Performance Prediction and Resource Bricolage for Database Systems

by

Jiexing Li

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN-MADISON

2014

Date of final oral examination: 06/02/2014

Committee in charge:
- Jeffrey F. Naughton, Professor, Computer Sciences
- David J. DeWitt, Emeritus Professor, Computer Sciences
- Jignesh M. Patel, Professor, Computer Sciences
- AnHai Doan, Professor, Computer Sciences
- Rimma V. Nehme, Senior SDE, Microsoft Jim Gray Systems Lab
- Sung Kim, Associate Professor, Operations and Information Management

I dedicate this thesis to my parents and my husband, Xiang, for their constant support and unconditional love.

Acknowledgements

Foremost, I would like to take this opportunity to express my wholehearted gratitude to my supervisor, Jeffrey Naughton, for all he has done for me in the past five years. To this day, I can still remember the first time when I went to his office to tell him about my research interests. He then gave me this very unique opportunity that completely aligned with my goals to conduct my research and to pursue my dreams. Jeff's inspiring advice and insightful criticism were extremely essential and valuable for the completion of my research projects and thesis. Most importantly, as an advisor, he has always tried to give me the kind of advice and training that helped me learn to handle problems and acquire the ability to work independently in the future. This kind of training allowed me to grow both professionally and personally.

Jeff has been much more than just a thesis advisor. He has also been a warm senior and an astute career counselor for me. He supported me when I made mistakes, encouraged me during hard times, tolerated me with patience when I was less productive, and praised me generously for my accomplishments. I could not have imagined having a better advisor for my Ph.D. study.

I would like to express my deepest appreciation to both Rimma Nehme and David DeWitt. Rimma worked closely with me in the past five years, sometimes like a mentor, sometimes like a colleague, and sometimes like a friend. She offered me constructive

comments and suggestions for my work. Her support and encouragement helped me come out of some hard times in my life. David generously supported me with a Microsoft Research Assistantship. He also gave me insightful comments like an advisor and warm encouragement like a family member throughout my Ph.D. study. This dissertation could not be completed without their help.

I really appreciate the feedback offered by Jignesh Patel and Chris Rè during my preliminary exam. I also want to thank AnHai Doan and my external examiner, Sung Kim, for their insightful comments on my dissertation.

I would like to thank Alan Halverson, Hideaki Kimura, Willis Lang, Karthik Ramachandra, Eric Robinson, Srinath Shankar, Nikhil Teletia, Dimitris Tsirogiannis, Donghui Zhang, and Melody Bakken from the Microsoft Jim Gray Systems Lab for their valuable suggestions. Working with them has been a fun and rewarding experience. I would like to offer my special thanks to Ian Rae. Advice and comments given by him have been very helpful for my experiment implementations and for writing papers.

I am grateful to have had the opportunity to spend a summer at Microsoft Research-Redmond. I would like to thank my mentors there: Vivek Narasayya, Surajit Chaudhuri, and Arnd Christian König.

I also want to thank my colleagues in the Department of Computer Sciences at the University of Wisconsin-Madison: Akanksha Baid, Spyros Blanas, Xiaoyong Chai, Fei Chen, Jaeyoung Do, Thanh Do, Avrilia Floratou, Chaitanya Gokhale, Heng Guo, Yeye He, Guoliang Jin, Arun Kumar, Yinan Li, Ji Liu, Jie Liu, Feng Niu, Jiasi Song, Junming Sui, Chong Sun, Bruhathi Handanahal Sundarmurthy, Khai Tran, Ba-Quy Vyong, Wentao Wu, Xi Wu, Jia Xu, Ce Zhang, Yupu Zhang, and Chen Zeng. They offered an relaxed and friendly environment that made my every day life enjoyable. I also want to thank them all for the advice and help they provided in many different ways. I would like to thank my roommates Lingxiao Li and Ying Zheng as well. They have made my five years' research

time joyful and wonderful.

I would like to thank my husband, Xiang Peng, for the love and support that he has given me through this journey. Finally, I am deeply grateful to my parents for their understanding and tolerance in supporting my Ph.D. study. Although they had their own concerns and doubts at the beginning, they have always had faith in me. I would not be the person I am today if it were not for their unlimited love and support!

Abstract

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 very large and complex data management systems. Although these systems can significantly reduce data processing time, issues such as hardware/software skew, resource contention, and failures are more likely to arise. All large and complex systems have to face this unwanted but inevitable fact. Due to all these issues, it gets harder to anticipate the future state of a system, and a one-time decision model used by schedulers, optimizers or resource managers will be vulnerable to state changes.

Meanwhile, 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. Very large data processing is increasingly becoming a necessity for modern applications. For database systems running in a heterogeneous cluster, the default data partitioning strategy may overload some of the slow machine while at the same time it may under-utilize the more powerful machines. Since the processing time of a parallel query is determined by the slowest machine, such an allocation strategy may result in significant query performance degradation.

It is not uncommon today for us to decide which computing resources should be used

to build a cluster or to run an application from a diverse range of such resources. Very often, when a new cluster is built or an old cluster is upgraded, there are various machines, low-end or high-end, that we can choose from. Different choices may lead to different costs or performance. Thus, we will encounter a resource selection problem if we have a limited budget or a performance goal.

This dissertation makes three contributions by addressing these three problems: *query progress estimation*, *data allocation*, and *resource selection*. The first contribution is the design and implementation of a new cost-based query progress indicator, called *GSLPI*, to produce more accurate progress estimates. The second contribution is a new technique we call *resource bricolage* that provides a recommended data partitioning scheme to minimize workload execution time in heterogeneous environments. The third contribution is the formalization and solutions for two resource bricolage problems with either a budget constraint or a time constraint. We show that the solution combining both data allocation and resource selection can achieve significant performance improvement over other alternatives.

This dissertation provides a new vision of deploying performance prediction technology in the areas of query optimization, scheduling, and execution, and it also points to promising directions for future studies to improve database performance running in the cloud.

Contents

Acknowledgements **ii**

Abstract **v**

1 Introduction **1**

1.1 The Necessity of Performance Prediction 2

1.2 Potential Use Cases of Performance Prediction 3

1.3 Summary of Contributions 5

2 GSLPI: a Cost-based Query Progress Indicator **8**

2.1 Introduction 9

2.2 Preliminaries 11

2.3 Evaluating MSRPI and WiscPI 14

2.3.1 When Are They Accurate? 14

2.3.2 When Are They Not Accurate? 16

2.4 Our Proposed PI: GSLPI 20

2.4.1 Speed-Independent Pipelines 20

2.4.2 Total and Wall-clock Costs 22

2.4.3 Speed Estimation 27

2.5 Utilizing Runtime Information 28
2.5.1 Refining Cardinality Estimates 28
2.5.2 Refining Cost Estimates 30
2.6 Experimental Evaluation 32
2.6.1 Experimental Setup 32
2.6.2 Effectiveness of Speed-independent Pipeline 32
2.6.3 Effectiveness of Wall-clock Pipeline Cost 34
2.6.4 Effectiveness of I/O Elimination Heuristics 40
2.6.5 Verification 41
2.6.6 Accurate Progress Estimations Challenges 42
2.7 Impact on a Commercial Product 43
2.8 Related Work 44
2.9 Summary 47

3 Resource Bricolage for Parallel Database Systems 48

3.1 Introduction 49
3.1.1 Motivation 49
3.1.2 Our Contributions 51
3.2 The Problem 52
3.2.1 Formalization 52
3.2.2 Potential for Improvement 55
3.2.3 Challenges 59
3.3 Quantifying Performance Differences 60
3.3.1 Estimating the Cost of a Pipeline 61
3.3.2 Measuring Speeds to Process the Cost 62
3.4 Resource Bricolage 63

3.4.1 Base and Intermediate Data Partitioning 64
3.4.2 The Linear Programming Model 65
3.4.3 Allowing Multiple Partitioning Functions 66
3.4.4 Handling Nonlinear Growth in Time 69
3.5 System Specific Challenges 72
3.6 Experimental Evaluation 76
3.6.1 Experimental Setup 76
3.6.2 Overall Performance 78
3.6.3 Execution Time Estimation 81
3.6.4 Investigating Optimal Improvements 83
3.6.5 Handling Nonlinearity 85
3.6.6 Overhead of Our Solution 87
3.7 Related Work 88
3.8 Summary 90

4 Resource Bricolage with Constraints **91**
4.1 Introduction 92
4.1.1 Motivation 93
4.1.2 Contributions 95
4.2 The Constrained Bricolage Problems 96
4.2.1 Problem Definitions 96
4.2.2 NP-hardness of the Problems 97
4.3 Solving the Problems 100
4.3.1 Minimum Time Resource Selection 101
4.3.2 Minimum Cost Resource Selection 102
4.4 Simulation Experiments 103

4.4.1 Experimental Setup 104
4.4.2 Experiments for Minimum Time Resource Selection 106
4.4.3 Experiments for Minimum Cost Resource Selection 112
4.5 Summary 116

5 Conclusion and Future Work 117
5.1 Conclusion 117
5.2 Future Work 118
5.2.1 Progress Indicators on Steroids 118
5.2.2 Improving MapReduce Performance in Heterogeneous Clusters . 122

Chapter 1

Introduction

In recent years, with the growth of the Internet, the world has seen an explosion in the amount of data. Virtually everyone is either experiencing or anticipating an unprecedented growth in the amount of data that will be available to them. As of 2012, we were creating 2.5 quintillion bytes of data every day – so much that 90% of the data in the world today has been created in the last two years alone [7]. Companies, such as Facebook, Google, Microsoft, and Walmart, need to store and analyze massive scale data to maintain their business and to provide customer services.

On the other hand, with the declining cost of hardware, it is possible for companies to set up very large and complex clusters to store and process this growing set of data. To make data-intensive computing accessible to more programmers, many big data processing platforms have been developed, including MapReduce [20] from Google, Hadoop [5] from Yahoo!, Hyracks [14] from UC Irvine, and Dryad [33] and SQL Server Parallel Data Warehouse (PDW) [10] from Microsoft, just to name a few. In fact, we have witnessed an explosive growth in the complexity, diversity, number of deployments, and capabilities of big data processing systems. This growth shows no sign of slowing; if anything, it is

accelerating, as more users bring more diverse data sets and more system builders strive to provide a richer variety of systems within which to process these data sets. To process a large amount of data, modern data management systems typically use massively parallel software running on tens, hundreds, or even thousands of servers. They achieve scalable performance through exploiting data parallelism.

1.1 The Necessity of Performance Prediction

Although these emerging systems can significantly reduce data processing time, due to the massive amount of input data, some tasks may still take minutes or even hours to process. For these long-running tasks, the ability to predict their performance could be very useful. For instance, performance predictions can be used to improve user experience by indicating whether the running tasks are actually making progress or just hanging there. System administrators may also use performance prediction to decide whether they want to cancel a task or allow it to finish. It has been pointed out that the ability to predict job latency is a major customer concern for the next generation of Hadoop [8].

Furthermore, performance prediction is crucial for improving system performance. As systems are getting more and more complex, issues such as hardware/software skew, resource contention, and failures are more likely to arise. All big data systems have to face this unwanted but inevitable fact. Due to all these issues, it gets harder to anticipate the performance of tasks running on such systems, and a one-time decision model used by schedulers, optimizers or resource managers will be vulnerable to system state changes. For example, suppose that a scheduler initially assigns a task of a MapReduce job to a machine based on the current state of the system. Then another program overloads the disk and significantly slows down the task. If the scheduler does not monitor the system and revise its decision, the whole job will be delayed due to this problem. This unexpected

problem can be avoided by monitoring and predicting task performance continuously and rescheduling the task to a different, more lightly-loaded machine. With such a monitor and prediction capability, "users" of the system (be they optimizers, schedulers, resource managers, or even humans submitting and waiting for jobs to complete) may be able to improve the usability and performance of the system.

While the capabilities of today's data management systems continue to grow impressively as measured by the variety, complexity, and quantity of computations they can provide, however, there has not been an accompanying growth in the ability of a system to monitor its state, to make useful performance predictions, and to revise these predictions as time passes. In some sense this situation is surprising. In a wide variety of complex endeavors, humans have built up mechanisms to make predictions, then to monitor and revise predictions as time passes. Examples can be found in weather forecasting, logistics, transportation, project management, and so forth. Airlines do not simply look at a snapshot of current takeoff and landing queues, make a schedule, and stick with it despite changes in weather. Weather forecasts are continuously revised and updated. Shipping schedules are modified in response to unforseen circumstances or changes in demand. Yet in complex data management systems one finds many examples of "users" simply making a scheduling or execution plan based on some ad-hoc collection of information, then sticking with this plan no matter what happens subsequently.

1.2 Potential Use Cases of Performance Prediction

Different components in a system can benefit from performance prediction techniques. For example, unsatisfactory performance of a task may imply that a bad plan is chosen by the optimizer, some indexes are missing, or there is a scarcity of resources. Recently, research work in [69] uses estimated task performance to select straggling tasks

for Hadoop's scheduler. A speculative copy of a lagging task is later scheduled to execute on another machine. The experiment results show that a MapReduce job's response time can be improved by a factor of 2 in clusters of 200 virtual machines on Amazon's Elastic Compute Cloud (EC2) [2]. We believe that this technology can be beneficial to any computation that makes decisions based on performance predictions. The more the computation relies on the predictions, the better decisions it can make by detecting the unexpected state changes. Here, we list several use cases for performance prediction technology, including but not limited to:

1. **User Interface**. Users would always like to get accurate feedback about task execution status, especially if tasks tend to be long-running.

2. **Scheduling**. By using performance prediction technology to obtain the current state of a task and roughly predict its future state, a scheduler can likely do a better job at determining the order in which to run tasks than it could without such information.

3. **Resource Management**. Performance predictions can help in limiting resource contention in a distributed system, and help decide when to re-allocate work, so that system resources are efficiently utilized.

4. **Skew Handling and Stragglers**. By using performance prediction technology, skew can be detected, and the system responds to skew in a way that fully utilizes the nodes in the cluster while still preserving the correct query semantics.

5. **Query Optimization**. Dynamic re-optimization is likely to be increasingly valuable in parallel data processing systems. Performance prediction technology can be useful in providing up-to-date task monitoring data, as well as possible future predictions about processing speed, resource availability and so forth, which can aid in re-optimization decisions.

6. **Performance Debugging**. Debugging queries and system performance is a very challenging task. Performance prediction technology can be useful here as well. For instance, the length of task runtime could be estimated prior to running, and then in the first few minutes of running and then used in identifying the possible reasons for why the process may have slowed down.

1.3 Summary of Contributions

Performance prediction includes many different aspects, such as estimating the execution time or the cache misses of a task on a given machine. This thesis focuses on execution time estimation for database queries. As we discussed before, performance prediction techniques can be used to improve system performance in many different ways, for example, schedule struggling tasks and re-optimize execution plans, and our work aims at improving parallel database performance in heterogeneous environments. The contributions of this dissertation are summarized as below.

Performance prediction. The purpose of this part is to propose a progress indicator to provide accurate execution time estimates for database queries. Typically, a progress indicator [15, 16, 41, 42, 48, 50, 51] estimates how much of the task has been completed and when the task will finish. Progress indicators for SQL queries were first published in 2004 with the independent proposals from Chaudhuri et al. [16] and Luo et al. [41] These progress indicators begin with a prediction of the query running time, and while the query executes, they modify their predictions based on the perceived progress of the query. Both of them also utilize improved information like intermediate result cardinality estimates. In our work, we implemented both progress indicators in the same commercial database system to evaluate their performance. We found that they may provide estimates that are far away from the actual progresses the query is making. The errors arise from their

assumption that in all phases of a query, a tuple takes the same amount of system resources to process. Although their approaches have varying performance for different queries, this problematic assumption is common to both of them. Thus, we proposed a new progress to rectify the problem. The experiment results indicate that our progress indicator can provide significantly more accurate progress estimates. The results also suggest that the remaining progress estimation error is mostly due to cardinality estimation error.

Performance improvement. The purpose of this part is to improve parallel database performance in heterogeneous environments by taking advantage of performance prediction techniques. In a parallel database system, data is typically partitioned across multiple machines to exploit data parallelism. By default, each machine in the cluster will get the same amount of data. In a heterogeneous cluster, the computing capacities of machines may vary significantly. A high-performance machine can finish processing data assigned to it much faster than low-performance counterparts. Unfortunately, the execution time of a parallel query is determined by the slowest machine, and a low-performance machine may substantially degrade system performance. In the first part of our work, we proposed a solution to minimize query execution times by trying to figure out what is the proper amount of data that should be allocated to each machine. Our approach consists of the following three major steps: (i) quantifying differences among machines, (ii) formalizing the data allocation problem as an optimization problem, and (iii) solving the optimization problem for the optimal data allocation scheme. In the second part of our work, we tackled two more problems with additional constraints that are related to the above problem. For the first problem, we need to select a subset of machines from the candidates subject to a *budget constraint*. We want to minimize query execution times with the same amount of money. In the other problem, we need to select a subset of machines from the candidates subject to a *time constraint*. We want to finish the queries at the same time by spending the least amount of money.

