various resources as they process workloads with diverse resource requirements. For the purpose of better resource utilization, 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. For the purpose of better resource selection, we formalize two problems: One minimizes the total workload execution time with a given budget, and the other minimizes the total budget with a given performance target. We then employ different mixed-integer programs to search for the optimal resource selection decisions. We verify the effectiveness of both resource bricolage and resource selection techniques with an extensive experimental study.

Keywords Resource bricolage · Resource selection · Parallel database systems · Heterogeneous clusters · Performance prediction · Data partitioning

✉ Jiexing Li
jiexing@google.com

Jeffrey F. Naughton
naughton@cs.wisc.edu

Rimma V. Nehme
rimman@microsoft.com

[1] Google Inc, Mountain View, CA, USA

[2] Department of Computer Sciences, University of Wisconsin, Madison, Madison, WI, USA

[3] Microsoft Jim Gray Systems Lab, Madison, WI, USA

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 [13,22,33,35].

1.1 Motivation

Performance differences among machines (either physical or virtual) in the same cluster pose new challenges for parallel database systems. In this paper, we discuss and address two such challenges.

Fully utilize cluster resources 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 underutilized, 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 Fig. 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 Sect. 4. The detailed cluster setup is described in Sect. 5. As can be seen from the graph, we can significantly reduce total query execution time by carefully partitioning the data.

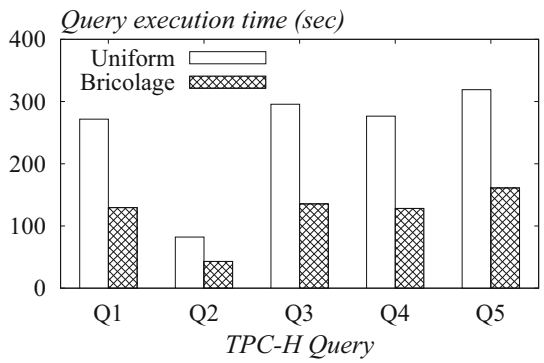

Fig. 1 Query execution times with different data partitioning strategies

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 [8,27,30,31]. 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 Sect. 9.

Fully utilize budget When a new cluster is built or when an old cluster is upgraded, there are various machines that we can choose from. By carefully selecting the most suitable machines for running a workload, we may achieve better performance with the same budget, or we may achieve the same performance requirements with a lower cost. To help customers deploy applications in the cloud, previous research has conducted performance evaluations or designed algorithms to seek virtual machine instances with better performance [13,28,29,41–43]. For example, customers can over-allocate instances and then terminate those instances with bad performance to optimize their cloud usage. By selecting better performing instances to complete the same task, cloud users can save up to 30 % of their total costs [28,29].

However, previous research has been focused on relatively simple workloads, such as workloads with a single bottleneck resource. For example, the work in [42] aims at only latency-sensitive applications where the response time of a service request largely depends on network connectivity between instances. As we will discuss in Sect. 2, our targeted workloads consist of SQL queries running in a parallel database system, which may be decomposed into a number of steps with different CPU, I/O, or network requirements. Thus, a decision that is made purely based on performance

evaluations of a single type of resource may result in poor performance. To select the best set of computing resources to process a workload, we must take into account the following three aspects simultaneously: First, we should select machines that are "suitable" for processing the workloads. In other words, we prefer machines that can process the workloads fast. Second, the selected machines should "collaborate" well in the same cluster (we will discuss an example where the set of fast machines do not collaborate well in Sect. 2.3). Finally, we should allocate data to the selected machines in a way that minimizes execution time. Unfortunately, the first two aspects sometimes can be contradictory. As we will see in the example in Sect. 2.3, a machine that is most suitable for our workloads when used individually may not collaborate well with other machines in the cluster. Thus, an optimal solution must balance all three factors for the best performance.

1.2 Our contributions

Resource bricolage To better utilize resources in a cluster, 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 Sect. 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) [34], and our experimental results show the effectiveness of our proposed solution.

Generalizations Resource bricolage can be generalized to a variety of different resource selection problems. In Sect. 6, we discuss two such problems. In these two more general problems, we do not know which machines are going to be used in the cluster. Our goal is to select the most suitable computing resources with budget constraints or time constraints. More specifically, in addition to the two challenges facing by *resource bricolage*, we also need to either (i) select a set of computing resources that minimizes the total execution time for a given budget, or (ii) select a set of computing resources that minimizes the budget for a given performance target.

We first formally define the resource selection problems and prove their hardness. As with our resource bricolage technique, we quantify differences among machines and formalize the two resource selection problems as optimization problems. We then formulate the problems as different mixed-integer programs (MIPs) to efficiently search for the optimal solutions. We finally solve the programs using a standard linear program solver to obtain both the resource selection decisions (e.g., which machines to use) and the data allocation decisions (e.g., the amounts of data allocated to selected machines). We compare the performance differences of our approaches and other alternatives with synthetic experiments that simulate different real-world scenarios, and we analyze various use cases to illustrate when a simple heuristic solution is effective and when a sophisticated solution is needed. Our experiments suggest that a solution that combines both data allocation and resource selection can yield significant performance improvement over other alternatives.

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 resource bricolage solution. Section 6 formalizes the resource selection problems and proves the NP-hardness of the problems. Section 7 presents our solutions for selecting resources. Section 8 evaluates the performance of different approaches for resource selection. Section 9 briefly reviews the related work. Finally, Sect. 10 concludes the paper with directions for future work.

2 The resource bricolage 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.

A uniform partitioning function may result in poor performance. Let us consider a simple example where we have two

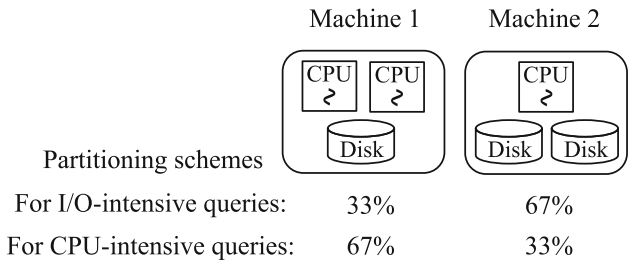

Fig. 2 Different data partitioning schemes

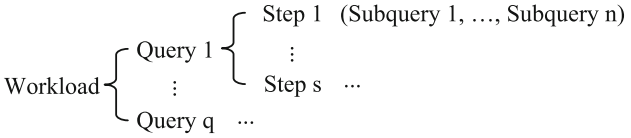

Fig. 3 A query workload

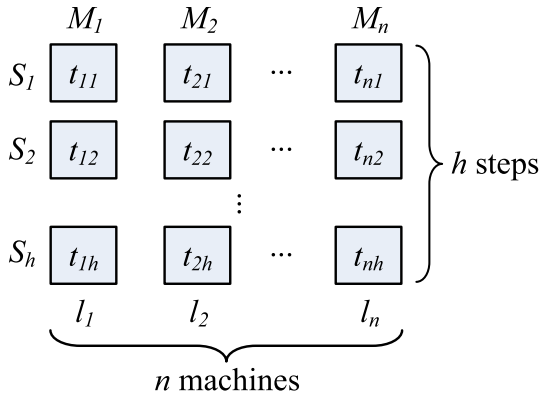

Fig. 4 Problem setting

machines in a cluster as shown in Fig. 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.

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 Fig. 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.

We visually depict our problem setting in Fig. 4. Let $M_1, M_2, \ldots, M_n$ 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: $S_1, S_2, \ldots, S_h$. 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), 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 $t_u = \frac{1}{n}r$, $t_{del} = \min\left\{\frac{1}{n}r, \frac{1}{np}\right\}$. 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 $100\left(1 - \frac{1}{rp}\right)$. 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.