The remainder of the thesis is organized as follows. In Chapter 2, we present our work on developing a progress indicator for SQL queries. In Chapter 3, we discuss our solution for improving database performance in heterogeneous clusters. Chapter 4 describes how to carefully select the most suitable machines for running parallel database system so as to achieve better performance with the same budget, or to meet the same performance requirements with lower cost. Chapter 5 concludes the thesis with a summary of our contributions and directions for future work.

Chapter 2

GSLPI: a Cost-based Query Progress Indicator

In this chapter, we implement two state-of-the-art progress indicators in the same commercial RDBMS to investigate their performance. We summarize common cases in which they are both accurate and cases in which they fail to provide reliable estimates. Although there are differences in their performance, much more striking is the similarity in the errors they make due to a common simplifying uniform future speed assumption. While the developers of these progress indicators were aware that this assumption could cause errors, they neither explored how large the errors might be nor did they investigate the feasibility of removing the assumption. To rectify this we propose a new query progress indicator, similar to these early progress indicators but without the uniform speed assumption. Experiments show that on the TPC-H benchmark, on queries for which the original progress indicators have errors up to 30X the query running time, the new progress indicator is accurate to within 10 percent. We also discuss the sources of the errors that still remain and shed some light on what would need to be done to eliminate them.

2.1 Introduction

Many modern software systems provide progress indicators (PIs) for long-running tasks (e.g., file downloads and software installations). Figure 2.1 shows an example of a progress indicator that we are trying to develop for database queries. It continuously updates the elapsed time, estimated remaining time and percentage of completion for a given query.

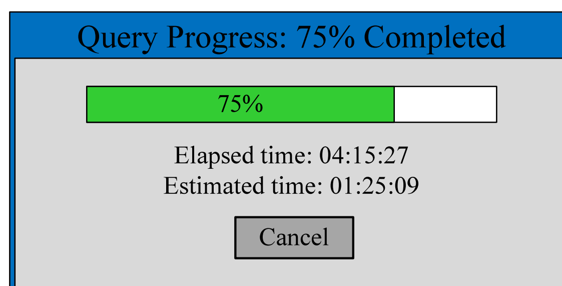


Figure 2.1: Estimated remaining time

Most commercial database vendors provide tools for monitoring queries (e.g., DB2 [19], Teradata [21], Microsoft SQL Server [47], and Oracle [55]). However, none of the existing tools in commercial DBMSs provides accurate progress estimates at a fine granularity. The first progress indicators, the MSRPI [16] and the WiscPI [41][1], were first published by two different research institutes independently in the same conference. The variants [15, 42, 48, 49] of the MSRPI and the WiscPI, were proposed later to explore issues such as broadening the class of queries handled, or investigating the interaction between concurrent queries, or reducing cardinality estimation errors. Despite the passage of time, no work has addressed the quality of the original progress indicators on the problems for which they were proposed.

[1] We name the progress indicators after the institutions where they were developed.

Accordingly, in this work we implemented both the MSRPI and the WiscPI in SQL Server [46] and studied their performance using the TPC-H benchmark. Depending on different query characteristics, we found that there are cases where the MSRPI and the WiscPI are expected to be accurate, and cases in which they fail to provide reliable estimates. For the cases in which they are inaccurate, although there are differences in their performance, the most striking thing we found is that the estimation errors mostly arise from a common simplifying uniform future speed assumption: *at any point in a query's execution, the time to process a unit of work is uniform throughout the remainder of its execution*, where a unit of work is one GetNext() call for the MSRPI and one byte processed for the WiscPI, respectively. While the inventors of the MSRPI and the WiscPI were aware that this assumption could cause inaccuracies, presumably they adopted it to simplify the problem to produce approximate estimates. They did not explore the impact of this assumption on the accuracy of progress indicators. As it is shown by our experiments, this assumption leads to highly inaccurate progress estimates. For some TPC-H queries, the remaining time estimates provided by these two progress indicators are 30 times longer than the actual remaining time. Since their variants inherit this uniform speed assumption, we expect similar performance from them.

Inspired by this observation, we designed and implemented a new *cost-based* progress indicator, called GSLPI, to rectify the problem. The basic idea of GSLPI is to decompose an execution plan into a set of *speed-independent pipelines* delimited by blocking/semi-blocking operators. Then for each pipeline, we estimate its speed of processing the remaining work by utilizing its *wall-clock pipeline cost*. Our experimental results indicate that our approach produces more accurate progress estimates than just making uniform speed assumptions. For some of the TPC-H queries, our progress indicator reduces the estimation errors by more than an order of magnitude.

Finally, we summarize the challenges that we encountered during the development of

GSLPI. Similar to the MSRPI and the WiscPI, our progress indicator also suffers from the well known difficulties of cardinality estimation inherited from the query optimizer. Some of the challenges are still open questions, e.g., skewed data layout, speed fluctuations inside pipelines, etc. We hope that our work can bring these issues to the attention of other researchers and serve as a foundation of building more accurate progress indicators.

The rest of the chapter is organized as follows. Section 2.2 describes the details of the MSRPI and the WiscPI. Section 2.3 presents our experimental evaluation of these two progress indicators. Section 2.4 elaborates on our new cost-based progress indicator. Section 2.5 discusses how to refine cardinality and cost estimates by using runtime information. Section 2.6 experimentally confirms the effectiveness of our techniques and discusses some remaining challenges that need to be resolved. Section 2.7 presents the impact of our work on a commercial database. Section 2.8 briefly reviews the previous work that is related to ours. Finally, Section 2.9 concludes the chapter with directions for future work.

2.2 Preliminaries

In this section, we give a brief overview of the MSRPI and the WiscPI for database queries. Their techniques serve as basic knowledge to understand the discussion in Section 2.3 and our new progress indicator in Section 2.4.

To estimate the progress of a given query, both the MSRPI and the WiscPI use an *execution plan*, which is a tree of physical operators chosen by the query optimizer. The physical operators include the most commonly used operators in a DBMS such as Table Scan, Index Scan, Index Seek, Filter, Hash Join, Merge Join, Nested Loops (NL) Join, Index Nested Loops (INL) Join, Group-by (Hash-based), Sort and Compute Scalar. An operator is referred to as a *blocking operator* if it does not produce any output before it

has processed all tuples in as least one of its inputs. An execution plan is divided into a set of *pipelines* delimited by blocking operators (e.g., Hash Join, Group-by and Sort). A pipeline consists of a set of concurrently running operators. The goal of partitioning an execution plan into multiple pipelines is to gain more insight into the intermediate progress of a query execution.

Every pipeline has a set of *driver nodes*. They are the set of all leaf nodes of the pipeline, except those that are in the inner subtree of a Nested Loops/Index Nested Loops join. Once a pipeline has processed all the tuples in its driver nodes, it finishes execution. In general, for certain pipelines to start executing, one or more other pipelines have to complete. An execution plan can be viewed as a partial order of pipelines, denoted as P_1, .. P_p according to the order in which they are scheduled, where p is the number of pipelines in the plan. Figure 2.2 shows an example execution plan that contains three pipelines (the driver nodes of the pipelines are shaded). Note that this is not the only way for dividing plans into pipelines. In fact, the MSRPI and the WiscPI deploy two slightly different, but essentially equivalent, ways for division. The MSRPI prefers to put a blocking operator together with its descendants which provide input for it, while the WiscPI binds a blocking operator along with its upper operator which consumes its output. In Figure 2.2, we follow the definition used in the MSRPI. For the WiscPI, its three pipelines are P_1 = {Table Scan A, Filter}, P_2 = {Table Scan B}, and P_3 = {Hash Join, Sort}. Both definitions of pipeline are equally adoptable for our solution in Section 2.4.

The MSRPI calculates a query's progress based on the number of GetNext() calls. The total work done by a query is the total number of GetNext() calls issued by all operators in its execution plan. The MSRPI models a query's completion percentage as the fraction of the total number of GetNext() calls that have finished. More formally, suppose the execution plan has n operators. Let N_i ($1 \leq i \leq n$) be the total number of tuples output

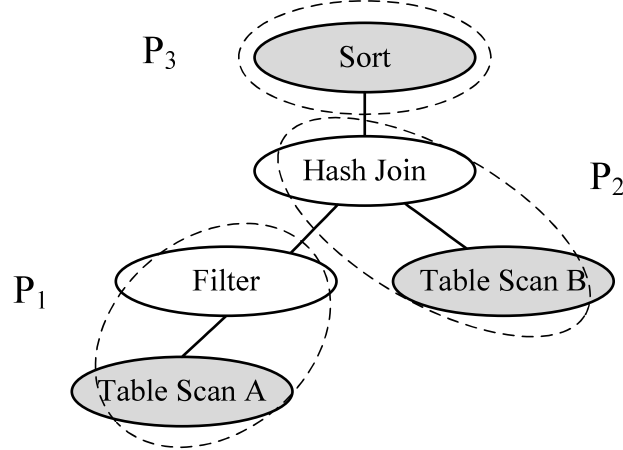


Figure 2.2: An execution plan with 3 pipelines

by operator Op_i (which indicates the number of GetNext() calls made by that operator) throughout the execution of the query, and let K_i be the number of tuples processed by operator Op_i so far. The fraction of a query executed is $percent = \frac{\sum_{i=1}^{n} K_i}{\sum_{i=1}^{n} N_i}$. This GetNext() model assumes that the total time required to execute the query is amortized across multiple GetNext() calls, and therefore the percentage of GetNext() calls done thus far is a good indicator of the time taken by the query.

The WiscPI estimates the remaining query execution time by deploying a model based on the number of bytes processed. It keeps track of the total number of bytes that have not been processed U_i ($1 \leq i \leq p$) in the input and output for each pipeline P_i. The remaining work for a query is therefore the sum of U_i for all pipelines in its plan. Additionally, it records the number of bytes S_i that have been processed by each pipeline P_i in the past T seconds, where T is a pre-defined parameter. The total number of bytes processed in the past T seconds can be thought of as the estimated execution speed of the query. Thus the estimated remaining execution time is $RT = \frac{\sum_{i=1}^{p} U_i}{\sum_{i=1}^{p} S_i}$.

Though the MSRPI does not explicitly give a formula for remaining time estimation, it assumes that the percentage of GetNext() calls done thus far is a good indicator of

the progress it has made toward completion. This corresponds to the idea that if $p\%$ of GetNext() calls have finished, then $p\%$ of execution time has elapsed. In other words, the remaining $(100-p)\%$ of GetNext() calls take $(100-p)\%$ of the execution time. We adopt this interpretation to convert percentage completion for the MSRPI to time remaining.

2.3 Evaluating MSRPI and WiscPI

In this section, we present our experimental evaluation of the MSRPI and the WiscPI on an isolated system where only the given query is running. We also summarize the cases in which they are accurate and the cases in which they fail to provide good estimates.

2.3.1 When Are They Accurate?

While the MSRPI and the WiscPI represent units of work for a SQL query differently for each progress estimate, they both assume that each unit of work in the unexecuted portions takes *the same amount of time to process*, regardless of which pipeline this unit of work belongs to. The speed is assumed to be uniform for the remainder of query and equal to the speed in the past T seconds. We implemented both the MSRPI and the WiscPI in Microsoft SQL Server 12 – the latest version of SQL Server, and tested their performance for different queries. For the WiscPI, we set the number of bytes processed by an operator to be the number of finished GetNext() calls times the average tuple width. The work of a pipeline is the sum of the number of bytes processed by the input and output operators of the pipeline. The datasets and the system setup we used in our experiments can be found in Section 2.6. We set T to be 10 seconds, which is the same as in [41].

Query 1: *select * from partsupp*

The first query that we tested is Query 1 as illustrated above. Its execution plan is

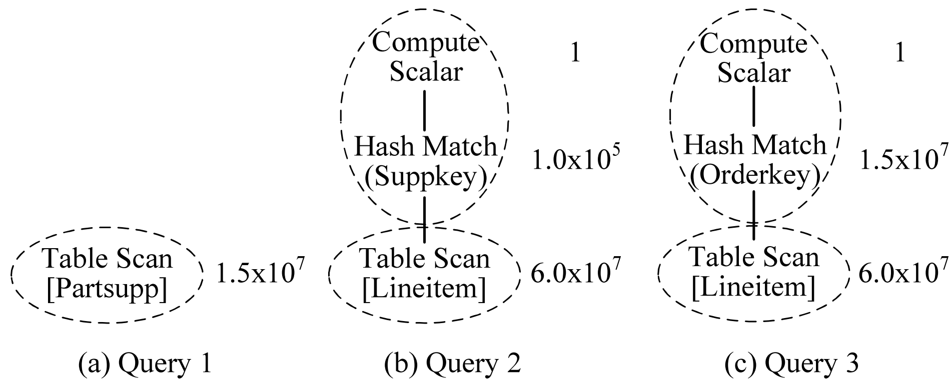
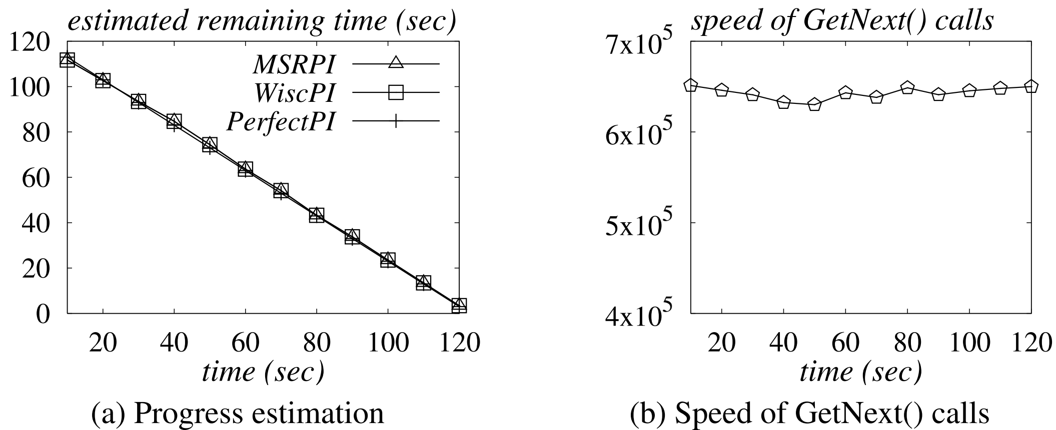


Figure 2.3: Execution plans for tested queries

shown in Figure 2.3a. It consists of only one pipeline, which contains a single scan operator with the output cardinality shown next to it. Figure 2.4a plots the remaining time estimated by the MSRPI, the WiscPI and the PerfectPI. The PerfectPI is a fictitious ideal progress indicator that knows exactly how much time is left for a query. Figure 2.4b shows that the rate of GetNext() calls is approximately constant. We omit the graph depicting the speed in terms of the number of bytes, as it is also approximately constant. Since the speed is stable, both PIs can accurately estimate query progresses. From this example, we can generalize the idea to queries containing a single pipeline: *when a query consists of a single, uniform-speed pipeline, the MSRPI and the WiscPI should be able to accurately estimate the remaining query execution time*.

Query 2: *select count(distinct suppkey) from lineitem*

The second query we tested is Query 2, which consists of two pipelines (see Figure 2.3b). The first pipeline scans tuples in *lineitem* table. In the second pipeline, the Hash Match operator first computes a hash value for each *suppkey* value and inserts it into a hash table; subsequently, after all the *suppkey* values have been inserted, the Compute Scalar operator counts the number of distinct *suppkey* values in the hash table to produce the final answer.



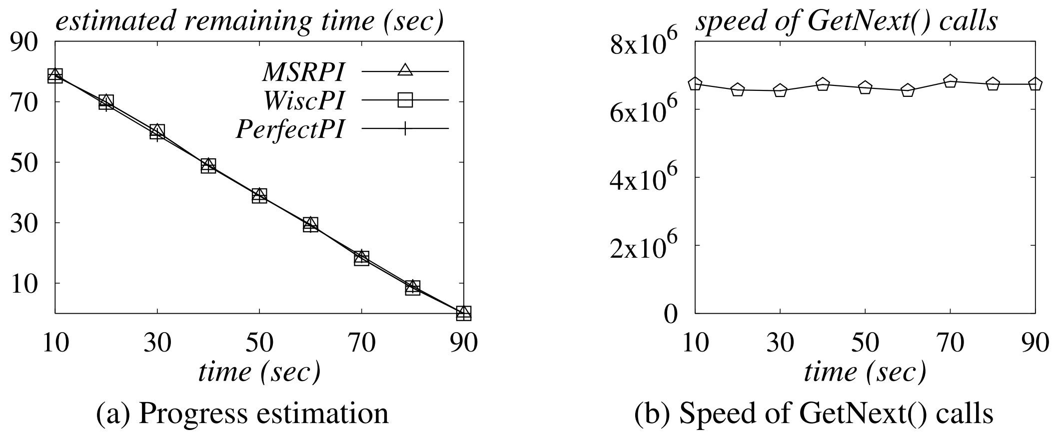
(a) Progress estimation (b) Speed of GetNext() calls

Figure 2.4: Test results for Query 1

The progress estimates produced and the speed of GetNext() calls are shown in Figure 2.5a and 2.5b, respectively. Although Query 2 contains two different pipelines that process tuples at varying speeds, the estimated remaining time is still accurate. Looking further into the query execution, we found that almost all the execution time has been spent on the first pipeline, and nearly all the GetNext() calls were issued within the same pipeline. We refer to such a pipeline as a *dominating pipeline*. The speed of GetNext() calls is stable throughout the query (see Figure 2.5b), since its non-dominating pipeline has almost no effect on the speed. Based on this observation, we can generalize to the following: *As long as PIs have an accurate speed estimate for the dominating pipeline, the remaining time estimate for the query will be accurate, even if they use inaccurate estimates for the other pipelines*.