From delete to optimal In this case, the improvement we can make is not so obvious. In Fig. 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

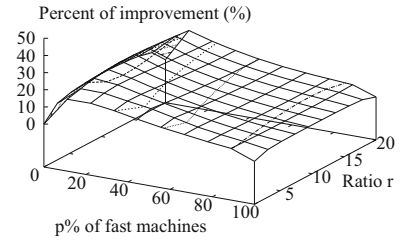

Fig. 5 Potential for improvement

	"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
	...			⋱	
$S_{n/2+1}$	ε	ε	ε	ε	$a-\varepsilon$

Fig. 6 A worst-case example

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 $100(1 - t_{opt}/t_{del}) = 100\left(1 - \frac{1}{rp+1-p}\right)$. Since $rp \leq 1$, we have $rp + 1 - p < 2$. As a result, the reduction $100\left(1 - \frac{1}{rp+1-p}\right)$ is less than 50 %. When $r > \frac{1}{p}$, we have $t_{del} = \frac{1}{np}$, and the reduction is $100\left(1 - \frac{1}{1+\frac{1}{rp}-\frac{1}{r}}\right)$. 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 Fig. 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 $a + \left(\frac{n}{2} + 1\right)\varepsilon$. This is longer than the workload execution time $a + \left(\frac{n}{2} - 1\right)\varepsilon$ 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.

Given these machines, Delete works as follows. First, it calculates the execution time of the workload when data are 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, and thus, it will try to delete $\{M_1, M_2, \ldots, M_{n/2}\}$ 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 $\frac{t_{del}}{t_{opt}} \approx \frac{n}{4}$. Note that the performance of Delete could be even worse if the actual t_{ij} values are not known and the good machines are deleted. As shown in the experimental section, Delete could degrade the performance due to errors in execution time estimation.

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 $2\times, 4\times$, or maybe $8\times$ differences in machine performance in the same cluster. This assumption is also consistent with what has been observed in a very large Google cluster [32]. 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 Sect. 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 [13].

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 [5,6,18,20,25]. 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 [6,20], 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, and Hash Join, a parallel database system also employs data movement operators, which are used for transferring data between DBMS instances running on different machines.

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 Fig. 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

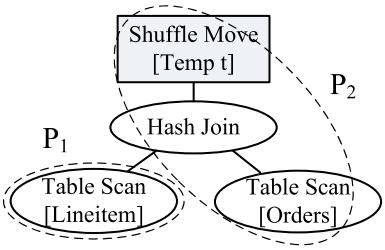

Fig. 7 An execution plan with two pipelines

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 [18] 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. We need to make sure that the query has a negligible I/O factor when calculating the CPU speed. 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. We repeat the measurement 3 times and take the average as the estimate.

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 cannot 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 technique

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 Fig. 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 Fig. 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 Sect. 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 $p_i t_{ij} \leq x_j$ 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 derive 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 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.

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

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.

For step S in workload W, let $\{T_{i1}, T_{i2}, \ldots, T_{iI}\}$ be the set of its input tables and T_o be its output table as we show in Fig. 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 parti-

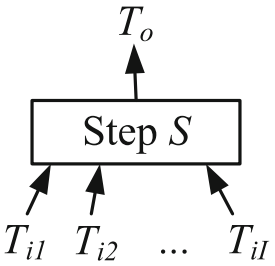

Fig. 8 The input and output tables for a step

tioning 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 [1,27,31]. 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:

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 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.

We go through a small example shown in Fig. 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 group 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

[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.

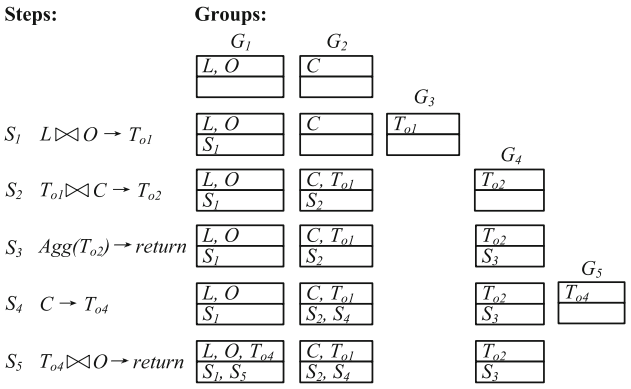

Fig. 9 Example of distribution-compatible group generation

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

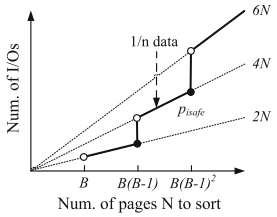

Fig. 10 I/O cost for Sort

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. 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 Fig. 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 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 Fig. 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 8 KB, an 8 MB 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 8×).

5 Experimental evaluation for resource bricolage

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 1 Gbit HP Procurve Ethernet switch. Each machine had two 2.33 GHz Intel E5410 quad-core processors, 16 GB of main memory, and eight SAS 10K RPM 147 GB 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.

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

Table 1 Partition keys for the TPC-H tables

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

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 200 GB 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, 1 Gb/s network bandwidth, and 8 GB 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 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 Sect. 3. For a given machine, the data we use to measure its I/O speed are a 50 MB Customer table, and the data we use to measure its CPU speed are a 2 GB 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 2a, we illustrate the predicted workload execution time for different approaches running with different

Table 2 Overall performance (22 TPC-H queries)

Strategy	CPU-intensive	Network-intensive	I/O-intensive (2)	I/O-intensive (4)	Mix-2	Mix-3
(a) Estimated execution time and percentage of time reduction for different data partitioning strategies						
Uniform (s)	5346	5628	5302	5583	6451	8709
Delete (s)	5346 (0.0 %)	5628 (0.0 %)	5103 (3.7 %)	3522 (36.9 %)	4760 (26.2 %)	8052 (7.5 %)
Bricolage (s)	4115 (23.0 %)	4583 (18.6 %)	3317 (37.4 %)	2431 (56.5 %)	3420 (47.0 %)	5202 (40.3 %)
(b) Actual execution time and percentage of time reduction for different data partitioning strategies						
Uniform (s)	7371	8720	6037	6275	7680	11,564
Delete (s)	7371 (0.0 %)	8720 (0.0 %)	6581 (−9.0 %)	4026 (35.8 %)	6107 (20.5 %)	9202 (20.4 %)
Bricolage (s)	6024 (18.3 %)	7205 (17.4 %)	4195 (30.5 %)	3236 (48.4 %)	5131 (33.2 %)	5767 (50.1 %)

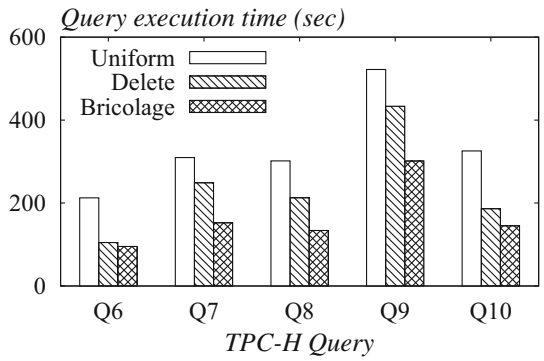

Fig. 11 Query execution time comparison

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 2b, we list the numbers we observe after running the workload. As we can see from the table, Bricolage is the best among the three strategies, and Delete outperforms Uniform with an exception in the I/O-intensive (2) configuration, where Delete wrongly removes the 4 slow machines due to the inaccuracy in query execution time estimation. 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 Fig. 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 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 Fig. 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 are 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 Fig. 12a, we plot the results for the CPU-intensive configuration. In this case, the estimated workload execution time is 5346 s, which is shorter than the actual execution time of 7371 s. 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 Fig. 12b. 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 Fig. 12b, we can also see that we underestimate performance for some machines (e.g., M_2) while overestimate 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

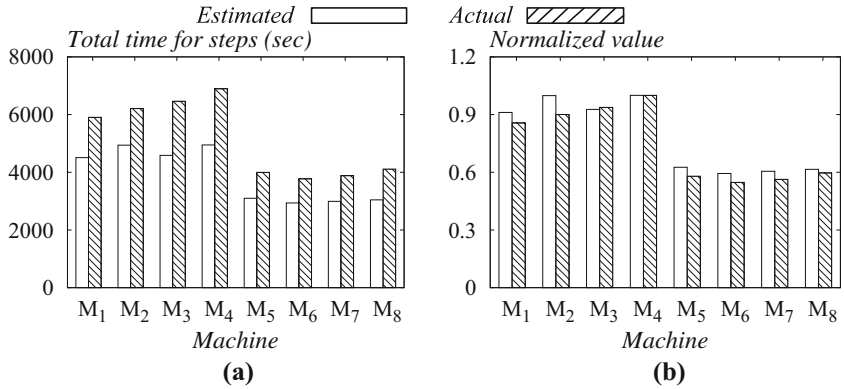

Fig. 12 Performance predictions for machines

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 further 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.