2.3.2 When Are They Not Accurate?

Unfortunately, there are many queries that do not fall into the cases described above. For example, for the queries in the TPC-H benchmark [31], we found that out of 22 queries only Q_1 contains a dominating pipeline, and only Q_6 contains a single pipeline. The rest



(a) Progress estimation (b) Speed of GetNext() calls

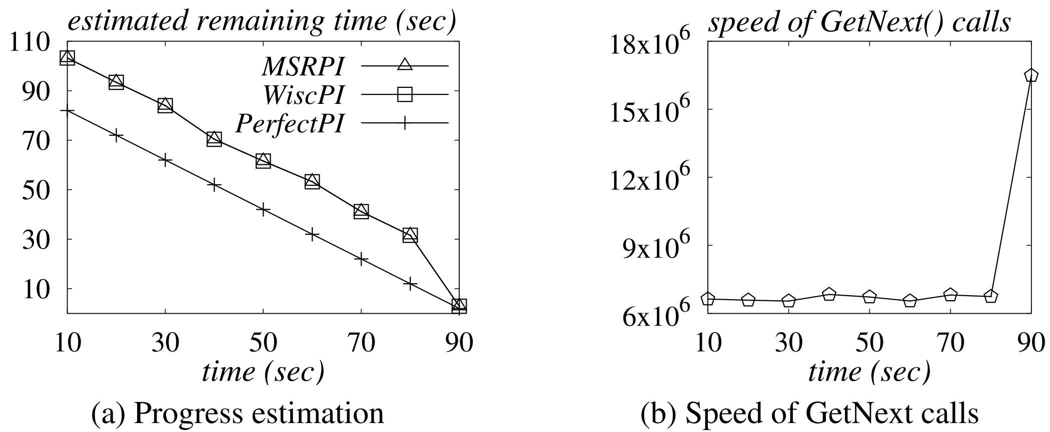
Figure 2.5: Test results for Query 2

of the queries typically contain between 3 and 9 different pipelines. In the following, we evaluate the performance of the progress indicators when a query has non-uniform speeds.

Query 3: *select count(distinct orderkey) from lineitem*

To understand the behavior of these PIs when a query contains multiple pipelines, we tested Query 3, which is similar to Query 2, as shown in Figure 2.3c . We increased the number of GetNext() calls in the Hash Match operator by changing the hash key to *orderkey*, which contains more distinct values.

The test results for Query 3 can be seen in Figure 2.6. The first pipeline P_1 processes about 6.7×10^6 GetNext() calls in 10 seconds, while the second pipeline P_2 can finish 1.5×10^7 GetNext() calls in only 3 seconds. The main reason is that tuples processed by P_1 are brought from disk to main memory, while tuples processed by P_2 are already in memory. When the first pipeline is running, if the PIs use the speed of P_1 to estimate the remaining time of P_2 (that has not yet started), they get $(1.5 \times 10^7)/(6.7 \times 10^6) \approx 22.4$ seconds, which is about 20 seconds slower than the actual execution time. As a result, the remaining time estimated by the MSRPI and the WiscPI is about 20 seconds longer



(a) Progress estimation

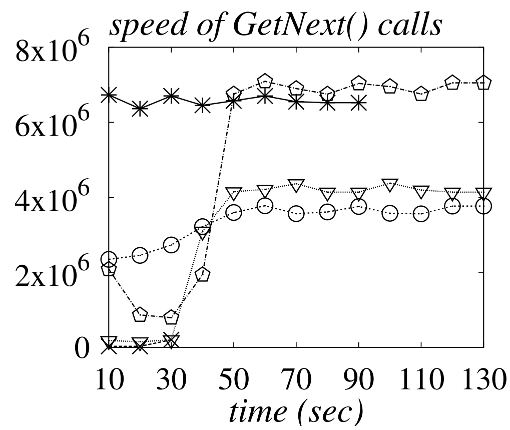
(b) Speed of GetNext calls

Figure 2.6: Test results for Query 3

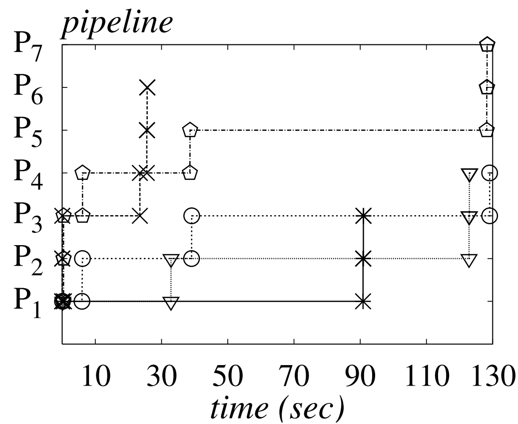
than the PerfectPI. If P_2 contained more tuples, or it could process tuples faster, the gap between the estimated and the true remaining time would increase. From this example, we can see that the MSRPI and the WiscPI make errors when the *speeds of pipelines are different and there is no dominating pipeline.*

TPC-H Example: To have a better understanding of the speed of a query, we tested the speed of GetNext() calls for all 22 TPC-H queries. Nearly all the queries exhibit different speeds in different execution periods, except for Q_1 and Q_6. As we have mentioned before, Q_1 contains a dominating pipeline and Q_6 contains a single pipeline. Figure 2.7a shows the speeds of the GetNext() calls for the first 5 TPC-H queries. As can be seen, four of them change their speeds dramatically during the execution. The execution times of the pipelines in each query are plotted in Figure 2.7b. For a given query, the length of the horizontal line segment on P_i ($1 \leq i \leq 7$) indicates the execution time spent on processing pipeline P_i, and the vertical line segment that goes from P_i to P_{i+1} ($1 \leq i \leq 6$) denotes the end of the execution of P_i and the beginning of the execution of P_{i+1}. If we compare the speed changing points with the pipeline switching points in Figure 2.7a and 2.7b, we can see that *when a query switches from one pipeline to another, its processing*

Q_1 Q_2 Q_3 Q_4 Q_5



(a) Speed of GetNext() calls



(b) Pipeline execution time

Figure 2.7: Experiment results for TPC-H Q_1 to Q_5

speed also changes. During the execution of a pipeline, the speed of processing usually tends to be approximately constant.

The assumption made by the MSRPI and the WiscPI that the speed of the unexecuted portions of the query is equal to the speed in the past T seconds, contradicts the reality that many queries have dramatically different processing speeds for different pipelines.

As a result, these PIs produce inaccurate progress estimates.

2.4 Our Proposed PI: GSLPI

In this section, we present GSLPI[2]: a new cost-based progress indicator for SQL queries. The distinguishing characteristics of GSLPI include: (1) the decomposition of an execution plan into a set of speed-independent pipelines, (2) the utilization of the wall-clock pipeline cost to represent the cost of a pipeline, and (3) the estimation of the speed of each future pipeline based on its wall-clock pipeline cost.

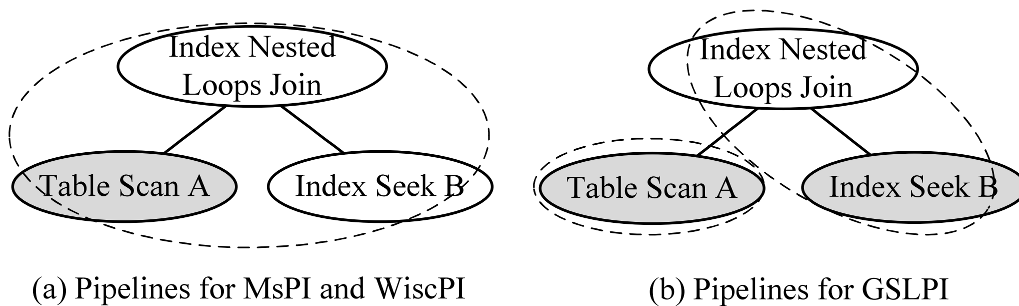
2.4.1 Speed-Independent Pipelines

From Figure 2.7, we can see that each pipeline processes tuples at its own speed, and this speed is usually stable during its execution. Motivated by this observation, we developed an approach for estimating the speed for each individual pipeline. In [16, 41], execution plans are divided into pipelines by blocking operators, and a pipeline may contain a Nested Loops/Index Nested Loops join and its inner and outer subtrees, as shown in Figure 2.8a.

In the case that the physical implementation of the join is an (Index) Block Nested Loops (BNL) join, it behaves like a semi-blocking operator. Note that a Hash Join is a blocking operator, since it must consume all tuples from the build relation before it can produce any output. Compared to Hash Join, (Index) BNL Join must first consume a certain number of tuples from its inner subtree before it can process its outer subtree and produce output. Since the inner/outer subtrees may include different relations and operators, they may process tuples at different speeds. Furthermore, when one of them

[2]Named after Jim Gray Systems Lab, where the progress indicator was developed.

is executing and another one is halted, the speed of the executing operator(s) is independent of the speed of other operator(s). If we group both inner and outer subtrees into a single pipeline, we may mix up two sets of operators with different speeds. Such a combination will cause difficulties in accurate estimations of how fast the pipeline is running. To separate operators processing tuples at different speeds from each other, we define a *speed-independent pipeline*.



(a) Pipelines for MsPI and WiscPI

(b) Pipelines for GSLPI

Figure 2.8: Pipelines for progress indicators

Definition 1. *A **speed-independent pipeline** is a group of interconnected operators that execute concurrently and process tuples at a speed independent of the speeds of other operators (in other pipelines) in the execution plan.*

Based on the above definition, we break an execution plan into a set of speed-independent pipelines and estimate their speeds. A breaking point in a plan is a blocking or semi-blocking operator. Since we introduce additional breaking points (semi-blocking operators), we may break a "traditional" pipeline into finer pieces. For simplicity of discussion, we refer to a speed-independent pipeline as a "pipeline" in the rest of the chapter. The driver nodes of a pipeline are defined as the set of all leaf nodes in the pipeline. Figure 2.8b shows the new pipelines and driver nodes. We adopt the model of work in terms of the number of GetNext() calls involved, and define the work of a pipeline as the total

number of GetNext() calls in the driver nodes of a pipeline. The speed of a pipeline is the amount of work that has been done for the pipeline in the past T seconds, where T is a user-set parameter. Next, we describe our approach to predict the speed for each unfinished pipeline using estimated CPU and I/O costs provided by the query optimizer. The method proposed here is equally applicable to the model where work is defined as the number of bytes processed.

2.4.2 Total and Wall-clock Costs

To process tuples, each operator in a pipeline needs to perform CPU and/or I/O tasks. The cost of these tasks represents the "cost of the pipeline", which is the actual amount of work that takes time to finish. In addition, we make a distinction between total and wall-clock pipeline costs, which is imperative for remaining time estimation.

Definition 2. *The **total pipeline cost** is the total amount of CPU and I/O done by a pipeline from its beginning until the end. The **wall-clock pipeline cost** is the maximum amount of non-overlapping CPU and I/O done by a pipeline from its beginning until the end.*

The difference between the total and the wall-clock pipeline costs is that the total pipeline cost represents the total amount of work that "must be done" by a pipeline during its execution, regardless of whether the execution is parallelized or not. The wall-clock pipeline cost, on the other hand, identifies the non-overlapping parts of work and finds the most expensive one.

For example, consider a plan that consists of only a Table Scan operator on *lineitem*. It has only one pipeline P_1, which is similar to the bottom pipeline depicted in Figure 2.3b. Let us assume that the cost of the operator obtained from the optimizer is *CPU_cost = 66* and *I/O_cost = 767*. Here, the total cost denoted $Tol(P_1)$ is 833, while the wall-clock cost $Clk(P_1)$ is equal to 767. Since CPU and I/O tasks can be performed in parallel by this

operator, the execution time is dominated by the I/O cost, and the CPU cost has a trivial contribution to the overall execution time.

Redistribution of Costs

When an execution plan is chosen and returned by a query optimizer, it typically includes both CPU and I/O costs for each operator. Intuitively, one might think that we can just sum up the CPU and I/O costs of all operators in a pipeline to get its total cost. We found, however, that sometimes this is not true in practice. For an operator Op contained in a pipeline P, it is possible that part of the "work" of Op may be done during the execution of other pipelines. Consider the query from Figure 2.3b where P_1 works as a producer providing tuples to be consumed by P_2. After a tuple is fetched from the base table by P_1, the tuple will be immediately inserted into the hash table by the Hash Match operator. When P_1 is running, P_2 does not generate any output tuples, since it needs to wait until *all* of the tuples output by P_1 have been inserted into the hash table. After P_1 is finished, P_2 reads tuples from the hash table and counts the number of distinct tuples. Though the work of building the hash table is done by an operator that belongs to P_2, it actually occurs during the execution of P_1 and affects the execution time of P_1. Thus, the corresponding cost should be added to P_1 and subtracted from P_2. We consider this kind of cost to be the *post-processing cost* of P_1 that provides the data and the *pre-processing cost* of P_2 that consumes the data.

To do this cost redistribution, for each piece of work we must determine *in which pipelines the work is done* and *how much of that work is done* in each of the pipelines. If an operator is in the middle of a pipeline, all the work done is within the life span of the pipeline. However, there are two operators that are exceptions: Hash Join and Group-By (Hash-based). If these operators are at the boundaries of a pipeline, part of their work

is done in the producer pipeline(s), and part of their work is executed in the consumer pipeline(s). To redistribute the cost of a Hash Match or a Hash Join operator, we must first understand how much of the CPU and I/O is done by each pipeline. The better we can do the distribution, the more precise the time estimation can be attained by the progress indicator. For simplicity, we approximate each part of the work using a set of actions that take roughly the same amount of CPU and I/O.

For a Hash Match operator, the first part (done by the producer pipeline) is to build the hash table. The insertion of a tuple into a hash table contains the following actions: (1) reading the tuple, (2) computing the hash value, (3) finding the right bucket, (4) assigning an empty slot, and (5) inserting the tuple into the hash table. The second part (done by the consumer pipeline) consists of only one action: reading the tuples from the hash table. Let the number of input tuples be a, and the number of output tuples be b. The cost assigned to the producer pipeline is: $5a/(5a + b) \times cost$, and the cost assigned to the consumer pipeline is $b/(5a + b) \times cost$ (the *cost* represents CPU or I/O cost).

Similarly, for a Hash Join operator, its first execution part (done by producer pipeline) consists of building the hash table, and the second part (done by consumer pipeline) is responsible for probing the hash table and outputting the result tuples. The probe by a tuple is modeled as reading the tuple, computing its hash value, finding the right bucket, finding the right tuple and doing the join. Suppose the build child contains a_1 tuples and the probe child has a_2 tuples, and b tuples are output as a result. The cost assigned to the producer pipeline is: $5a_1/(5a_1 + 5a_2 + b) \times cost$, and the cost assigned to the consumer pipeline is: $(5a_2 + b)/(5a_1 + 5a_2 + b) \times cost$.

Note that the need for cost redistribution is not due to our definition of pipeline. The Hash Match operator in Figure 2.3b is doing work for both P_1 and P_2, thus it is tricky to allocate it to one of the pipelines. In this thesis, we assign it to P_2. One may use a different definition and assign it to P_1, or even assign it to both P_1 and P_2. But eventually,

we will still end up with the problem of how much of the work of this operator is done by pipeline P_1 and P_2, respectively. As a result, we can adopt either of these three pipeline definitions as long as we can distribute the cost to the corresponding pipeline correctly.

Calculating the Costs

To calculate the wall-clock cost for a pipeline, we must know which parts of the work in the pipeline overlap. In this section, we present the details of our solution for a simple case, where only one processor and one disk are used for processing a query. However, the idea can be extended to a scenario where multiple processors and disks are used for processing. Since there is one processor and one disk available, the CPU cost and the I/O costs of each operator are expected to overlap. For a set of operators that execute concurrently in the same pipeline, their CPU and I/O costs also overlap. Thus, the wall-clock pipeline cost $Clk(P)$ of a pipeline P is as follows: $Clk(P)$ = max($\sum_{i=1}^{m} CPU(Op_i) + post_CPU(P) - pre_CPU(P)$, $\sum_{i=1}^{m} IO(Op_i) + post_IO(P) - pre_IO(P)$), where m is the number of operators in P and $post_$ and $pre_$ denote the post_processing and pre_processing CPU or I/O cost of a pipeline, respectively. Next, we extend the concepts of total and wall-clock costs of a pipeline to the entire query, and formally define them as follows:

Definition 3. *The **total query cost** is the total amount of CPU and I/O done by a query, and the **wall-clock query cost** is the maximum amount of non-overlapping CPU and I/O.*

Since there is no overlapping work between different pipelines, the wall-clock query cost is the sum of the wall-clock pipeline costs of all pipelines. To understand these different costs better, we calculated these costs for Query 2 and Query 3 (from Figure 2.3). The CPU and I/O costs obtained from SQL Server for Table Scan[Lineitem] are 66 and 767, for Hash Match(Suppkey) are 275 and 0, and for Hash Match(Orderkey) are 670 and 0,

respectively. The cost of Compute Scalar is ignored since it is almost 0. The total and wall-clock costs for pipelines and queries are depicted in Table 2.1.

The Hash Match operator in Query 3 needs to do much more CPU than the same operator in Query 2. As a result, the total query cost (1503) for Query 3 is obviously larger than the total query cost for Query 2 (1108). But from the table, we can observe that the actual execution time of Query 3 is only slightly longer than that of Query 2. The reason is that for both queries, most of the CPU work required to be done by Hash Match is done by pipeline P_1, and pipeline P_1 spends a lot of time doing I/O, which dominates the execution time compared to CPU. As a result, both P_1s in Query 2 and Query 3 take about 90 seconds to finish, and both P_2s finish fast, since they have only a little CPU to do (compared with P_1). For Query 3, we observe that while the number of GetNext() calls issued by P_2 is about 25% of that issued by P_1, the actual execution time is much lower than 25% of P_1's execution time. This is because tuples processed by P_2 are in memory and the average CPU or I/O needed for each tuple in P_2 is much lower than that in P_1.

Query Q	**Pipeline P**	**Tol(P)**	**Clk(P)**	**Tol(Q)**	**Clk(Q)**	**Exe. Time**
Query 2	P_1	1107.9	767	1108	767.1	90 (sec)
	P_2	0.1	0.1			
Query 3	P_1	1471	767	1503	799	93 (sec)
	P_2	32	32			

Table 2.1: Comparison of the costs

In summary, having more GetNext() calls or bytes to process does not imply more CPU or I/O to do, thus it does not lead to longer execution time. The CPU and/or I/O is the actual "work" which needs to be done by a pipeline and takes time to finish; the non-overlapping part of the work (represented by wall-clock pipeline cost) determines the execution time.

2.4.3 Speed Estimation

To estimate the remaining time of a pipeline, we must know how fast it can process its work. Given a pipeline, the driver nodes provide sources of tuples to be processed by the remaining operators. We assume that the total amount of CPU and I/O is amortized across all tuples provided by the driver nodes. How fast tuples are being processed by the driver nodes reflects how fast the CPU and I/O requests are being processed by the entire pipeline. For the purpose of remaining time estimation, it is sufficient for us to consider the total number of tuples in the driver nodes as the total work, and the rate of processing these tuples as the speed.

Estimating the speed for an executing pipeline is straightforward, and the number of tuples processed by its driver nodes in the past T seconds is considered as its processing speed. For a pipeline that is pending, we can predict its speed based on its wall-clock pipeline cost and how fast the system can process CPU and I/O tasks. For a pipeline P_i, suppose its wall-clock cost is C_i, and the total number of tuples in its driver nodes is N_i, among which K_i have been processed. Let S_i be the speed that P_i can process its tuples in driver nodes. Without loss of generality, suppose that P_1 is the running pipeline and P_2 is the pipeline of which the speed that we want to predict. We assume that the system can process the same amount of CPU or I/O for a query in every T seconds. Let S_1 be the speed of P_1, and S_2 be the speed of P_2 that needs to be predicted. We have $\frac{C_1}{N_1} \times S_1 = \frac{C_2}{N_2} \times S_2$. Then the speed of P_2 is:

$$S_2 = S_1 \times \frac{C_1 \times N_2}{N_1 \times C_2}.$$

The remaining time for an unfinished P_i can be estimated as $(N_i - K_i)/S_i$, and the remaining time for the entire query is the sum of the remaining time of all pipelines. In the

formula above, the amount of CPU or I/O processed for this query in T seconds equals the amount of cost finished by the running pipeline in the past T seconds. In case the running pipeline has random speed fluctuation, we can take the average rate for processing the CPU or I/O by the running pipeline in the past to smooth the estimation.

The new speed estimate for an unfinished pipeline using its wall-clock pipeline cost can be much closer to its actual speed than one generated using the uniform speed assumption. For example, for Query 3 in Figure 2.6, when P_1 is running, the speed of P_2 is estimated as $6.7 \times 10^6 \times \frac{767 \times 1.5 \times 10^7}{6 \times 10^7 \times 32} \approx 4 \times 10^7$ GetNext() calls per 10 seconds. The estimated remaining time of P_2 is $\frac{1.5 \times 10^7}{4 \times 10^7} \times 10 \approx 3.75$ seconds, which is close to the actual value (about 3 seconds). By contrast, the MSRPI assumes that $S_2 = S_1 \approx 6.7 \times 10^6$, and gives an estimate of 22.4 seconds.

2.5 Utilizing Runtime Information

As is the case in query optimization, a key challenge for progress indicators is to accurately estimate cardinalities and costs of operators in the query plan. In the following, we describe how to collect execution feedback to continuously refine the cardinality and cost estimates.

2.5.1 Refining Cardinality Estimates

Both the MSRPI and the WiscPI collect information to refine cardinality estimates. The refinement in the MSRPI is based on refining upper/lower bounds of cardinality estimates [16], while the WiscPI relies on linear interpolation [41]. Since these two methods are compatible and each provides additional information for refining cardinalities, we adopt both upper/lower bounds and linear interpolation to refine cardinality estimates.

Before GSLPI estimates the remaining time of a query, the cardinality of each unfinished operator is first computed using linear interpolation. For an operator in a running pipeline, the percentage of the tuples in the driver nodes that have been processed is used to refine the output cardinality. Let E_d be the input cardinality of the driver node and K_d of them have been processed, then the percentage of the tuples that has been processed is $p = K_d/E_d$. For an operator (other than driver nodes) in the same pipeline, suppose its original estimate of cardinality is E_1 (before the pipeline has started executing) and it has processed K tuples. The new cardinality estimate based on linear interpolation is then $E_2 = K/p$. In case that the pipeline has only one driver node, a heuristic formula is used to estimate the final output: $E = p \times E_2 + (1 - p) \times E_1$. In cases where a pipeline contains more than one driver node, the driver node which finishes processing relatively fast is chosen to calculate the value of p. Then the upper bound and the lower bound are calculated for each unfinished operator and are used to further refine their cardinality estimate. Intuitively, for each operator the estimated number of output tuples should never be less than the number of tuples seen so far (i.e., the lower bound). For most operators except joins, the estimated number of output tuples should never exceed the number of input tuples (i.e., the upper bound). For more details on upper/lower bounds, we refer the reader to [16].

Normally, the output cardinality of an operator does not affect the costs of other operators, unless it is a descendant of those operators. However, if the operator is in the outer subtree of a Nested Loops (NL) join or an Index Nested Loops (INL) join, it may affect both the CPU and I/O costs of the operators in the inner subtree. If the outer subtree produces more (fewer) tuples for the NL/INL join, the number of executions of the inner subtree may increase (decrease) as well (this is known as as *Rebinds* and *Rewinds* in SQL Server [45]). As a result, for a NL/INL join, the estimated output cardinality of the outer subtree should be propagated to the inner subtree to refine the CPU and I/O

cost of the operators. Take the query plan in Figure 2.8 for example. If the number of tuples produced by Table Scan operator increases, the number of index seeks done by inner subtree also increases. Suppose the original and updated estimated output tuples for the outer subtree is E_1 and E_2, respectively. Let Ex_1 be the original estimated number of executions of the inner subtree. Then the updated number of executions of the inner subtree is $Ex_2 = E_2/E_1 \times Ex_1$. The updated cost (CPU or I/O) for each operator in the inner subtree is the cost for one execution times the updated number of executions.

2.5.2 Refining Cost Estimates

Since the wall-clock pipeline cost is critical to the accuracy of GSLPI, when the cardinality estimate of an operator changes, we also need to revise its cost estimate accordingly. The cost refinement is based on algebraic properties of the operator. For every operator, we use a function f to approximate its cost estimate with respect to its properties and cardinalities. If the cost of an operation increases linearly with the input cardinalities (e.g., Table Scan, Index Scan, Filter, etc.), the function is simply $f(N) = N$, where N is the input cardinality of the operator. Suppose C_1 is the cost (CPU or I/O) obtained from the optimizer for an operator when its input cardinality is N_1. When its input cardinality changes to N_2, its cost gets updated to $C_2 = C_1 \times \frac{N_2}{N_1}$. For costs that do not increase linearly with respect to input, a more complicated function is used.

Estimating I/O costs is even more error-prone than estimating CPU costs. This is due to the following two reasons: (1) the memory granted to the query may be different from the available memory assumption made by the optimizer, and (2) the execution of an operator may bring in the data needed by another operator that runs sometime later. To alleviate the problem caused by I/O estimation error, we introduce two heuristic methods to eliminate the I/O cost of an operator, if it does not need to do any I/O (e.g., all the

data can fit in memory). We first consider the maximum amount of memory required by a pipeline. For a non-blocking operator in a pipeline, a small fraction of memory is sufficient, since once a tuple is output by a non-blocking operator, it will be immediately propagated on to the next operator (if there is any). For a blocking operator, tuples are collected in the operator until all the input tuples are consumed, therefore, blocking operators are memory consuming and take up most of the memory. Thus, given a pipeline, the memory-consuming operators include (i) a Sort or a Group-by (Hash-based) operator providing data for the pipeline, (ii) a Hash Match or a NL/INL join operator inside a pipeline, and (iii) a blocking operator that takes in the output tuples of the pipeline. We ignore the amount of memory used by the non-blocking operators. Then the maximum amount of memory required by a pipeline is defined as the total amount of memory that is needed to hold the data for the three types of memory-consuming operators above.

If the memory available is more than the maximum amount of memory required by a pipeline, none of these memory consuming operators needs to do any I/O when their corresponding pipeline executes. Thus, we can safely remove this part of I/O cost from the wall-clock pipeline cost of the pipeline. Let P_i and P_j $(i < j)$ be two different pipelines that contain a same subtree of operators, which generate the same intermediate results. In this case, SQL Server will detect the common subtrees and try to reuse the intermediate results. We check whether the intermediate results generated by P_i are still in memory when P_j runs. Let M_i be the maximum amount of memory required by pipeline P_i, and $Size_r$ be the size of the intermediate results. If $\sum_{k=i}^{j} M_k + Size_r$ is less than the available memory for this query, the data brought in by P_i should still be in memory when P_j runs. We subtract all the costs (both CPU and I/O) of the operators in the subtree from the wall-clock pipeline cost of P_j.

2.6 Experimental Evaluation

This section presents experimental results showing the effectiveness of our proposed techniques. We first describe the experimental setup, and then evaluate the performance of our progress indicator and compare it to the MSRPI and the WiscPI.

2.6.1 Experimental Setup

We implemented all three progress indicators in Microsoft SQL Server 11. Our experiments use a TPC-H 10GB database with all tables stored on a single disk. We measured the performance of the progress indicators using all 22 queries in the TPC-H benchmark. Most of them contain more than 8 operators and 4 different pipelines. The experiments were run on a machine with an Intel Core 2 Duo CPU, with 8 gigabytes of memory. In the experiments, only the database server was executing, and we run each query one at a time. When a query is running, a progress indicator wakes up periodically (every 10 seconds, which is the same as the setting in [41]), collects the runtime information, and estimates the remaining execution time of the query. For all queries, GSLPI provides progress estimates with less than 1% overhead, and this is similar to the overhead introduced by the MSRPI or the WiscPI. With a smaller T, the PIs could adapt quicker to changes in speed, but it would not substantially improve the performance of either the MSRPI or the WiscPI with respect to the main issue (the uniform speed assumption) we address.

2.6.2 Effectiveness of Speed-independent Pipeline

To justify the necessity of using semi-blocking operators for dividing pipelines, we tested Query 4, a simple and easy to understand example that well illustrate our point. Its execution plan is shown in Figure 2.9, which contains a Nested Loops join operator. The

remaining time predictions made by the MSRPI, the WiscPI and GSLPI are ploted in Figure 2.10.

Query 4: *select * from nation loop join customer on nationkey = 1*

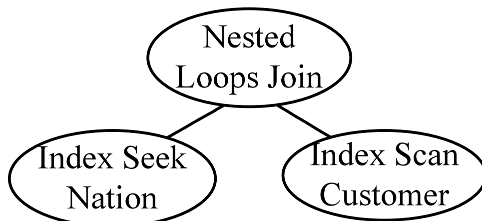


Figure 2.9: The execution plan of Query 4

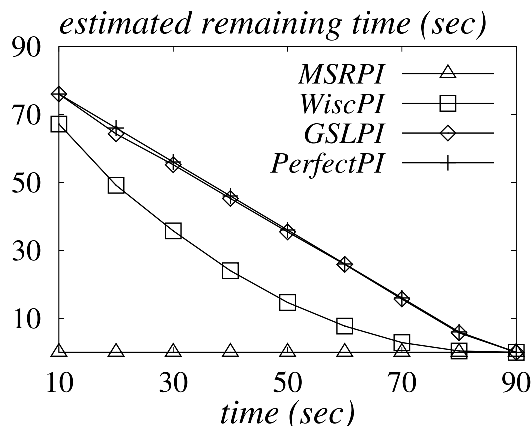


Figure 2.10: Progress estimation for Query 4

For the MSRPI and the WiscPI, all operators belong to the same pipeline, and the driver node is the Index Seek Nation operator (refer to Section 2.2 for its definition). The MSRPI makes a driver node hypothesis, which says that the overall query progress can be estimated by the progress of only the driver node(s) of the pipeline. Unfortunately, this is not true for pipelines containing semi-blocking operators, where the execution of the inner subtree is independent of the execution of the outer subtree. For example, when the Index Seek operator in Query 4 finishes processing all its tuples at the very beginning,

the MSRPI assumes that the entire pipeline is finished, and its estimated remaining time is 0 after that. But in fact, the Nested Loop operator and its inner subtree are still running and takes about 90 seconds to finish. The WiscPI makes a similar assumption about the driver node(s) (called dominant input(s) in the original paper). Although it uses a heuristic formula to smooth fluctuations, which prevents it from jumping to 0 directly, its estimates are still not quite accurate. Our GSLPI splits the query plan into two speed-independent pipelines, which truly represent two sets of operators that run independently, thus produces accurate predictions. For TPCH queries, we also observe similar cases in which the inner subtree starts running independently after the outer subtree consumes a certain amount of tuples and halts. For these cases, GSLPI exhibits better performance.

2.6.3 Effectiveness of Wall-clock Pipeline Cost

In this section, we examine the accuracy of our progress indicator. We first show the overall performance of GSLPI for TPC-H queries, then provide an analysis for our progress indicator and compare its remaining time estimates with the other progress indicators.

Overall Performance

To evaluate the performance of our progress indicator, we employ the evaluation metric called estimation error used in [16, 50]. Assume that the query starts at t_0 and ends at t_n. Then let t_i ($t_0 \leq t_i \leq t_n$) be the time when the progress estimation is taken, and RT_i be the estimated remaining time. Together with RT_i, a progress indicator also returns a value f_i derived from RT_i to indicate the percentage of the time completed: $f_i = 100(t_i - t_0)/(RT_i + t_i - t_0)$. The estimation error at time t_i is defined as:

$$e_i = |\frac{100(t_i - t_0)}{(t_n - t_0)} - f_i|,$$

where $100(t_i - t_0)/(t_n - t_0)$ represents the actual percent-time done. For each query, we calculate its average and maximum estimation error. The results are shown in Table 2.2. As we can see from the table, the average error is typically small (only 3 of them are above 5%), and the maximum error is usually below 10%. The larger errors in the estimates made by our progress indicator (e.g., Q_3, Q_{12}, Q_{20} and Q_{21}) are due to the cardinality estimation errors inherited from the query optimizer (see the verification in Section 2.6.5).

Query	**Mean**	**Max**	**Query**	**Mean**	**Max**
Q_1	0.5%	0.6%	Q_{12}	10.5%	24.7%
Q_2	0.9%	1.9%	Q_{13}	1.2%	3.5%
Q_3	4.0%	10.4%	Q_{14}	1.0%	2.2%
Q_4	1.5%	8.5%	Q_{15}	0.1%	0.5%
Q_5	2.4%	10.3%	Q_{16}	1.6%	4.0%
Q_6	0.3%	0.7%	Q_{17}	0.5%	0.9%
Q_7	3.6%	9.4%	Q_{18}	0.3%	1.6%
Q_8	3.2%	8.2%	Q_{19}	0.2%	0.5%
Q_9	1.3%	6.3%	Q_{20}	16.8%	41.1%
Q_{10}	1.7%	7.7%	Q_{21}	5.7%	16.0%
Q_{11}	1.1%	3.2%	Q_{22}	1.1%	3.6%

Table 2.2: Estimation errors in TPC-H queries

Utility of Wall-clock Pipeline Cost

In this experiment we use TPC-H query Q_1 to show the necessity of redistributing the CPU and I/O costs to the pipelines where the actual work is being done. We tested and compared our GSLPI against a modified version that does not perform redistribution. The non-redistributing version of our progress indicator simply sums up the CPU and I/O costs of all operators inside a pipeline respectively, and chooses the larger value as the cost of the pipeline. We use GSLPI$_{dis}$ to denote our original GSLPI, which redistributes the cost

to get the wall-clock pipeline costs, and GSLPI$_{nod}$ to denote the variant. The comparison of these two progress indicators for Q_1 is depicted in Figure 2.11.