Table 3 Estimated time reductions using actual runs

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

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.

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 0.25 or 0.5 GB, and we vary the amount of data assigned to it from 10 to 50 %. The memory sizes of the other machines are set to 8 GB, 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 Fig. 13a. In both cases, the total time increases linearly with data size.

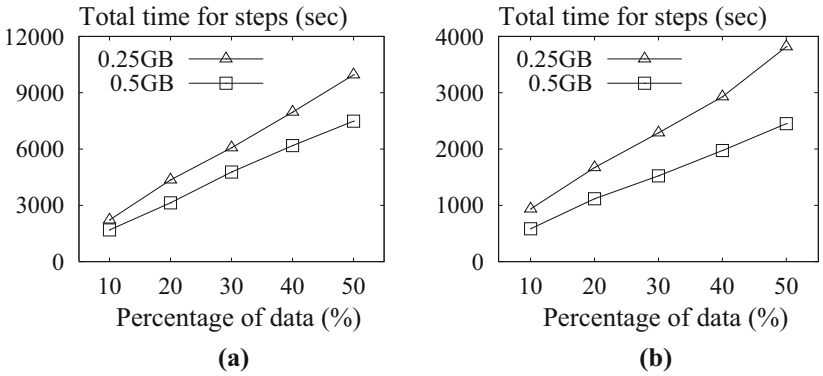

Fig. 13 Execution time versus data size

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

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

When memory size is 0.5 GB, all memory-consuming operators need no more than one pass, and when the memory size is 0.25 GB, 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 0.25 GB (2 passes are needed), the line should also have a steeper slope. To see this more clearly, we plot the results in Fig. 13b for a only subset of the most memory-consuming queries.

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 0.5 GB (where no operator needs more than one pass) for the last machine and 8 GB 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 are assigned to it compared to its original configuration with 8 GB memory.

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 0.25 GB 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 Sect. 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 min 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 min 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 s. 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 s.

6 The resource selection problems

In this section, we will discuss some natural generalizations of resource bricolage. We cover these generalizations with the broad term *resource selection*.

6.1 Problem definitions

The problem setting for resource selection is similar to that of resource bricolage in Sect. 2 (see Fig. 4 for more details), and it has some additional parameters and constraints, which are summarized as below.

Machines We have a set of n heterogeneous machines denoted as $M_1, M_2, \ldots, M_n$. A machine M_i $(1 \leq i \leq n)$ has a storage limit l_i, which represents the maximum percentage of the entire data set that it can hold. In addition, each machine M_i also has a price, which we refer to as $price_i$.

Workload We have a workload with a union of h steps: $S_1, S_2, \ldots, S_h$, and we use t_{ij} to represent the execution time for step S_j running on machine M_i if all the data were assigned to M_i.

Constraints The resource selection problems we consider here have either one of the following constraints: (i) The total price of the machines that we select must be no more than a budget $\mathcal{B}$, or (ii) we must finish the workload within time T.

Differences For the resource bricolage problem in Sect. 2, the n machines are the machines in a cluster, and we can use all to execute the workload. The n machines in the resource selection problems are considered as candidates. Due to the additional constraints, we only select a subset from them to execute the workload.

These n machines might belong to different classes with different prices. On the other hand, it is also possible that they have identical prices but varying performance. For instance, virtual machines of the same type provided by cloud service companies cost the same amount of money, but they may exhibit measurable different performance [13,22,33,35]. We will use the same approach to solve both cases (when they have identical prices or different prices). To simplify our discussion, we use the case where machines have the same prices as the primary case throughout the section. In the discussion, we will emphasize the differences between these two cases, if any.

When machines have the same price, a fixed budget $\mathcal{B}$ can buy a fixed number of machines from the candidate pool. We use b to denote the number of machines we can afford with a budget $\mathcal{B}$. In the first problem that we consider, our goal is to select b out of n $(b \leq n)$ machines to achieve the best performance. We call it a *minimum time resource selection* problem, and it is defined as follows.

Problem 1 Given a positive integer b, the **minimum time resource selection** problem is to select a subset of at most b machines that minimize the total execution time of a workload.

In the second problem that we consider, the goal is to process the workload within time T. As long as we can meet the performance requirement, it is desirable for us to spend less money to achieve this goal. In other words, we want to achieve the same performance with the minimum number of machines. We call this a *minimum cost resource selection* problem, which is defined as below.

Problem 2 Given a positive real number T, the **minimum cost resource selection** problem is to select the smallest number of machines that can finish the workload within time T.

6.2 NP-hardness of the problems

A straightforward solution for these two problems is to use a greedy algorithm, which works in the following way. For each machine, we first obtain its total execution time to process all the steps of the workload. A machine with less processing time usually indicates that it is more suitable for processing the workload. Thus, we can pick those machines with the least execution time first.

Unfortunately, using the same example we presented in Fig. 6, we can show that the greedy algorithm may produce a solution that is much worse than the optimal solution. Here, we illustrate how this is possible. For the minimum time resource selection problem, suppose that for the n machines, we want to select half of them to process the workload. Since machines in the second half $\left(M_{\frac{n}{2}+1}, M_{\frac{n}{2}+2}, \ldots, M_n\right)$ are considered "fast" for processing the workload, the greedy algorithm will choose them as the solution. The workload execution time for a cluster with these machines is $(a - \varepsilon) + \frac{2}{n}\varepsilon$. The optimal solution is to choose the set of "slow" machines, and the workload execution time is only

$\frac{2}{n}(a+\varepsilon)+\varepsilon$, which is roughly $\frac{2}{n}$ of the execution time of the greedy solution.

For the minimum cost resource selection problem, the greedy algorithm also prefers relatively fast machines, hoping that they can finish early to meet a given performance goal T. In the example presented in Fig. 6, the greedy algorithm will prefer machines from the "fast" half. For example, if we set $T \approx \frac{a+\varepsilon}{2}$, it will choose all the $\frac{n}{2}$ "fast" machines first. At this point, the workload execution time for the $\frac{n}{2}$ "fast" machines is $(a-\varepsilon)+\frac{2}{n}\varepsilon$, which is still worse than the performance goal. Thus, it needs to choose two more "slow" machines, resulting in a total of $\frac{n}{2}+2$ machines. However, for the same example, the optimal solution is to use roughly two "slow" machines to achieve the performance goal. Since these "fast" machines do not collaborate well in the same cluster and the "slow" machines work better together as a set, the greedy algorithm ends up using a lot more machines to meet the execution time requirement.

For the cases where machines have different prices, a greedy algorithm may select machines with smaller time-to-cost ratios first. However, this subtle difference does not change the fact that a greedy algorithm does not take into account collaboration between machines and thus may select machines with poor performance when working as a set. In the following, we prove the NP-hardness of both Problem 1 and Problem 2.

Theorem 1 *Given a workload of h steps, both the minimum time resource selection and the minimum cost resource selection problems are NP-hard when* $h > 1$.