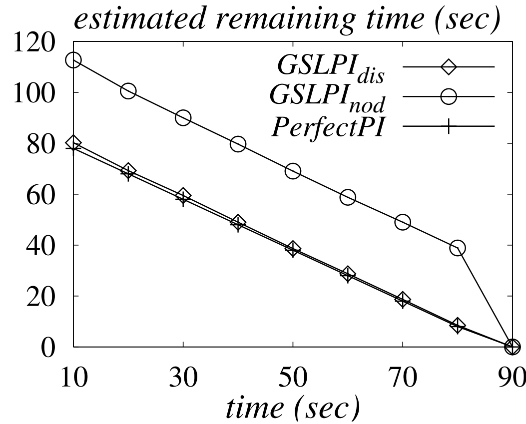


Figure 2.11: Progress estimation for Q_1

The graph above illustrates that GSLPI$_{dis}$ is almost identical to the PerfectPI, while the estimates of GSLPI$_{nod}$ are at least 40% longer than the actual remaining time. The reason is when the first pipeline in Q_1 scans a table, it also does most of the work (building the hash table) for the Hash Match operator in the second pipeline. Since I/O takes a longer time to finish, the execution time for building the hash table (by doing CPU) is excluded from the estimates. Without redistribution, GSLPI$_{nod}$ assumes that this part of work is done in the second pipeline, and it takes about 32 seconds. This assumption leads to the error gap between GSLPI$_{nod}$ and the PerfectPI in the graph.

The improved WiscPI [42] suggested an idea of using the CPU and I/O costs of the input and output operators in the pipeline to scale its speed. It also does not consider the redistribution of the costs among the pipelines as well. As a result, in the best case the improved WiscPI can provide estimates similar to that in GSLPI$_{nod}$. Since Hash Match and Group-By operators are common in query plans (e.g., 21 out of 22 TPC-H execution plans chosen by the SQL Server optimizer contain this kind of operator), the idea of cost

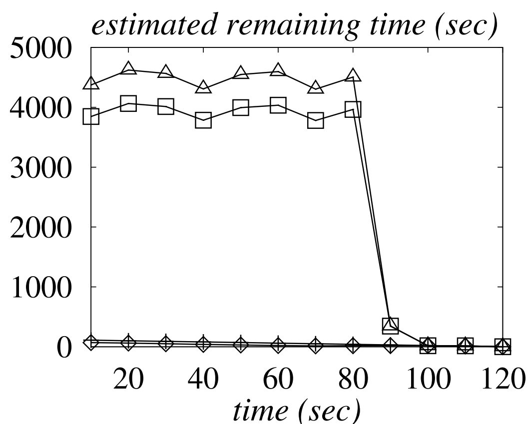
redistribution must be deployed if better progress estimates are desired. By introducing the wall-clock pipeline cost, we are able to address this problem and get more accurate estimates.

Comparison of Progress Estimates

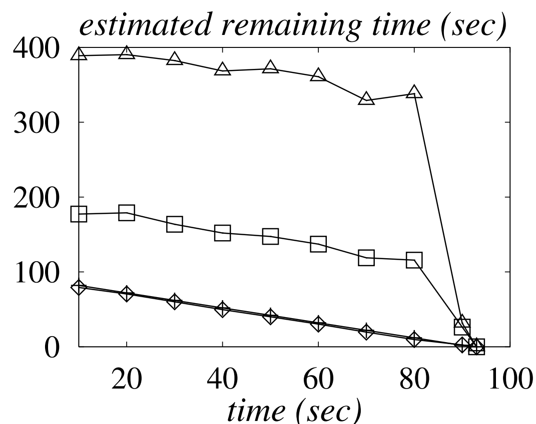
In this section, we show that using wall-clock pipeline costs to scale the speeds of the pipelines leads to better estimates than making the uniform speed assumption. We tested all the TPC-H queries and compared the estimates provided by the MSRPI, the WiscPI, GSLPI and the PerfectPI. From our experiments, we observed improvements for 20 queries (for the remaining 2 queries, all the progress indicators produce accurate results).

We illustrate the results for Q_{12} in Figure 2.12a. Q_{12} takes about 120 seconds to finish, and it spends around 90 seconds on P_1 and around 30 seconds on P_2. Although it has 4 different pipelines, only the first two pipelines are important in our discussion (the other two finish very fast). When P_1 is running, the estimates of both the MSRPI and the WiscPI are over 4000 seconds, which are far away from the PerfectPI as plotted at the bottom in the figure. GSLPI has an average estimation error of only 10.5%.

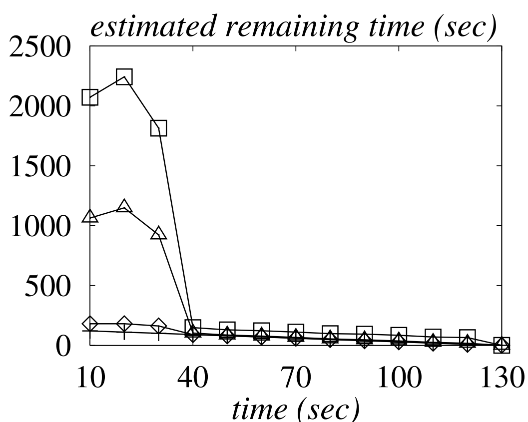
To identify the reasons for the errors made by the MSRPI and the WiscPI, we measured the speed of GetNext() calls. As illustrated in Figure 2.13, the GetNext() calls speed of P_1 is much slower than that of P_2. In fact, P_1 processes about 3.4×10^4 GetNext() calls every 10 seconds, while P_2 processes 5×10^6 GetNext() calls in the same amount of time. Since P_2 has 1.56×10^7 GetNext() calls, if it also processed tuples with the same speed as P_1 (the uniform speed assumption), it should take more than 4000 seconds to finish. However, in reality it finishes in about 30 seconds. Similarly, for the WiscPI, the number of bytes processed (computed as the number of GetNext() calls times the average tuple size) is also slow for P_1 and fast for P_2. As a result, it generates similar progress esti-



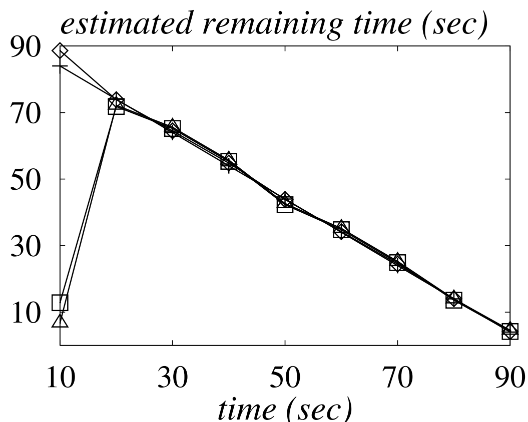
(a) Query Q_{12}



(b) Query Q_{14}



(c) Query Q_{10}



(d) Query Q_{19}

Figure 2.12: Progress estimation for TPC-H queries

mates. GSLPI, on the other hand, utilizes the wall-clock pipeline costs of P_1 and P_2 to estimate the speed of P_2 when P_1 is running. Since the wall-clock pipeline cost of P_2 is smaller than that of P_1 and the number of GetNext() calls in P_2 is much larger than that in P_1, the per tuple cost of P_2 becomes far smaller than P_1, which suggests that each tuple

in P_2 takes much less time to process. By taking advantage of this information, GSLPI can obtain a more accurate speed estimate for P_2 and produce a more precise remaining time estimate.

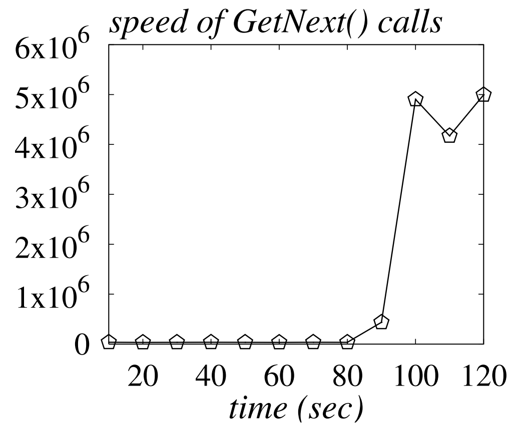


Figure 2.13: Speed of GetNext() calls for Q_{12}

As shown in Figure 2.12b, we obtain similar results for TPC-H Q_{14} for the same reason. The performance of the MSRPI and the WiscPI is better compared to that for Q_{12}, but the estimated remaining time is still much longer than the actual remaining time. The GSLPI's results, however, are nearly identical to the PerfectPI's estimates.

If we compare the MSRPI against the WiscPI, we can see that the WiscPI is the winner for Q_{12} and Q_{14}. But according to our understanding, it is challenging to tell which one produces more accurate estimates in general. Figure 2.12c demonstrates a case where the MSRPI performs better than the WiscPI. For these two queries, a slow pipeline P_s starts first, and both progress indicators must estimate the speed of the fast pipeline P_f that runs later. When P_s is running, WiscPI takes the product of the speed of GetNext() calls and the average tuple size of P_s and uses it as the speed of P_f. If the average tuple size of P_s and P_f are the same, the estimates made by the MSRPI and the WiscPI will be the same. But in Q_{14}, the average tuple size of the slow pipeline happens to be larger than that of the

fast pipeline, thus, the WiscPI ends up using a faster speed for P_f. As a result, the WiscPI provides more accurate progress estimates. The opposite case, where the slow pipeline has a smaller average tuple size, happens in Q_{10}, which makes the MSRPI perform better. Since more tuples or bytes does not necessarily imply longer execution time, using them for time estimation is inadequate.

The three example queries above demonstrate that when a slow pipeline runs first, both the MSRPI and the WiscPI overestimate the execution time for the queries. However, GSLPI is able to address this problem by using the wall-clock pipeline cost approach. Figure 2.12d shows a case when a fast pipeline starts first. In this case, both the MSRPI and the WiscPI underestimate the execution time, whereas GSLPI also handles this case successfully.

2.6.4 Effectiveness of I/O Elimination Heuristics

To show the effectiveness of the proposed I/O elimination heuristics, we tested another modified version of GSLPI$_{noh}$, where no elimination heuristics are used. In this section, we compare the performance of GSLPI$_{noh}$ with our original progress indicator, denoted as GSLPI$_{heu}$ here. Significant improvements can be observed for two queries, namely Q_{11} and Q_{18}. Figure 2.14 shows the result for Q_{18}. As can be seen, GSL$_{heu}$ progress indicator is almost identical to the PerfectPI, while GSL$_{noh}$ underestimates the execution time at the beginning. When we look into the execution plan, we find that there is one Hash Match operator, which behaves very different from what is suggested by the query optimizer. The I/O cost provided by the optimizer for this operator indicates that it should have some I/O work to do, but when the query is running, it obtains enough resources to store the entire hash table in the main memory. Thus, no I/O is actually performed. GSL$_{noh}$ does not consider this runtime information: when the first pipeline P_1 is running

at the beginning, GSL_{noh} assumes that P_1 is performing the I/O and gets a bigger wall-clock pipeline cost for P_1. Since no I/O is actually performed and tuples get processed quickly, GSL_{noh} believes that the system processes the CPU or the I/O tasks fast, and thus the query will finish in a short time. GSL_{heu}, on the other hand, collects the runtime information for memory. Since the granted memory is more than the required memory, GSL_{heu} subtracts the I/O cost from the Hash Match operator's cost. The revised I/O cost depicts more accurately what happens in reality.

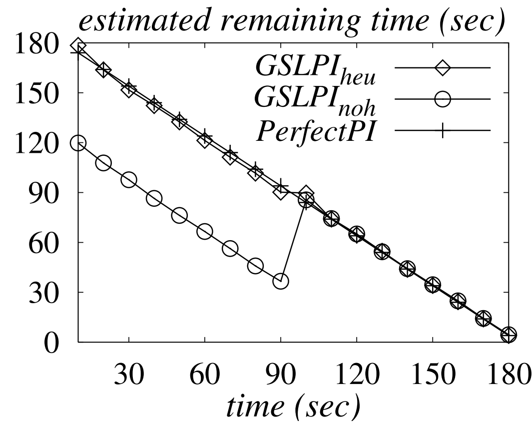


Figure 2.14: Progress estimation for Q_{18}

2.6.5 Verification

For all the TPC-H queries that we tested, most of the estimation errors made by GSLPI are actually due to *cardinality estimation errors*. To verify this, we tested our progress indicator based on true cardinalities (obtained after one execution of the query). For the problematic queries shown in Table 2.2, significant improvement occurs to Q_3 Q_{12}, Q_{20}, and Q_{21}. Figure 2.15 shows the estimated remaining time with the actual cardinalities (denoted GSLPI_{act}) and the cardinalities obtained from the optimizer at compile-time (denoted as our original progress indicator GSLPI_{est}). Since the cardinality estimates for

5 time-consuming operators (in 3 different pipelines) are significantly wrong, this leads to most of the remaining time estimates to become quite inaccurate as well. With the true cardinalities, GSLPI can successfully estimate the speed of each pipeline, thus the remaining time of the query.

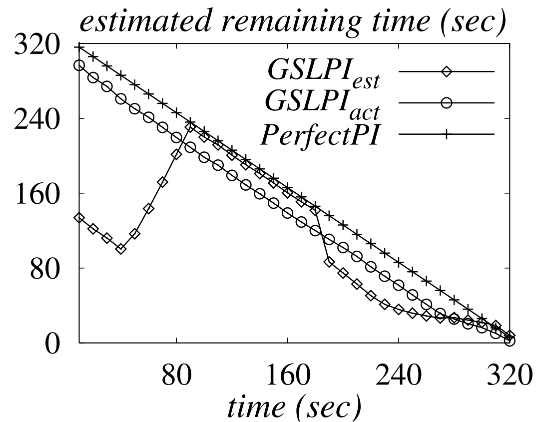


Figure 2.15: Progress estimation for Q_{21}

2.6.6 Accurate Progress Estimations Challenges

In the above sections, we illustrate the improvement that we made over the MSRPI and the WiscPI. In the following, we present the challenges that we encountered, with a hope of inspiring the development of more accurate progress indicators.

A key challenge for GSLPI (as well as for any other progress indicators) is accurate cardinality estimation. This problem has been faced by query optimizers for a long time, and all the progress indicators proposed so far suffer from the cardinality estimation errors. As we mentioned above, cardinality estimation errors contribute to most of the errors made by GSLPI. In addition, if we can not provide a robust guarantee for the problem of cardinality estimation, it is impossible for us to develop a progress indicator with guarantees.

Even if we could obtain accurate cardinality estimates for a query, the layout of the data may also affect the estimation of the progress. For the queries that we consider, the processing speeds of most pipelines usually tend to be constant. If we have a skewed distribution of the data, the speed may vary during the execution. For example, if most of the tuples which satisfy the select conditions are clustered at the beginning of the table, the speed may increase after finish processing these tuples. A progress indicator which does not aware of the layout of the data is likely to produce inaccurate estimates.

Another challenge problem we have found is the Nested Loops join operator in Q_{20}. The inner subtree consists of only one Index Seek operator, and the speed of GetNext() calls for the pipeline in the outer subtree keep increasing quickly. One possible reason for the increasing speed is that the index seeks finished earlier brought in pages from disk needed by the Index Seeks operator executed later on. To provide accurate remaining time estimates for this Nested Loops join, we must be able to model or predict its speed for processing tuples. We do not have a satisfying solution for it so far, and more effort must be made for solving this problem.

In addition, the speed may be affected by available resources, interaction among different parts of the query, and queries that arrive at or leave the system, etc. While multi-query progress indicators are clearly the ultimate goal, it is our belief that identifying and resolving problems in this simplified setting is a useful step in moving toward addressing more complicated problems related to progress estimation.

2.7 Impact on a Commercial Product

Presently, the existing tools in commercial DBMSs provide little feedback on the progress of a running query. This feedback can be extremely beneficial to both end users and internal users such as database developers. End users can utilize this information to decide

whether they want to cancel a long running query or allow it to continue, while internal users can consume this information to diagnose performance related issues. Today, SQL Server Management Studio (SSMS) and other related tools built for commercial DBMSs deliver information about estimated and the actual query plans for a given query. However, these query plans do not contain any run-time information (e.g., rows processed, memory consumption, etc.) that can be of significant assistance in debugging performance issues.

As of the time when this dissertation was written (June 2014), we collaborated with Microsoft Research and a SQL Server team to provide live query statistics for running queries. This project is focused on displaying progress of in-flight queries in SSMS. The new monitoring tool will provide the ability to view in-progress queries from SSMS. The active query plan will contain the following information: (i) the set of query operators, (ii) runtime statistics for each query operator, and (iii) an additional progress bar showing overall query progress. During the time when the query is executing, the progress bars and runtime statistics of individual operators will be updated periodically. When the query finishes, the progress bar will indicate its completion. We are now integrating the main ideas proposed in this chapter for estimating overall query progress into this monitoring tool for SQL Server 2014. We used two benchmark workloads (TPC-H and TPC-DS) and three real-world workloads to test our current implementation. It offers a significant improvement over other alternatives we have tried.

2.8 Related Work

Recently, there has been an increasing interest in the development of progress indicators for database queries. Previous work can be roughly classified into three categories: commercial progress indicators, research progress indicators, and techniques that can be

useful for query progress estimation. In the following, we survey these three categories of work.

The first category includes progress indicators provided by commercial database vendors. Tools for monitoring queries are available in DB2 [19], Teradata [21], Greenplum [30], SQL Server [47], and Oracle [55]. Some progress indicators collect and return statistics (e.g., number of rows and pages processed, current execution operators, etc.) for a given running query. Some progress indicators [21, 30] decompose an execution plan into a number of steps, and indicate which steps are completed/running. Some progress indicators calculate the percent-complete for long running operators [55] or percent-complete for certain validation and recovery statements [47]. These progress indicators are simple and coarse-grained.

The second category includes progress indicators for database queries proposed by research groups. They aim at providing estimates at sufficiently fine granularity. Two pioneering progress indicators are introduced in [16] and [41], respectively. We refer to them as the MSRPI and the WiscPI in the context of this thesis. Both the MSRPI and the WiscPI adopt the idea of dividing a plan into a set of pipelines. The MSRPI calculates the percentage of GetNext() calls finished as an estimation of the current query progress. The follow-up work [15] proves that in the worst case, it is impossible for the MSRPI to provide robust guarantees for the problem of progress estimation. The WiscPI, on the other hand, estimates the remaining execution time by modeling the work of a query as the number of bytes processed at the input and output of pipelines. The two follow-up papers [42, 43] on the WiscPI aim to increase the coverage of the progress indicator to a wider set of SQL queries and extend the single-query progress estimation to enable progress estimation for multiple queries. In [49] and [48], the authors propose a lightweight progress indicator, and they focus on improving cardinality estimation accuracy for various operators in the query plan. Since refining cardinality estimates is not the focus of our work,

we do not incorporate them into our progress indicator. Recently, machine leaning techniques have been adopted for query performance prediction. Kernel canonical correlation analysis is used in [27] to find the relationship between query plan feature matrices (e.g., number of joins and the cardinality sum for the query) and performance feature matrices (e.g., execution time and number of disk I/Os). Regressions are used in [23] to predict the execution time of concurrently running queries. For these two approach, a training model must be built first, and then the model can be used for prediction. For an ad-hoc query, if its characteristics (e.g., number of joins and the operators used) are very different from queries in the training sample, the prediction made by these two approach will be inaccurate. Unlike these two performance predictor, other progress indicators work for any ad-hoc queries. Our work falls in this category. We address the problems faced by existing fine-granularity query progress indicators and provide solutions for improving progress estimation accuracy.

The third category consists of techniques that do not aim at developing progress indicators directly, but can be used by progress indicators instead. The cardinality estimation techniques [32, 59] can provide the basic information for many progress indicators. Both the MSRPI and the WiscPI take advantage of runtime statistical information to refine initial cardinality estimation by optimizers [36, 53]. The cost estimates [34] are exploited by our proposed query progress indicator. Since a challenging problem for progress estimation is to estimate the total work/cost of a query, any method that can be used to increase the cardinality/cost estimation accuracy, either before the query starts or during its execution, fall into the third category.

2.9 Summary

Previous progress indicators have made a uniform speed assumption for progress estimation. We present a deeper insight into a query's execution, which directly affects prediction accuracy. We also provides the first performance evaluation for the MSRPI and the WiscPI in the same hardware and software framework, and point out where they do and do not give good estimates. To address their limitations, we introduce a new cost-based progress indicator GSLPI, which utilizes wall-clock pipeline cost to produce higher quality progress estimates. The effectiveness of our techniques are verified with extensive experiments.

This work lays down a foundation for further development of progress indicators. One interesting direction would be to extend our progress indicator for parallel database systems where additional challenges exist (e.g., data skew, new operators, etc.). Another promising direction would be to provide a cost-based progress indicator for multiple concurrently running queries and utilize the information provided by progress indicators for better workload and resource management.

Chapter 3

Resource Bricolage for Parallel Database Systems

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

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

scheme. We verify the effectiveness of our technique with an extensive experimental study on a commercial database system.

3.1 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 [26, 44, 62, 63].

3.1.1 Motivation

Performance differences among machines (either physical or virtual) in the same cluster pose new challenges for parallel database systems. By default, parallel systems ignore differences among machines and try to assign the same amount of data to each. If these

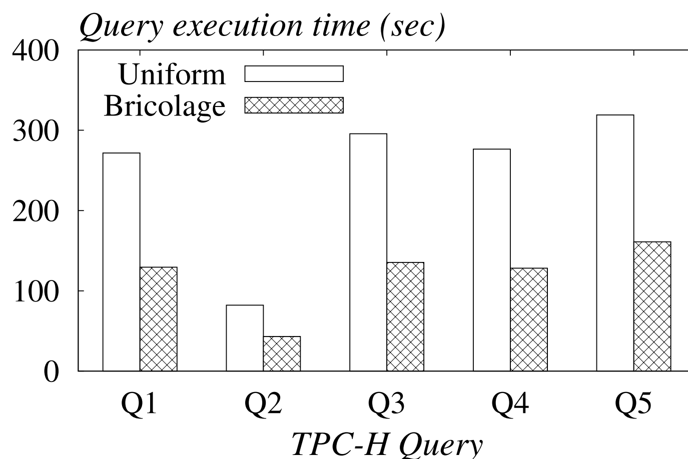


Figure 3.1: Query execution times with different data partitioning strategies.

machines have different disk, CPU, memory, and network resources, they will take varying amounts of time to process the same amount of data. Unfortunately, the execution time of a query in a parallel database system is determined by its slowest machine. At worst, a slow machine can substantially degrade the performance of the query.

On the other hand, a fast machine in such a system will be under-utilized, finishing its work early, sitting idle, and waiting for the slower machines to finish. This suggests that we can reduce execution time by allocating more data to more powerful machines and less data to the overloaded slow machines, in order to reduce the execution times of the slow ones. In Figure 3.1, we compare the execution times of the first 5 TPC-H queries running on a heterogeneous cluster with two different data partitioning strategies. One strategy partitions the data uniformly across all the machines, while the other partitions the data using our proposed technique, which we present in Section 3.4. The detailed cluster setup is described in Section 3.6. As can be seen from the graph, we can significantly reduce total query execution time by carefully partitioning the data.

Our task is complicated by the fact that whether a machine should be considered powerful or not depends on the workload. For example, a machine with powerful CPUs

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

Automated partitioning design for parallel databases is a fairly well-researched problem [17, 52, 58, 60]. The proposed approaches improve system performance by selecting the most suitable partitioning keys for base tables or minimizing the number of distributed transactions for OLTP workloads. Somewhat surprisingly, despite the apparent importance of this problem, no existing approach aims directly at minimizing decision support execution time for heterogeneous clusters. We will provide detailed explanations in Section 3.7.

3.1.2 Our Contributions

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

In the context of our problem, a set of heterogeneous machines are the available re-

sources, and we want to use them to process a database workload as fast as possible. Thus, to implement resource bricolage, we must know the performance characteristics of the machines that execute database queries, and we must also know which machines to use and how to partition data across them to minimize workload execution time. To do this, we quantify differences among machines by using the query optimizer and a set of profiling queries that estimate the machines' performance parameters. We then formalize the problem of minimizing workload execution time and view it as an optimization problem that takes the performance parameters as input. We solve the problem using a standard linear program solver to obtain a recommended data partitioning scheme. In Section 3.4.4, we also discuss alternatives for handling nonlinear situations. We implemented our techniques and tested them in Microsoft SQL Server Parallel Data Warehouse [10], and our experimental results show the effectiveness of our proposed solution.

The rest of the chapter is organized as follows. Section 3.2 formalizes the resource bricolage problem. Section 3.3 describes our way of characterizing the performance of a machine. Section 3.4 presents our approach for finding an effective data partitioning scheme. Section 3.5 summarizes the challenges that are specific to our database system. Section 3.6 experimentally confirms the effectiveness of our proposed solution. Section 3.7 briefly reviews the related work. Finally, Section 3.8 concludes the chapter with directions for future work.

3.2 The Problem

3.2.1 Formalization

To enable parallelism in a parallel database system, tables are typically horizontally partitioned across machines. The tuples of a table are assigned to a machine either by applying

a partitioning function, such as a hash or a range partitioning function, or in a round-robin fashion. A partitioning function maps the tuples of a table to machines based on the values of specified column(s), which is (are) called the partitioning key of the table. As a result, a partitioning function determines the number of tuples that will be mapped to each machine.

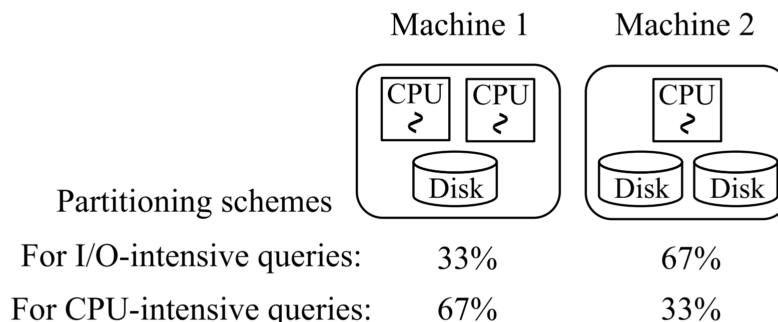


Figure 3.2: Different data partitioning schemes.

A uniform partitioning function may result in poor performance. Let us consider a simple example where we have two machines in a cluster as shown in Figure 3.2. Let the CPUs of the first machine be twice as fast as that of the second machine, and let the disks of the first machine be 50% slower than that of the second machine. We want to find the best data partitioning scheme to allocate the data to these two machines. Suppose that we have only one query in our workload, and it is I/O intensive. This query scans a table and counts the number of tuples in the table. The query completes when both machines finish their processing. To minimize the total execution time, it is easy for us to come up with the best partitioning scheme, which assigns 33% of the data to the first machine and 67% of the data to the second machine. In this case, both machines will have similar response times. Assume now that we add a CPU-intensive query to the workload. It scans and sorts the tuples in the table. Determining the best partitioning scheme in this case becomes a non-trivial task. Intuitively, if the CPU-intensive query takes longer to execute than the

I/O-intensive query, we should assign more data to the first machine to take advantage of its more powerful CPUs, and vice versa.

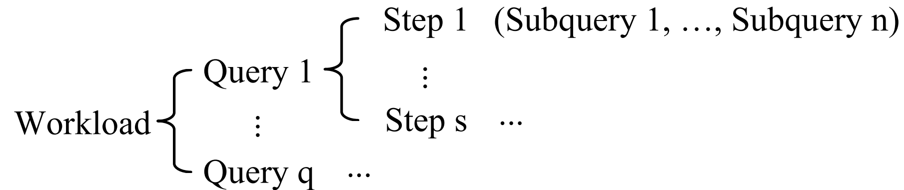


Figure 3.3: A query workload.

In general, we may have a set of heterogeneous machines with different disk, CPU, and network performance, and they may have different amounts of memory. At the same time, we have a workload with a set of SQL queries as shown in Figure 3.3. A query can be further decomposed into a number of *steps* with different resource requirements. For each step, there will be a set of identical subqueries executing concurrently on different machines to exploit data parallelism. A step will not start until all steps upon which it depends on, if any, have finished. Thus, the running time of a step is determined by the longest-running subquery. The query result of a step will be repartitioned to be utilized by later steps, if needed.

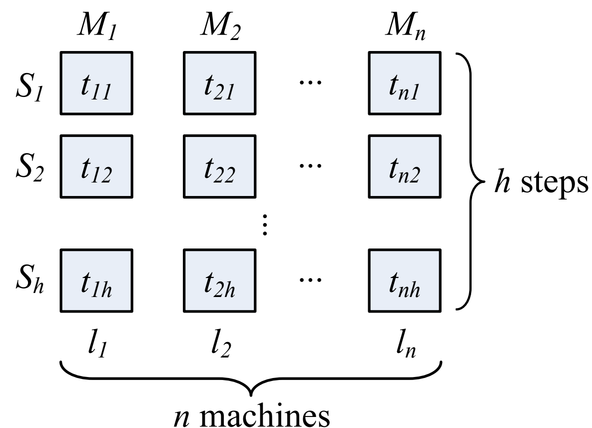


Figure 3.4: Problem setting.

We visually depict our problem setting in Figure 3.4. Let M_1, M_2, ..., M_n be a set of machines in the cluster, and let W be a workload consisting of multiple queries. Each query consists of a certain number of steps, and we concatenate all the steps in all of the queries to get a total of h steps: S_1, S_2, ..., S_h. Assume that t_{ij} would be the execution time for step S_j running on machine M_i if all the data were assigned to M_i. Each column in the graph corresponds to a machine, and each row represents the set of subqueries running on the machines for a particular step. In addition, we assume that a machine M_i also has a storage limit l_i, which represents the maximum percentage of the entire data set that it can hold. The goal of resource bricolage is to find the best way to partition data across machines in order to minimize the total execution time of the entire workload.

3.2.2 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 existing approach for this problem, we propose Delete as a

simple heuristic that attempts to handle resource heterogeneity. It deletes some slow machines before it partitions the data uniformly to the remaining ones. It tries to delete the slowest set of machines first, and then the second slowest next. This process is repeated until no further improvement can be made. Optimal is the ideal data partitioning strategy that we want to pursue. It distributes data to machines in a way that can minimize the overall workload execution time. The corresponding query execution times for these strategies are denoted as t_u, t_{del}, and t_{opt}, respectively. According to the definitions, we have $t_u \geq t_{del} \geq t_{opt}$.

We start with a simple case with n machines in total, where a fraction p of them are fast and $(1-p)$ are slow. Our workload contains just one single-step query. For simplicity, we assume that one fast machine can process all data in 1 unit of time (e.g., 1 hour, 1 day, etc.), and the slow machines need r units of time ($r \geq 1$). We also assume that, for each machine, the processing time of a step changes linearly with the amount of data. The value r can also be considered to be the ratio between execution times of a slow machine and a fast machine. We omit the underlying reasons that lead to the performance differences (e.g., due to a slow disk, CPU, or network connection), since they are not important for our discussion here. It is easy to see that $t_u = \frac{1}{n}r$, $t_{del} = \min\{\frac{1}{n}r, \frac{1}{np}\}$. In this limited specialized case that we are considering, calculating t_{opt} is easy and can be conducted in the following way. We denote the fractions of data we allocate to a fast machine as p_1 and to a slow machine as p_2, respectively. The optimal strategy assigns data to machines in such a way that the processing times are identical. This can be represented as $p_1 = rp_2$. Since the sum of all p_1s and p_2s is 1, we can derive $t_{opt} = \frac{r}{n(rp+1-p)}$.

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

trying to come up with Optimal.

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

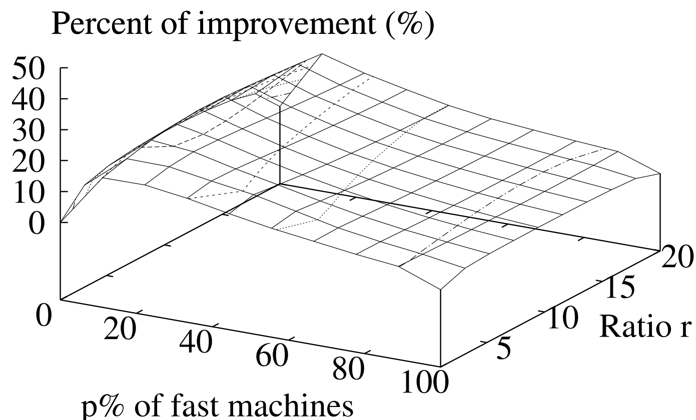


Figure 3.5: Potential for improvement.

From Delete to Optimal. In this case, the improvement we can make is not so obvious. In Figure 3.5, we plot the percentage of time that can be reduced from t_{del} to t_{opt}. We vary p from 0 to 100% and r from 0 to 20. As we can see from the graph, when r is fixed, the percentage of reduction increases at first and then decreases as p gets bigger. Similarly, when p is fixed, the percentage of reduction also increases at first and then decreases as we vary r from 0 to 20. More precisely, when $r \leq \frac{1}{p}$, $t_{del} = \frac{1}{n}r$. The percentage

of reduction can be calculated as $100(1 - t_{opt}/t_{del}) = 100(1 - \frac{1}{rp+1-p})$. Since $rp \leq 1$, we have $rp + 1 - p < 2$. As a result, the reduction $100(1 - \frac{1}{rp+1-p})$ is less than 50%. When $r > \frac{1}{p}$, we have $t_{del} = \frac{1}{np}$, and the reduction is $100(1 - \frac{1}{1+\frac{1}{rp}-\frac{1}{r}})$. Since $rp > 1$, the denominator is no larger than 2. Therefore, the percent of reduction is also less than 50%.

Now, let us consider a more complicated example with n machines and $n/2+1$ steps. In this example, we will show that in the worst case, the performance gap between Delete and Optimal can be arbitrarily large. The detailed t_{ij} values are indicated in Figure 3.6, where a is large constant and ε is a very small positive number. If we use each machine individually to process the data, the workload execution time for a machine in the first half on the left is $a + (\frac{n}{2} + 1)\varepsilon$. This is longer than the workload execution time $a + (\frac{n}{2} - 1)\varepsilon$ for a machine in the second half. When we look at these machines individually, the first $n/2$ of them are considered to be relatively slow.