Proof They are optimization problems, and their decision versions are the following: Is there a set of b instances so that the execution time of the workload is within T? To prove this theorem, it suffices to show that the decision versions of our problems is NP-complete, since an optimization problem is NP-hard if it has an NP-complete decision version. We prove that the decision problem is NP-complete in three steps. We first show that a Max-Intersection problem is NP-complete, and then, we prove that it can be reduced to a Min-Union problem (details will follow shortly). Finally, we reduce the Min-Union problem to the decision versions of our problems.

Given a finite universe $U = \{e_1, e_2, \ldots, e_h\}$, a set S of n sets $u_1, u_2, \ldots, u_n$ whose union is equals to U, and two integers k and s. The Max-Intersection problem is to determine whether there exists $u_{i_1}, u_{i_2}, \ldots, u_{i_k}$ such that $|\bigcap_{j=1}^{k} u_{i_j}| \geq s$. This problem is clearly in NP, since given a k-subset of S, we can easily verify whether the cardinality of their intersection is at least s. The remaining question is to prove the hardness. Consider the following known NP-hard problem: Given a bipartite graph, does there exist a complete bipartite subgraph, with each partition of size k (which is called a k-balanced biclique) [16]? We can reduce this known NP-hard problem to the Max-Intersection problem in the following way. For each vertex in the left partition, we create a set u_i, and the elements in u_i are the neighbors of this vertex in the bipartite graph. Let $s = k$. We claim that there is a k-balanced biclique if and only if there exists k subsets with an intersection of size at least k. As a result, the Max-Intersection problem is NP-complete.

The Min-Union problem is to determine whether there exists $u_{i_1}, u_{i_2}, \ldots, u_{i_k}$ such that $|\bigcup_{j=1}^{k} u_{i_j}| \leq s$. Let u^c denotes the complement of u in U. We have $\bigcap_{j=1}^{k} u_{i_j} = \left(\bigcup_{j=1}^{k} u_{i_j}^c\right)^c$. Therefore, if we know that $|\bigcup_{j=1}^{k} u_{i_j}^c| \leq |U| - s$, we also know that $|\bigcap_{j=1}^{k} u_{i_j}| \geq s$. Thus, the Min-Union problem is also NP-complete.

We can reduce the Min-Union problem to the decision versions of our problems. Consider a set $u_i \in S$ as a machine M_i and an element $e_j \in U$ as a step in our problems. We set t_{ij} to k if u_i contains e_j, and 0 otherwise. For each machine I_i, we set the maximum percentage of data it can hold to $\frac{1}{k}$. Let $b = k$, and $T = s$. We claim that there exists a k-subset of S whose union cardinality is less than or equals to s if and only if there is a set of k machines such that the execution time of the workload is within s. This completes the reduction.

The case where machines may have different prices is more general than the case where they have identical price. If we can find an efficient algorithm to solve the general case, we can use the same algorithm to solve the special case where machine prices are the same. Since we have proved that the special case is NP-hard, the more general case must be NP-hard as well. In the next section, we present our solution for solving these two problems.

7 A resource selection technique

Like the problem in Sect. 4, we first need to estimate all t_{ij} values. We use the approach presented in Sect. 3 to do the estimation. Once all the t_{ij} values are computed, we have enough information to solve the problems. Next, we present our approaches, which employ different solvers based on mixed-integer and constraint programming to search for the optimal solutions.

7.1 Minimum time resource selection

In this problem, we know that we can only afford to use b machines due to a budget constraint, and the goal is to select a set of b machines to minimize workload execution time. To solve this problem, in addition to the variables that we introduced in Sect. 4.2, we use one more binary variable m_i to indicate whether machine M_i will be selected or not. We model the problem using mixed-integer programming as follows.

$$\text{minimize} \sum_{j=1}^{h} x_j$$

$$\text{subject to } p_i t_{ij} \le x_j \qquad 1 \le i \le n, 1 \le j \le h$$

$$\sum_{i=1}^{n} p_i = 1$$

$$0 \le p_i \le l_i \qquad 1 \le i \le n$$

$$m_i \in \{0, 1\} \qquad 1 \le i \le n$$

$$m_i \ge p_i \qquad 1 \le i \le n$$

$$\sum_{i=1}^{n} m_i \le b$$

The objective function we want to minimize is the total execution time of all steps. The first three constraint functions are the same as those in Sect. 4.2. The variable m_i can take a value of either 0 or 1, where 1 indicates that machine M_i is selected, and 0 indicates that it is not. Since the total number of machines we want to select is at most b, we have $\sum_{i=1}^{n} m_i \le b$. We also want to enforce that we will not allocate any data to a machine that is not selected. In other words, when $m_i = 0$ $(1 \le i \le n)$, we must also have $p_i = 0$; and when $m_i = 1$, p_i can be greater than 0. We use the constraint function $m_i \ge p_i$ to enforce this requirement. Since m_i can only take two values, when $m_i = 0$, $m_i \ge p_i$ implies that p_i must be 0 as well. When $m_i = 1$, p_i can be any nonnegative value less than or equal to 1. A solution that satisfies all the constraint functions gives us a set of b machines that minimize the execution time of our workload. The constraints can be revised to deal with the case where machines have different prices (e.g., some are low-end machines which cost less money). Assume that the budget we have is $\mathcal{B}$, we can replace $\sum_{i=1}^{n} m_i = b$ with a new function $\sum_{i=1}^{n} m_i * price_i = \mathcal{B}$ to deal with this case, where $price_i$ is the price for choosing machine M_i.

Note that the resource bricolage problem is actually a special case of the minimum time resource selection problem, where we have unlimited (or enough) budget to use all n machines. Thus, we can replace b with n (the total number of machines) in the MIP to solve the resource bricolage problem. However, due to the integer variables, solving the MIP is an NP-hard problem. Thus, we use the simpler program in Sect. 4.2 to solve the resource bricolage problem.

7.2 Minimum cost resource selection

In this problem, we do not have a fixed budget that limits the number of machines that we can use. The constraint we have is that we need to finish the workload within a desired amount of time T with least number of machines. We formulate this problem as below.

$$\text{minimize} \sum_{i=1}^{n} m_i$$

$$\text{subject to } p_i t_{ij} \le x_j \qquad 1 \le i \le n, 1 \le j \le h$$

$$\sum_{i=1}^{n} p_i = 1$$

$$0 \le p_i \le l_i \qquad 1 \le i \le n$$

$$m_i \in \{0, 1\} \qquad 1 \le i \le n$$

$$m_i \ge p_i \qquad 1 \le i \le n$$

$$\sum_{j=1}^{h} x_j \le T$$

In the objective function, we minimize the total number of selected machines. The first five constraint functions are the same as those in Sect. 7.1. Since we have a performance target for the workload, the total execution time of all x_js should not exceed the desired execution time. Thus, we have the additional constraint function $\sum_{j=1}^{h} x_j \le T$. In the case where the machines have different prices, we can modify the objective function to $\sum_{i=1}^{n} m_i * price_i$ to minimize the total cost of the selected machines.

Query execution time estimation is known to be a challenging problem, and previous work has proved that it is impossible to provide execution time estimates with worst-case guarantees due to cardinality estimation errors [5]. As a result, when meeting a performance goal is critical, we use $\sum_{j=1}^{h} x_j \le T * \alpha$ instead of $\sum_{j=1}^{h} x_j \le T$ as a constraint function, where $0 < \alpha \le 1$. When we are confident in the query execution time estimation, we can choose an α that is close to 1, and we can use a smaller α when we are not.

On the other side, clouds are typically elastic. They allow users to request resources dynamically. In the future, we plan to study the problem of how to automatically expand and shrink a cluster when our initial estimates are off or the server performance changes. In any case, our technique proposed here can provide us with a reasonable set of resources to start with.