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

Figure 3.6: A worst-case example.

Given these machines, Delete works as follows. First, it calculates the execution time of the workload when data is partitioned uniformly across all machines. The runtime for the first step S_1 is $\frac{1}{n}(a + \varepsilon)$. The runtime for a later step S_j ($j \geq 2$) is $\frac{1}{n}(a - \varepsilon)$, which is the processing time of machine $M_{n/2+j-1}$. In total, we have $n/2$ number of such steps. As a result, the execution time of all steps is $\frac{1}{n}(a + \varepsilon) + \frac{1}{2}(a - \varepsilon)$. Then Delete tries to reduce the execution time by deleting slow machines, thus it will try to delete $\{M_1$, M_2,

..., $M_{n/2}\}$ first. We can prove that the best choice for Delete is to use all machines. On the other hand, the optimal strategy is to use just the "slow" machines and assign $\frac{2}{n}$ of the data to each of them, and we have $t_{opt} = \frac{2}{n}(a + \varepsilon)$. Although Delete uses more machines than Optimal, it is easy to get that $\frac{t_{del}}{t_{opt}} \approx \frac{n}{4}$.

3.2.3 Challenges

Although the worst-case situation may not happen very often in the real world, our main point here is that when there are many different machines in a cluster and we have queries with various resource demands, the heuristic (Delete) that works well for simple cases may generate results far from optimal. In addition, the heuristic works by deleting the set of obviously slow machines. However, simple cases where we can divide machines in the same cluster into a fast group and a slow group may not happen very often. According to Moore's law, computers' capabilities double approximately every two years. If cluster administrators perform hardware upgrades every one or two years, it is reasonable to assume that we may see 2x, 4x, or maybe 8x differences in machine performance in the same cluster. This assumption is also consistent with what has been observed in a very large Google cluster [61]. Normally, we would not add a machine to a cluster that is significantly different from the others to perform the same tasks. On the other hand, machines that are too slow and out of date will be eventually phased out. For systems running on a public cloud, requesting a set of VM instances of the same type to run an application is the most common situation. As we discussed in Section 3.1, 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 [26].

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. Next, we will discuss how we handle these challenges.

3.3 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 [15, 16, 39, 41, 50]. 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.

3.3.1 Estimating the Cost of a Pipeline

Like previous work on execution time estimation [16, 41], we use the execution plan (refer to Section 2.2 for execution plan and pipeline definitions) for a query to estimate its runtime. In addition to the most commonly used operators in a single-node DBMS, such as Table Scan, Filter, Hash Join, etc., a parallel database system also employs data movement operators, which are used for transferring data between DBMS instances running on different machines. In Figure 3.7, we show an execution plan with a data movement operator as the root. The example plan is divided into two different pipelines.

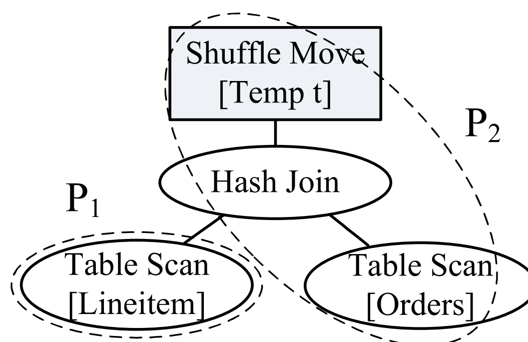


Figure 3.7: An execution plan with two pipelines.

Since pipelines are executed one after another, if we can estimate the execution time for each pipeline, the total runtime of a query is simply the sum of the execution time(s) of its pipeline(s). To estimate a pipeline's execution time, we first predict what is the work of the pipeline and what is the speed to process the work. We then estimate the runtime of a pipeline as the estimated work divided by the processing speed. For each

pipeline, we use the optimizer's cost model to estimate the work (called *cost*) that needs to be done by CPUs, disks, and network, respectively. These costs are estimated based on the available memory size. We utilize the optimizer estimated cost units to define the work for an operator in a pipeline. We follow the idea in Section 2.4 to calculate the cost for a pipeline.

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).

3.3.2 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. In other words, we get the time needed to process the estimated cost on the machine. Then we calculate the I/O speed for this machine as the estimated I/O cost divided by the query execution time.

Measuring CPU speed. To measure the CPU speed, we test a CPU-intensive query: *select T.a from T group by T.a* from a warm buffer cache. For this query, we can also get

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

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

Finally, for a pipeline P, we can 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.

3.4 Resource Bricolage

After we estimate the performance differences among machines for running our workload, we now need to find a better way to utilize the machines to process a given workload as

fast as possible. We model and solve this problem using linear programming, and we deploy special strategies to handle nonlinear scenarios.

3.4.1 Base and Intermediate Data Partitioning

Data partitioning can happen in two different places. One is base table partitioning when loading data into a system, and the other one is intermediate result reshuffling at the end of an intermediate step. For example, consider a subquery of a step that uses the execution plan shown in Figure 3.7. This plan scans two base tables: Lineitem and Orders, which may be partitioned across all machines. The result of this subquery, which can be viewed as a temporary table, is served as input to next steps, if there are any. Thus, the output table may also be redistributed among the machines.

The execution time of a plan running on a given machine is usually determined by the input table sizes. For example, the runtime of the plan in Figure 3.7 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 de-

cide 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 work.

3.4.2 The Linear Programming Model

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

Recall that in our problem setting, we have n machines, and the maximum percentage of the entire data set that machine M_i can hold is l_i. Our workload consists of h steps, and it would take time t_{ij} for machine M_i to process step S_j if all data were assigned to M_i. The actual t_{ij} values are unknown, and we use the technique proposed in Section 3.3 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 proportion 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 intro-

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

duce an additional variable x_j to represent the execution time of step S_j. Thus, the total execution time of the workload can also be represented as $\sum_{j=1}^{h} x_j$. The linear program that minimizes the total execution time of the workload can be formulated below.

For step S_j, since the execution time x_j is the longest execution time of all machines, we must have $p_i t_{ij} \le x_j$ for machine M_i. We also know that the percentage of data that can be allocated to M_i must be at least 0 and at most l_i. The sum of all p_is is 1, since all data must be processed. We can solve this linear programming model using standard linear optimization techniques to derive the values for p_is ($0 \le i \le n$) and x_js ($0 \le j \le h$), where the set of p_i values represents a data partitioning scheme that minimizes $\sum_{j=1}^{h} x_j$. Note that we may use only a subset of the machines, since we do not need to run queries on a machine with 0% of the data. Thus, the data partitioning scheme suggests a way to select the most suitable set of machines and a way to utilize them to process the database workload efficiently.

3.4.3 Allowing Multiple Partitioning Functions

When different partitioning functions are allowed to be used by different tables, we are given more flexibility for making improvements. Thus, we want to apply different partitioning functions whenever possible. In order to do this, we need to identify sets of tables

that must be partitioned in the same way to produce join-compatible distributions, and we apply different partition functions to tables in different sets.

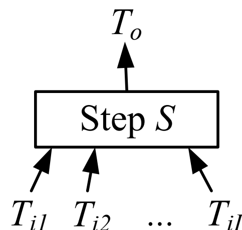


Figure 3.8: The input and output tables for a step.

For step S in workload W, let $\{T_{i1}, T_{i2}, ..., T_{iI}\}$ be the set of its input tables and T_o be its output table as we show in Figure 3.8. An input table to S could be a base table or an output table of another step, and all input tables will be joined together in step S. In order to perform joins, tuples in these tables must be mapped to machines using the same partitioning function, otherwise tuples that can be joined together may go to different machines[1].

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

Given a database, we assume that the partitioning keys for base tables and whether two base tables should be partitioned in a distribution-compatible way or not are designed by a database administrator or an automated algorithm [11, 52, 60]. As a result, we know which base tables should belong to a distribution-compatible group. For intermediate

[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.

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.

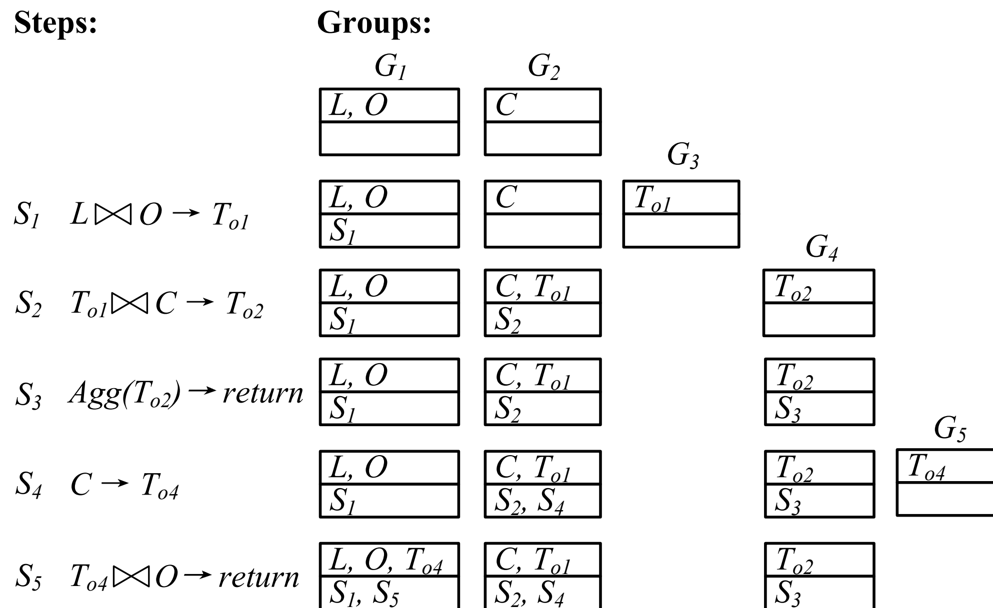


Figure 3.9: Example of distribution-compatible group generation.

We go through a small example shown in Figure 3.9 to demonstrate how it works. The example has only five steps and three base tables: L, O, and C, where L and O are

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

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

3.4.4 Handling Nonlinear Growth in Time

In our proposed linear programming model, we assume that query execution time changes linearly with the data size. Unfortunately, this assumption does not always hold true for database queries. However, as we will see later in our experiments, the assumption is valid in many cases, and even when it does not strictly hold, it is a reasonable heuristic that yields good performance.

This assumption is valid for the network cost of a query, where the transmission time increases in proportion to data size. It is also true for the CPU and I/O costs of many

database operators, such as Table/Index Scan, Filter, and Compute Scalar. These operators take a large proportion of query execution times for analytical workloads.

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

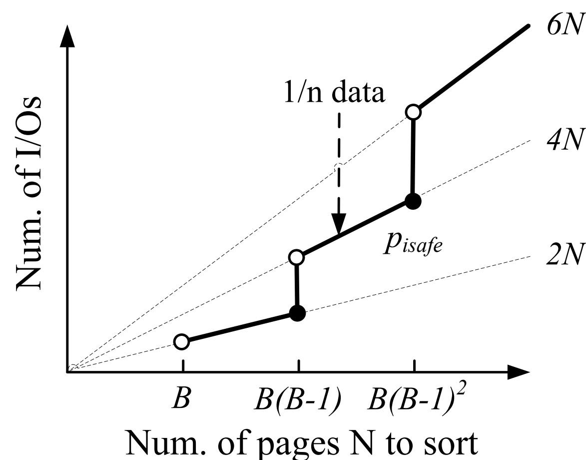


Figure 3.10: I/O cost for Sort.

If the plan is the same as the estimated plan and the operator costs increase linearly with the data size, everything will work as is. However, since the input table sizes could be different from our assumption, the plan may change, and some multi-phase operators may need more or fewer passes to perform their tasks. We use the I/O cost for Sort as our running example, and the I/O cost for Hash Join is similar. To sort a table with N

pages using B buffer pages, the number of passes for merging is $\lceil \log_{B-1} \lceil N/B \rceil \rceil$. In each pass, N pages of data will be written to disk and then brought back to memory. The number of I/Os for Sort[2] can be calculated as $2N \lceil \log_{B-1} \lceil N/B \rceil \rceil$, and we plot this nonlinear function in Figure 3.10. 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,

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

we can force the problem to stay in the region, where query execution times grow linearly with data size. For the example shown in Figure 3.10, 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. We leave this as an interesting direction for future work.

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

3.5 System Specific Challenges

In previous sections, we presented our approach to run database queries efficiently with heterogeneous resources. Ideally, for a given query, we hope that a database system can provide the execution plan for it, together with the estimated costs for different resources. Furthermore, to correctly handle the nonlinearity in time, the estimated costs should be accurate for different phases of a multi-phase operator. However, every database system

is different. When we implemented the technique in Microsoft SQL Server PDW, we encountered some additional challenges that are specific to our system. We summarize these challenges below.

PDW is a shared-nothing database appliance. It consists of a single control node that manages one or more compute nodes. Figure 3.11 is a graphical depiction of a PDW appliance with two compute nodes. The **control node** provides the external interface to the appliance, and there is a DBMS running on it for storing "shell databases". A shell database is a database that defines all metadata and statistics about tables, but does not contain any user data. All user data is stored as tables permanently in the DBMSs running on the compute nodes. These tables could be hash-partitioned or replicated. Each node has a Data Movement Service (DMS) component running, which is responsible for transferring data between nodes.

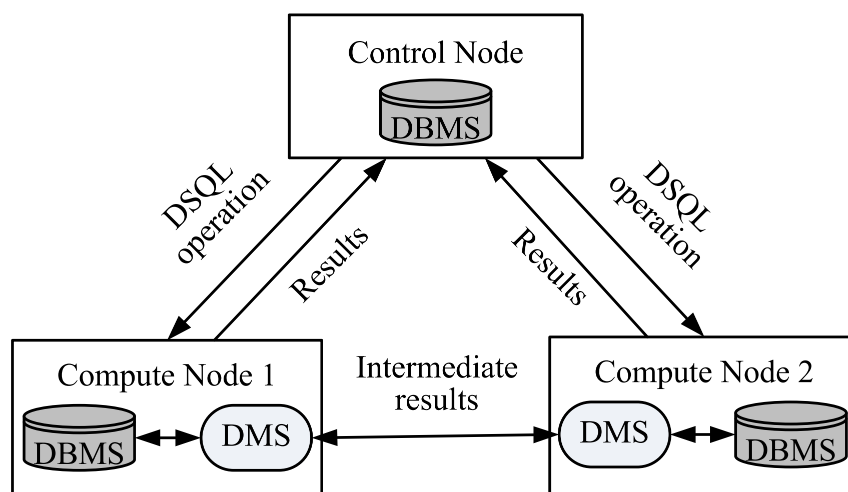


Figure 3.11: Microsoft SQL Server PDW

The first problem we encountered is about plan generation. For a given query, the PDW optimizer does not return detailed execution plans for steps before execution, instead it returns a schedule with only the SQL text strings (the subqueries) for each step.

Actual plans for a step are generated on the fly, after all intermediate results are ready and right before the step is about to start. With more accurate statistics, delayed plan generation can choose more cost efficient plans. To generate execution plans for steps before execution, we create empty tables on the compute nodes. Similar to the ideas of shell databases, we do not load any user data into these tables. For the purpose of plan selection, we modify the metadata to store the estimated statistics, such as number of tuples, average tuple sizes, and histograms for them.

Another problem we found is that the estimated costs provided by the optimizer are not complete. For example, the costs for a DMS operator are missing. A DMS operator is not a traditional operator in SQL Server, and its costs are not provided by the SQL Server optimizer. Since it competes for resources in the system, we need to add its costs to the estimates. The SQL Server cost model also ignores some CPU costs for aggregation operators, such as the CPU costs for sum and avg. Since any plan for the query needs to do the same computations and there may be no other alternatives for doing these, ignoring the costs may not change the ranking of candidate plans. The optimizer may still pick the optimal plan for the query. However, for execution time estimation, we need to add these missing costs, and they must be calculated in a way that is consistent with the way that is used to calculate the original cost estimates. Thus, we mimic the SQL Server optimizer cost model to do this. The I/O cost of a DMS operator is easy to get. It can be calculated as the number of estimated I/Os times the cost per I/O operation. The second value is taken from the optimizer cost model. For the network cost, it is up to us to define the network cost unit for DMS operators. Since the original SQL Server operators do not transfer any data between nodes, there are no network costs for them. We simply take the number of bytes to send as the network cost for a DMS operator. To estimate the CPU cost of a DMS operator, we first run a CPU-intensive query that transfers a large set of data to learn a parameter. For the pipeline P in the query that contains

the DMS operator, we assume that its execution time $t(P) = (Original_CPU_cost + Missing_CPU_cost)/Speed_{cpu}$. In this formula, we know $t(P)$ (by running this query), $Speed_{cpu}$ (by running the CPU test query), and the original estimated CPU cost. Thus, we can calculate how much CPU cost is missing for this pipeline that should be added. This CPU cost for the DMS operator is required for computing a hash value for a tuple, finding its destination, and packing it. Based on the number of tuples sent, we can derive a new parameter to represent the average CPU cost for a DMS operator to transfer a tuple. Then we use this computed parameter to estimate CPU costs for other DMS operators. We use the same idea to estimate another parameter to represent the average CPU cost to perform a scalar operation. A small trick to get a CPU-intensive pipeline is to use as few CPUs as possible, and at the same time, configure the system for the maximum I/O and network speeds. To make sure that the pipeline is really CPU intensive, we can run the query again with more CPUs. If we can shorten the query runtime by adding CPUs, we know that the query is CPU bound in the original setting.