8 Simulation experiments for resource selection

In this section, we construct a number of cases that simulate different real-world scenarios to evaluate the performance of various alternatives for solving the resource selection problems. We first give a summary of the experimental settings and the techniques that we evaluate, and then, we compare the performance of these techniques. Note that our simulation is by no means a complete coverage of all possible scenarios in practice. The main purpose of the simulation experiments

is to gain some insights into the performance differences of alternative approaches.

8.1 Experimental setup

We consider the following cases.
(1) *With identical machines* This is the simplest and ideal setting where all machines are identical.
(2) *With exceptional machines* In this case, most of the machines are identical. But there are outliers that are much slower than the majority of machines. This case corresponds to the real world scenario where all the machines are supposed to be the same (i.e., with the same hardware and configurations); however, a small number of them may exhibit poor performance due to a defect, such as a bad disk or a corrupted memory card.
(3) *With proportional machines* There are multiple types of machines in this case, and for any two types of machines, they have the following property. Assume that for one type of machine, its resources can be quantified as $res_1, res_2, \ldots, res_x$, where x is the number of different resource types. For the other type of machine, its amounts of resources must be $res_1 * \gamma, res_2 * \gamma, \ldots, res_x * \gamma$, where γ is the ratio between the capabilities of these two types of machines. This case may correspond to the scenario where we have machines of multiple generations, and the newer generation machine is more powerful than the order generation machine in every aspect with the same ratio.
(4) *With arbitrary machines* Each machine can have arbitrary amounts of computing resources. It represents the most generalized case in practice.

In Sect. 1.1, we pointed out that we must consider three aspects simultaneously, including capabilities of machines, collaboration between machines, and allocation of data, in order to provide a satisfactory solution for our resource selection problems. We compare the performance of the following approaches, which take into account none, some, or all of the three aspects, respectively.

Blind Blind completely ignores differences among machines and randomly selects a set of machines to process the workload. Data are partitioned uniformly across all the selected machines. This approach disregards all three principles that we believe should be considered.

Greedy with uniform data allocation As we have discussed in Sect. 6.2, this approach prefers powerful machines. Data are allocated to machines in a uniform fashion. This approach respects the criteria that we should choose machines that can process the workload fast. We refer to this approach as Greedy_U.

Greedy with best data allocation This approach greedily selects fast machines and then uses the technique we proposed in Sect. 4 to find the optimal data partitioning scheme to allocate data. This approach selects individual powerful machines and allocates the proper amount of data to them to minimize the workload execution time, but these machines may not collaborate well in the cluster. We refer to this approach as Greedy_B.

Optimal This is the technique we presented in Sect. 7, and it provides the optimal solution by taking into account all the three aspects.

Table 5 Overall performance of different approaches

Approaches	Case (1)	Case (2)	Case (3)	Case (4)
Blind	√	×	×	×
Greedy_U	√	√	×	×
Greedy_B	√	√	√	×
Optimal	√	√	√	√

For each case listed above, we compare the performance of these four approaches. In Table 5, we summarize the overall performance of these approaches. We use √ to indicate that the corresponding approach can find the optimal solution for a given case, and we use × to show that it may not. In Case (1) where machines have no difference, this is the simplest case, and even the simplest approach Blind is sufficient. In Case (2) where there are a few slow machines (outliers), Blind has an equal probability of selecting those outliers, which may result in system performance degradation. The greedy algorithms, with or without a best data allocation scheme, can successfully exclude the outliers, and thus, they can choose the right set of machines. In Case (3) where we have a group of proportional machines, it is obvious that Blind may also select those slow machines. Although the greedy algorithms can select the fastest machines, Greedy_U may overload the slow machines and at the same time underutilize the fast ones, and thus, it cannot always provide the optimal solution. In Case (4) where machines have arbitrary computing capabilities, Blind and Greedy_U do not work well, and the reasons are similar to those for Case (3). Greedy_B could not provide the optimal solution either. The reasons are presented in detail in Sect. 6.2.

Next, we conduct a set of experiments to validate our theory and to study of the performance of different approaches in Cases (2), (3), and (4).

8.2 Experiments for minimum time resource selection

Results for Case (2) We simulate 100 machines with a small number of outliers in our simulation experiment. Our workload consists of just one step, and the results are similar for multi-step workloads. For simplicity, we assume that the normal machines can process all data in 1 unit of time (e.g., 1 h, 1 day), while the slow machines need r units of time ($r \geq 1$) to process the same data. Suppose that our bud-

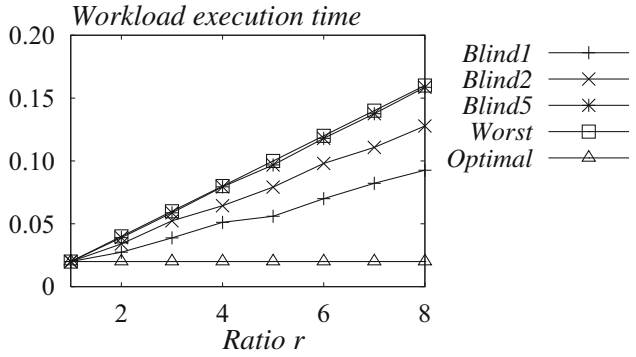

Fig. 14 Workload execution time comparison for Case (2)

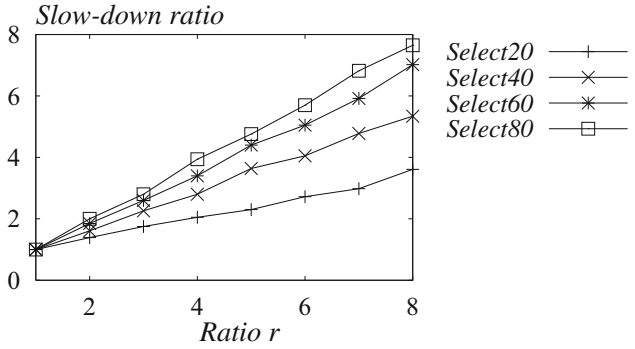

Fig. 15 Slow-down ratio for Case (2)

get can only afford 50 machines, and the goal is to select 50 out of 100 machines to minimize workload execution time. We employ the four approaches to select machines. In Fig. 14, we compare the workload execution times of Blind and Optimal. Note that the workload processing times for the two greedy approaches are the same as Optimal, and thus, we omit the results from the figure. We vary r from 1 to 8 to cover the most common range of performance differences among machines, and we could have 1, 2, or 5 slow machines out of 100. We use Blind1, Blind2, and Blind5 to represent the experiment results for these cases, respectively. Since Blind randomly selects machines from the candidate set, its performance varies with the selected machines. Therefore, we repeat the same experiment for 100 times, and we take the average values as the results. We also include the workload execution times for the worst-case scenario where the slow machines are always selected.

Since we select 50 machines for the workload, each machine gets 2 % of the data. A regular machine takes 0.02 unit of time to process the data, while a slow machine takes $0.02r$ unit of time. When the machines are identical ($r = 1$), the workload execution times of all approaches are 0.02. When there are some slow machines in the candidate set, Blind may end up selecting some of them. Even when there is only one such machine, the performance of Blind1 is much worse than Optimal. The performance of Blind gets worse when the number of slow machines increases. When there are 5 slow machines, the performance of Blind5 is almost as bad as that of the worst-case scenario.

We then fix the number of slow machines to 2 to measure the performance of Blind and Optimal. The number of machines to be selected are 20, 40, 60, or 80, per our budget. Suppose that for an approach A, by using b machines, it can process the workload in time $t_A(b)$. We define the slow-down ratio of an approach A with respect to Optimal as $t_A(b)/t_{Opt}(b)$. We measure the slow-down ratios of Blind for different cases and show the results in Fig. 15. When more machines need to be selected, Blind has a higher chance of getting a slow machine, and as a result, it performs worse. For most of the cases that we tested, the slow-down ratios of Blind are greater than 2. In other words, the workload execution times of Blind are more than twice as long as that of Optimal.

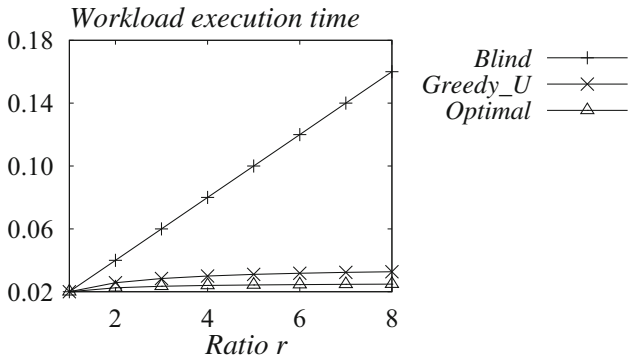

Fig. 16 Workload execution time comparison for Case (3)

For Case (2), a "blind" approach may produce very bad solutions compared to Optimal, and a simple greedy heuristic is necessary, since it can eliminate bad choices to provide a better solution.

Results for Case (3) We simulate 100 machines of 10 different types. Like Case (2), the workload consists of just one step, and the results for multi-step workloads are similar. We assume that the most powerful machine can process all data in 1 unit of time, and the ith best machine can process the same data in $1 + \frac{(i-1)(r-1)}{9}$ unit of time, where r ($r \geq 1$) is the ratio between execution times of the slowest and the fastest machines for processing the same amount of data. We want to select 50 out of the 100 machines to process the workload. Figure 16 compares the workload processing times of Blind, Greedy_U, and Optimal. As mentioned earlier, we measure the average workload processing time for Blind. Note that the performance of Greedy_B is the same as Optimal; therefore, it is not presented here.

Recall that a slow machine takes $0.02r$ unit of time to process the data, and the processing time of a step is

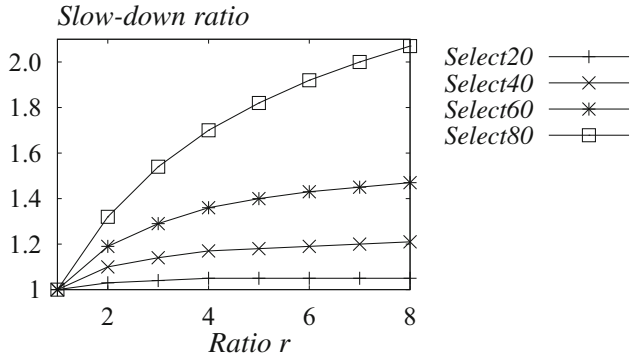

Fig. 17 Slow-down ratio for Case (3)

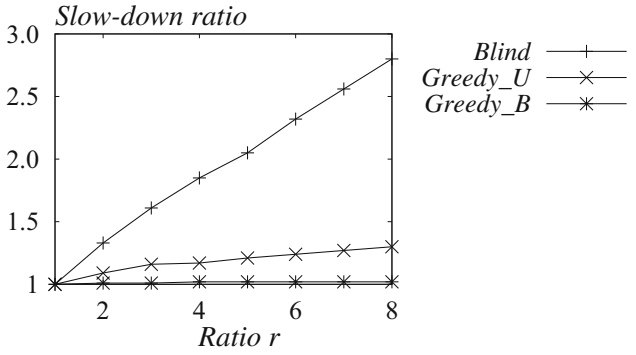

Fig. 18 Slow-down ratio for Case (4)

determined by a slowest machine. Since Blind has a high probability of getting at least one slow machine, its workload processing time increases linearly with the value of r. Greedy_U can dramatically speed up the processing by excluding slow machines. The optimal approach chooses the same set of machines, but it uses the technique in Sect. 4 to allocate data. Thus, it can further reduce workload execution time by another 30 % in our simulation.

Next, we further investigate the performance of Greedy_U by varying the number of selected machines. In Fig. 17, we show that the slow-down ratios of Greedy_U when 20, 40, 60, or 80 machines are selected. As we can see, when a higher percentage of the computing resources are chosen, the performance of Greedy_U gets worse. The reason is that when more machines need to be included, machines with worse performance will be added. Since Greedy_U allocates data in a uniform fashion, slow machines will severely degrade performance.

For Case (3), a greedy heuristic can pick the same set of machines as the Optimal approach and thus greatly decrease data processing time. However, since it may overload slow machines, its performance may still be worse than Optimal. We also need to employ a good data allocation scheme for better performance.

Results for Case (4) We simulate 100 machines, and the workload consists of 100 steps. We noticed that when there is more than one step in the workload, the experiment results look quite similar. For Case (4), we do not make any assumption about how long it takes for a machine to process the data. For each machine, we randomly pick a processing speed $Speed_{Res}$ from $[Speed_{low}, Speed_{high}]$ for each type of resource Res in {CPU, I/O, Network}. In our simulation experiment, $Speed_{high}$ is set to be 8 times as fast as $Speed_{low}$. For each step, we randomly pick a cost $Cost_{Res}$ from $[Cost_{low}, Cost_{high}]$ for each type of resource Res in {CPU, I/O, Network}. $Cost_{high}$ is set to be 8 times as large as $Cost_{low}$ as well. Then, the processing time of a step running on a machine is the maximum of $C_{Res}/Speed_{Res}$, for any Res in {CPU, I/O, Network}. The absolute values of $Speed_{low}$ and $Cost_{low}$ do not make much difference to the results.

In the simulation experiment, we generate 100 machines and 100 steps using the approach described above. We then pick 50 out of 100 machines using the four different approaches, and we calculate the slow-down ratios of Blind, Greedy_U, and Greedy_B. The same procedure is repeated 100 times, and we measure the average slow-down ratios for each approach. The results are shown in Fig. 18.

The performance of Blind and Greedy_U is worse than Optimal as expected. Somewhat surprisingly, Greedy_B is almost as good as Optimal, with an average slow-down ratio less than 1.02. After further investigation, we found that for Greedy_B to have bad performance, the set of n machines must have the following three properties: (i) It has a subset of "incompatible" machines (e.g., a set that contains machines with very powerful CPUs but very limited other resources and machines with very fast disks but very limited other resources), (ii) it has a subset of "compatible" machines, and (iii) a machine in the "incompatible" subset is faster than a machine in the "compatible" subset when they are used individually. Property (iii) is a trick to deceive Greedy_B into picking the subset of "incompatible" machines. Thus, for steps with different resource requirements, we will always have some of the machines act as a bottleneck due to resource scarcity, resulting in longer execution time. When all the three conditions are satisfied, Greedy_B will perform much worse than Optimal. However, the 100 machines that we generate randomly seldom simultaneously fulfill all three conditions. As a result, the performance of Greedy_B is usually very good.

In Fig. 19, we demonstrate cases where Greedy_B has a slow-down ratio greater than 1.02 (note that for each value of r, we repeat the experiment 100 times). As we can see, although Greedy_B usually perform well in our simulation experiment, its slow-down ratios can get very high for some cases (when all three conditions stated earlier are satisfied). There is no guarantee of its performance.

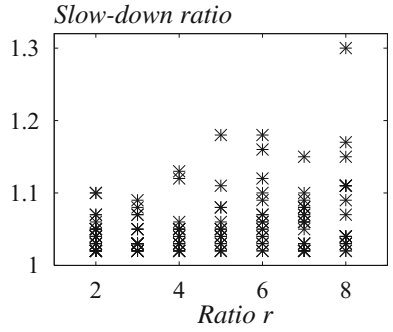

Fig. 19 Slow-down ratio of Greedy_B for Case (4)

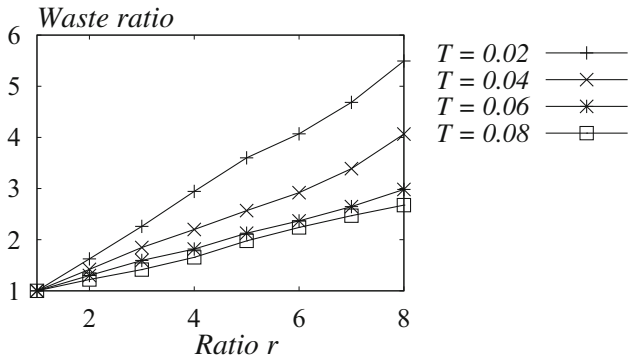

Fig. 21 Waste ratio for Case (2)

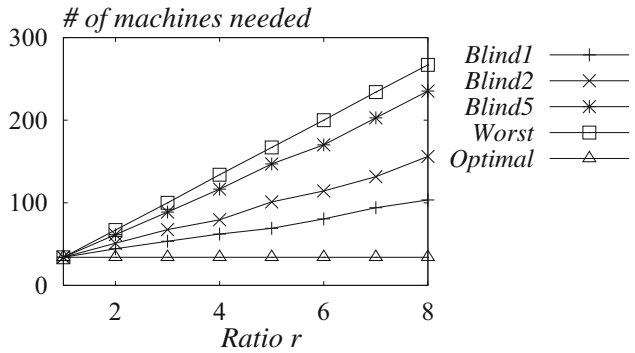

Fig. 20 Number of machines needed for Case (2)

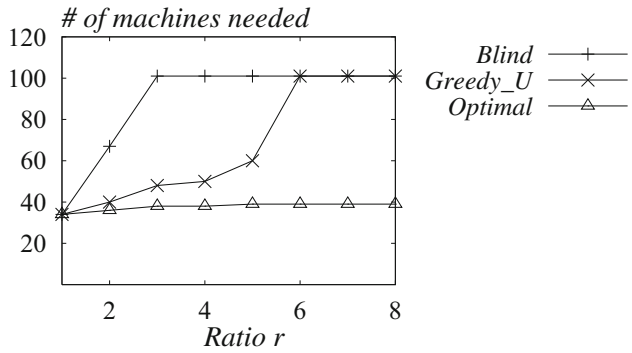

Fig. 22 Number of machines needed for Case (3)

8.3 Experiments for minimum cost resource selection

The machines and workloads we employ here for the experiments are the same as those we use in Sect. 8.2 for corresponding cases. The goal is to select a subset of machines with minimum cost to process the workload within a given time T. When setting up the experiments, we want to choose a reasonable value for T to make sure that the targeted time is achievable.

Results for Case (2) The performance goal T is set to 0.03, so roughly 30 % of the machines are needed for Optimal to achieve this goal. Blind works in the following way when selecting the computing resources. It starts with an empty set and repeatedly chooses a random new machine to add to the set. This process stops when it runs out of machines or when it achieves the performance goal T. However, in this case, we assume that when Blind uses up all the original 100 machines, it can continue to add an unlimited amount of fast machines to achieve the goal. We repeat the experiment 100 times to measure the average number of machines it needs to achieve the goal. In Fig. 20, we compare the (average) number of machines it needs to achieve the same goal $T = 0.03$ for Blind and Optimal. We can see that Optimal always uses 34 machines. As the number of slow machines increases, on average, Blind needs more machines to achieve the goal. Thus, the performance gap between Blind and Optimal increases.

We then fix the number of slow machines to two while varying T from 0.02 to 0.08 to evaluate the performance of Blind and Optimal. An optimal approach produces a solution with minimum number of machines (minimum cost), and any additional machines used by a non-optimal approach can be considered as a waste. Suppose that the number of machines needed by an approach A to achieve a performance goal T is $Num_A(T)$. We define the waste ratio of an approach A with respect to Optimal as $Num_A(T)/Num_{Opt}(T)$. We calculate the waste ratio of Blind and present the results in Fig. 21. Note that the numbers are averaged from 100 repeated experiments. From the graph, we can see that when we have a higher targeted performance (T is smaller), Blind will waste more machines and have a higher waste ratio.

Results for Case (3) We set the performance goal T to 0.03, and we compared the number of machines needed by different approaches to achieve this goal. The results are illustrated in Fig. 22. The results for Greedy_B are not included, since they are the same as that of Optimal. For all r that we considered, Optimal needs less than 40 machines to meet the performance goal. Blind and Greedy_U usually need much more machines for the same goal. When r equals 6, 7, or 8, Greedy_U cannot achieve the performance goal even if it uses all the 100 machines. When r is greater than two, it

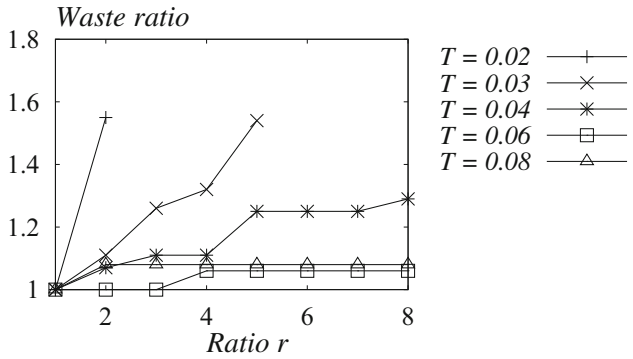

Fig. 23 Waste ratio for Case (3)

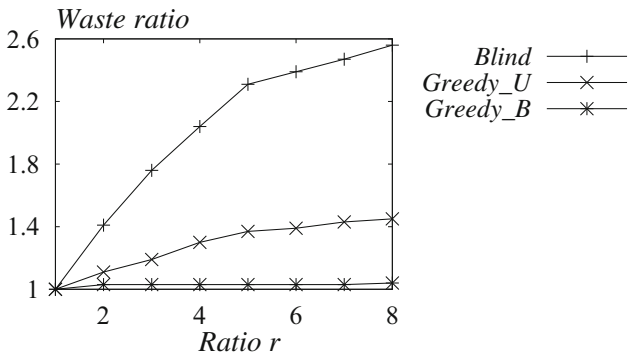

Fig. 24 Waste ratio for Case (4)

is impossible for Blind to achieve this goal either. For these cases, we plot their data points with the value of 101 in Fig. 22 to indicate that the target performance is unachievable for the corresponding approaches.

More results for Greedy_U are shown in Fig. 23, where we vary T from 0.02 to 0.08. We omit the results when the performance goal cannot be achieved from the graph. As we mentioned before, Greedy_U cannot meet the goal when T equals 0.03 and r equals 6, 7, or 8. When T gets smaller (e.g., $T = 0.02$), Greedy_U requires more machines, and it is harder to meet the goal.

Results for Case (4) Since the machines and workloads are randomly generated using the strategy described in Sect. 8.2, the achievable performance goal varies with the machines and workloads generated. In light of this, when a set of machines and a workload are generated, we run our program for the minimum time resource selection problem with the value of b set to 33. Based on the outputs of the program, we obtain the minimum workload processing time when no more than 33 machines are used. This minimum time achievable with 33 machines is used as the targeted time, and we evaluate the performance of Blind and the greedy algorithms. In this case, Optimal will always use 33 machines, and we calculate the waste ratios for Blind and the greedy algorithms. The results are shown in Fig. 24. Note that the numbers we show here are averaged from 100 times of repeated experiments.

As we can see, the performance of Blind is the worst. The greedy approach with uniform data allocation is much better than Blind, but is worse than Optimal. The greedy approach with best data allocation is usually as good as Optimal, and this matches our analysis in Fig. 19.

9 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 [5,17,18,20,21,24]. 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 [26] is an exception. However, it cannot provide accurate estimates for queries running on database systems [19]. On the other hand, the second category of work focuses on query running time prediction before a query starts [4,12,14,36,37]. In [37], 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 [4,12,14]. 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 [15] investigates different multi-attribute partitioning strategies, and it tries to place tuples that satisfy the same selection predicates on fewer machines. The work in [7,23] 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 [27,31], 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 [8,30] 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.

Our work is also related to skew handling in parallel database systems [11,38,39]. 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 [2,10,40]. A suite of optimizations are proposed in [2] to improve MapReduce performance on heterogeneous clusters. Zaharia et al. [40] 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. The work in [10] proposes a scalable data center scheduler to assign workloads to servers. It classifies incoming applications with respect to platform heterogeneity and workload interference. This is achieved by utilizing collaborative filtering techniques that combine a minimal profiling signal about the new application with the large amount of data available from previously scheduled applications. It relies mostly on information from previously scheduled workloads. Since we do not have any information about previously scheduled workloads, this technique cannot be directly applied here for selecting machines. Besides, [10] does not consider the data partitioning problem that needs to be addressed in our work.

10 Conclusion and future work

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. For resource bricolage, extensive experiments confirm that our technique can provide good and reliable partition recommendations for given workloads with minimal overhead. We also discussed two resource selection problems: one with budget constraints and the other with time constraints. We deployed mixed-integer programming techniques to solve these two problems. In our experiments, we showed that completely ignoring performance differences of the candidate machines can result in poor performance. The combination of greedy resource selection and heterogeneity-aware data allocation techniques can generally provide satisfactory performance; however, there is no guarantee of its performance in some cases. Our proposed models provide the best performance among these alternatives, at a price of complexity.

This paper also lays down a foundation for several directions toward future studies to improve database performance running in the cloud. In our paper, we have generalized resource bricolage into two resource selection problems. It would be interesting to explore other generalizations. One such generalization is to consider dynamic characterization of resources to accommodate sharing in the cloud environment, where sometimes a machine is fast when it is dedicated, and sometimes it is slow when it is shared. The programming 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. Furthermore, a query that runs concurrently with another query could have negative or positive impact on the other query's performance, and previous work has shown that an interaction-aware query scheduler can provide significant performance improvements [3]. Thus, 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. Previous research has revealed that the supposedly identical instances provided by a public cloud often exhibit measurable performance differences. 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.

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

References

1. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD (2004)
2. Ahmad, F., Chakradhar, S.T., Raghunathan, A., Vijaykumar. T.N.: Tarazu: Optimizing mapreduce on heterogeneous clusters. In: ASPLOS (2012)
3. Ahmad, M., Aboulnaga, A., Babu, S., Munagala, K.: Modeling and exploiting query interactions in database systems. In: CIKM (2008)
4. Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., Zdonik, S.B.: Learning-based query performance modeling and prediction. In: ICDE (2012)
5. Chaudhuri, S., Kaushik, R., Ramamurthy, R.: When can we trust progress estimators for SQL queries? In: SIGMOD (2005)

6. Chaudhuri, S., Narasayya, V., Ramamurthy, R.: Estimating progress of execution for SQL queries. In: SIGMOD (2004)
7. Copeland, G., Alexander, W., Boughter, E., Keller, T.: Data placement in Bubba. In: SIGMOD Record (1988)
8. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. In: PVLDB (2010)
9. Dantzig, G.B., Thapa, M.N.: Linear Programming 1: Introduction. Springer, Berlin (1997)
10. Delimitrou, C., and Kozyrakis, C.: Paragon: Qos-aware scheduling for heterogeneous datacenters. In: ASPLOS (2013)
11. DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical skew handling in parallel joins. In: VLDB (1992)
12. Duggan, J., Cetintemel, U., Papaemmanouil, O., and Upfal, E.: Performance prediction for concurrent database workloads. In: SIGMOD (2011)
13. Farley, B., Juels, A., Varadarajan, V., Ristenpart, T., Bowers, K.D., Swift, M.M.: More for your money: exploiting performance heterogeneity in public clouds. In: SoCC (2012)
14. Ganapathi, A., Kuno, H., Dayal, U., Wiener, J.L., Fox, A., Jordan, M., Patterson, D.: Predicting multiple metrics for queries: better decisions enabled by machine learning. In: ICDE (2009)
15. Ghandeharizadeh, S., DeWitt, D.J., Qureshi, W.: A performance analysis of alternative multi-attribute declustering strategies. In: SIGMOD (1992)
16. Johnson, D.S.: The NP-completeness column: An ongoing guide. J. Algorithms. **6**(3), 434–451 (1985)
17. König, A.C., Ding, B., Chaudhuri, S., Narasayya, V.: A statistical approach towards robust progress estimation. In: PVLDB (2012)
18. Li, J., Nehme, R.V., Naughton, J.F.: GSLPI: A cost-based query progress indicator. In ICDE (2012)
19. Li, J., Nehme, R.V., Naughton, J.F., Toward progress indicators on steroids for big data systems. In CIDR (2013)
20. Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M.W.: Toward a progress indicator for database queries. In: SIGMOD (2004)
21. Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M.W.: Increasing the accuracy and coverage of SQL progress indicators. In: ICDE (2005)
22. Mangot, D.: EC2 Variability: The Numbers Revealed. http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed (2009)
23. Mehta, M., DeWitt, D.J.: Data placement in shared-nothing parallel database systems. VLDB J. **6**(1), 53–72 (1997)
24. Mishra, C., Koudas, N.: A lightweight online framework for query progress indicators. In: ICDE (2007)
25. Morton, K., Balazinska, M., Grossman, D.: Paratimer: a progress indicator for MapReduce DAGs. In: SIGMOD (2010)
26. Morton, K., Friesen, A., Balazinska, M., Grossman, D.: Estimating the progress of MapReduce pipelines. In: ICDE (2010)
27. Nehme, R., Bruno, N.: Automated partitioning design in parallel database systems. In: SIGMOD (2011)
28. Ou, Z., Zhuang, H., Lukyanenko, A., Nurminen, J.K., Hui, P., Mazalov, V., Yla-Jaaski, A.: Is the same instance type created equal? Exploiting heterogeneity of public clouds. IEEE Trans. Cloud Comput. **1**(2), 201–214 (2013)
29. Ou, Z., Zhuang, H., Nurminen, J.K., Ylä-Jääski, A., Hui, P.: Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In: HotCloud (2012)
30. Pavlo, A., Curino, C., Zdonik, S.: Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: SIGMOD (2012)
31. Rao, J., Zhang, C., Megiddo, N., Lohman, G.: Automating physical database design in a parallel database. In: SIGMOD (2002)
32. Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., Kozuch, M. A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: SoCC (2012)
33. Schad, J., Dittrich, J., Quiané-Ruiz, J.-A., Runtime measurements in the cloud: observing, analyzing, and reducing variance. In: PVLDB (2010)
34. SQL Server 2012 Parallel Data Warehouse. http://www.microsoft.com/en-ca/server-cloud/products/analytics-platform-system/
35. Wang, G., Ng, T.S.E.: The impact of virtualization on network performance of amazon EC2 data center. In: INFOCOM (2010)
36. Wu, W., Chi, Y., Hacígümüş, H., Naughton, J.F.: Towards predicting query execution time for concurrent and dynamic database workloads. In: PVLDB (2013)
37. Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., Naughton, J.F.: Predicting query execution time: are optimizer cost models really unusable? In: ICDE (2013)
38. Xu, Y., Kostamaa, P.: Efficient outer join data skew handling in parallel DBMS. In: PVLDB (2009)
39. Xu, Y., Kostamaa, P., Zhou, X., Chen, L.: Handling data skew in parallel joins in shared-nothing systems. In: SIGMOD (2008)
40. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I. Improving mapreduce performance in heterogeneous environments. In: OSDI (2008)
41. Zhuang, H., Liu, X., Ou, Z., Aberer, K.: Impact of instance seeking strategies on resource allocation in cloud data centers. In: CLOUD (2013)
42. Zou, T., Bras, R.L., Salles, M.V., Demers, A., Gehrke, J.: ClouDiA: a deployment advisor for public clouds. In: PVLDB (2013)
43. Zou, T., Wang, G., Salles, M.V., Bindel, D., Demers, A., Gehrke, J., White, W.: Making time-stepped applications tick in the cloud. In: SOCC (2011)