The last problem we encountered is about cost estimation for multi-phase operators. SQL Server cannot provide accurate cost estimates for these operators when memory is extremely sparse. Take Hash Join as an example. When the smaller relation S for the join contains $|S|$ pages and the memory has less than $\sqrt{|S|}$ pages, the optimizer estimated I/O cost for the join is much less than what we expect. Although the costs are not accurate, the operators are allowed to adapt online to available memory by changing the number of phases used. If a specific operator (e.g., a hash join) is typically the best choice when memory size is small, ignoring the actual costs may be fine for an optimizer. In our work, without accurate costs, we cannot use feedback to improve our partition recommendations when operators change the number of phases needed. We can either issue a warning saying that we do not have much confidence in our estimates or set guarding points to prevent us from entering the uncertain regions. However, in most cases, we believe that

the available memory in a system should be larger than the square root of the smaller input table size to Hash Join (or the input table size to Sort).

3.6 Experimental Evaluation

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

3.6.1 Experimental Setup

We implemented and tested our techniques in Microsoft SQL Server Parallel Data Warehouse. Our cluster consisted of 9 physical machines, which were connected by a 1Gbit HP Procurve Ethernet switch. Each machine had two 2.33GHz Intel E5410 quad-core processors, 16GB of main memory, and eight SAS 10K RPM 147GB disks. On top of each physical machine, we created a virtual machine (VM) to run our database system. One VM served as a control node for our system, while the remaining eight were compute nodes. We artificially introduced heterogeneity by allowing VMs to use varying numbers of processors and disks, limiting the amount of main memory, and by "throttling" the network connection.

The parallel database system we ran consists of single-node DBMSs connected by a distribution layer. As a result, 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

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

Table 3.1: Partition keys for the TPC-H tables.

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

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

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

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

Table 3.2: Overall performance.

the parallel distribution layer. We used a TPC-H 200GB database for our experiments. Each table was either hash partitioned or replicated across all compute nodes. Table 3.1 summarizes the partition keys used for the TPC-H tables. Replicated tables were stored at every compute node on a single disk.

3.6.2 Overall Performance

To test the performance of different data partitioning approaches, we used a workload of 22 TPC-H queries. By default, each VM used 4 disks, 8 CPUs, 1Gb/s network bandwidth, and 8GB memory. In the first set of experiments, we created 6 different heterogeneous environments as summarized below to run the queries. In these cases, we vary only the number of disks, CPUs, and the network bandwidth for the VMs. We will study the impact of heterogeneous memory in a separate subsection later.

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

2. ***Network-intensive configuration***: similarly, to make more queries network bound, we reduce the 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 in the cluster. 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 the 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 chapter (we refer to it as Bricolage). We use Uniform and Delete as the competitors, since to the best of our knowledge, there are no previously proposed solutions in the literature. The improvement in execution time due to our bricolage techniques depends on differences among machines. For each cluster configuration, we first measure the processing speeds for each machine using the profiling queries and the network test program described in Section 3.3. 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. As a result, we can derive the estimated query execution time for each query and the estimated t_{ij} values for each machine. 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 [18]. The model returns a recommended data partitioning scheme together with the targeted workload execution time. In Table 3.2(a), we illustrate the predicted workload execution time for different approaches running with different cluster configuration. 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 3.2(b), we list the numbers we observe after running the workload. As we can see from the table, although in some cases, our absolute time estimates are not very precise, the percentage improvement we achieve is close to our predictions. As a result, we can conclude that our model is reliable for making recommendations.

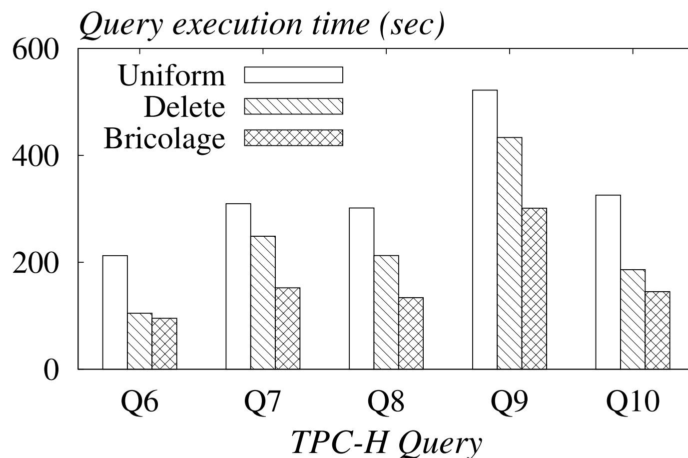


Figure 3.12: Query execution time comparison.

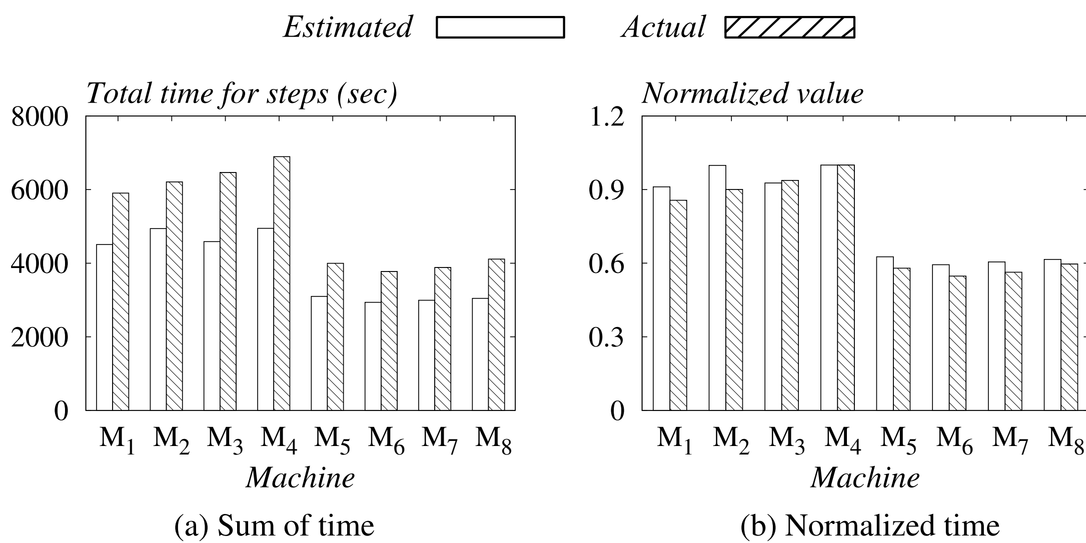
In Figure 3.1, we show the execution times of the first 5 TPC-H queries (Q_1 to Q_5) running with the I/O-intensive (4) configuration. In Figure 3.12, we show the results for

the next 5 TPC-H queries (Q_6 to Q_{10}) along with the results for Delete. Compared to Uniform, Delete reduces query execution times by removing the slowest machines (the bottleneck) with just one disk. For Q_6, Delete and Bricolage have similar performance, since this query moves a lot of data to the control node, which is the bottleneck when data is partitioned using these two strategies. For other queries, Bricolage can further reduce query execution times by fully utilizing all the computing resources.

3.6.3 Execution Time Estimation

In our work, we quantify differences among machines using data processing times (the $t_{ij}s$). Thus, we want to see whether our estimated times truly indicate the performance differences. For each machine in the cluster, we sum up its estimated and actual execution times for all steps. In Figure 3.13(a), we plot the results for the CPU-intensive configuration. In this case, the estimated workload execution time is 5346 seconds, which is shorter than the actual execution time of 7371 seconds. From the graph, we can see that the estimated times for all machines are consistently shorter than the corresponding actual execution times. If we pick the machine with the longest actual processing time (M_4 in the graph) and use the actual (estimated) time for it to normalize the actual (estimated) times for other machines, we get the normalized performance for all machines as shown in Figure 3.13(b). Ideally, we hope that for each machine its normalized estimated value is the same as the actual value. Although our estimates are not perfect, the errors we make when predicting relative performance differences are much smaller than when predicting absolute performance.

From Figure 3.13(b), 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



(a) Sum of time (b) Normalized time

Figure 3.13: Performance predictions for machines.

much data to the overestimated ones, which leads to performance degradation. As a result, the actual improvement we obtained is usually smaller than the predicted improvement.

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

We also found that I/O processing speeds are easier to estimate than CPU and network speeds, and the time predictions for the I/O-intensive configurations are very accurate. In Figure 3.14, we compare the estimated and the actual query execution times for all 22 queries in our workload when they are running with the Mix-2 configuration. We can provide accurate estimates for most queries. In our experiments, the time estimates for

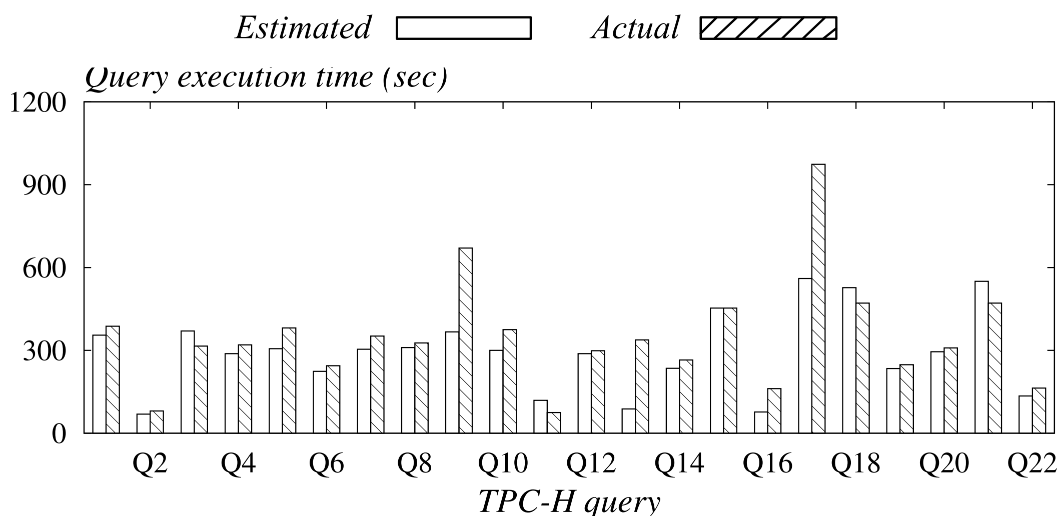


Figure 3.14: Estimated vs. actual query runtimes.

the queries running with the I/O-intensive (2) and (4) configurations are more accurate than the time estimates shown in Figure 3.14, while the time estimates for the queries running with the Mix-3 configuration are less precise.

3.6.4 Investigating Optimal Improvements

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

We try to derive the best possible improvements by using information obtained from actual runs of the queries to get more accurate t_{ij} estimates. An execution plan consists of multiple pipelines. 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. As a result, we know their actual execution times, and we can replace our estimated values with the actual values. However, for the last pipeline in a plan, 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 in the cluster. 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. As a result, 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.

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

Table 3.3: Estimated time reductions using actual runs.

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 3.3. 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.

3.6.5 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 the data size. The cluster is configured with the I/O-intensive (4) setting. We set the memory size of the last machine to 0.25GB or 0.5GB, and we vary the amount of data assigned to it from 10% to 50%. The memory sizes of the other machines are set to 8GB, respectively, and they evenly share the remaining data. We sum up the time to process all steps for the last machine and plot the results in Figure 3.15(a). In both cases, the total time increases linearly with data size. When memory size is 0.5GB, all memory-consuming operators need no more than one pass, and when the memory size is 0.25GB, some operators need two passes. Since these operators do not change the number of passes required when we vary data size, they stay in regions where processing time grows linearly. Furthermore, as we have shown in Figure 3.10, the line is steeper when more passes are needed. Thus, when memory size is 0.25GB (2 passes are needed), the line should also have a steeper slope. To see this more clearly, we plot the results in Figure 3.15(b) for a subset of the most memory-consuming queries, rather than all queries in our workload.

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

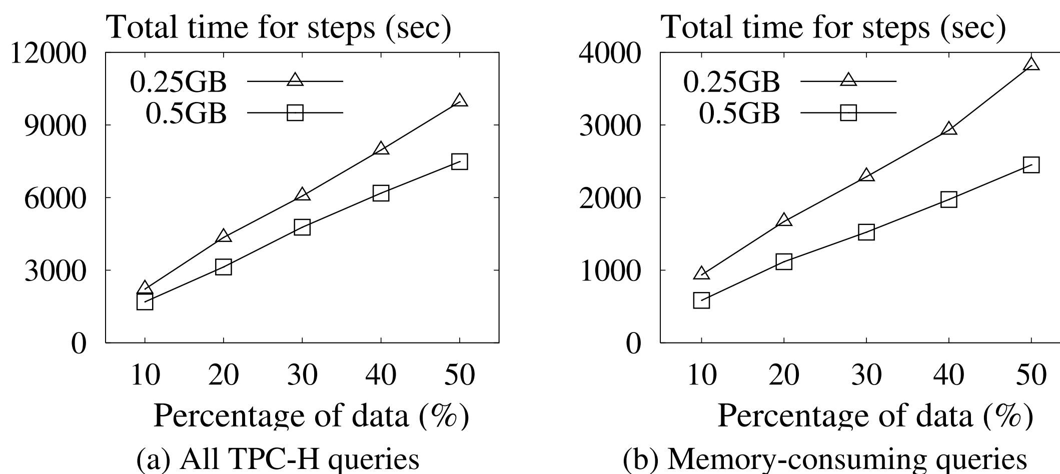


Figure 3.15: Execution time vs. data size.

an example. We set the DBMS memory size to 0.5GB (where no operator needs more than one pass) for the last machine and 8GB for other machines to repeat the experiment. The predicted and actual time reductions for our approach are 53.5% and 46.7%, respectively. The time estimates for the last machine correctly represent its performance differences compared to other machines, and thus less data is assigned to it compared to its original configuration with 8GB memory.

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

Table 3.4: Percentage of time reductions when memory size is 0.25GB for the last machine.

However, when memory is really scarce and more than one pass is required, the I/O cost estimates provided by our system are no longer accurate. Our predicted times are usually smaller than the actual processing times. In the first column of Table 3.4, we show the estimated and actual reductions in time for our default approach without guarding

points (we refer to it as Bricolage-d in the table), when the DBMS memory size is set to 0.25GB for the last machine. This is a really adversarial situation, since the last machine has the most powerful disks to accomplish more I/O work while at the same time, it does not have enough memory to accommodate the data. The actual performance we obtained is much worse than our prediction, since we assign too much data to the last machine.

We have proposed two strategies in Section 3.4.4 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 3.4. 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.

3.6.6 Overhead of Our Solution

Our approach needs to estimate the processing speeds for machines, estimate plans and their execution times, and solve the linear model. Here, we describe the overheads involved. In our experiments, we used 2 minutes each to test the I/O and the CPU speeds

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

3.7 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 [15, 37, 39, 41, 42, 48]. The key idea for this work is to collect runtime statistics from the actual execution of a query to dynamically predict the remaining work or execution 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 [51] is an exception. However, it cannot provide accurate estimates for queries running on database systems [40]. On the other hand, the second category of work focuses on query running time prediction before a query starts [13, 24, 28, 64, 65]. In [65] 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. The other work has explored the use of machine-learning based techniques [13, 24, 28] for the estimation of query runtime. 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. In [52, 60], this is achieved by selecting the most suitable partitioning key for each table 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. After the partitioning keys are selected, data is partitioned uniformly across all machines based on the key values. 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 [17, 58] 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 [22, 67, 68]. Skew handling is in a sense the dual problem of the one that we deal with in this chapter. 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 work is related to various approaches proposed for improving system per-

formance in heterogeneous environments [12, 69]. A suite of optimizations are proposed in [12] to improve MapReduce performance on heterogeneous clusters. Zaharia et al. [69] 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.

3.8 Summary

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

Our work in this chapter lays down a foundation for several directions towards future studies to improve database performance running in the cloud. Previous research has revealed that the supposedly identical instances provided by a public cloud often exhibit measurable performance differences. One interesting problem is to select the set of most cost-efficient instances to minimize the execution time of a workload. In the next chapter, we will study this problem. While the focus of this work has been on static data partitioning strategies, the natural follow-up to this work will be to study how to dynamically repartition the data at runtime, when our initial prediction was not accurate or system conditions change.

Chapter 4

Resource Bricolage with Constraints

In the previous chapter, we presented our solution to improve database performance in heterogeneous environments. In that problem setting, we assumed that we knew all the machines that can be used to execute a workload, and we focused on solving a *data allocation* problem where we tried to assign proper amounts of data to the machines to fully exploit their potential. Our approach can significantly reduce workload execution time for heterogeneous clusters. However, when a new cluster is built or when an old cluster is upgraded, there might be various machines that we can choose to do it. Since machines might have varying resources, different choices may lead to different costs or performance. Thus, we may encounter a *resource selection* problem where we need to decide which machines to use if we only have a limited budget or a performance goal. 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.

In this chapter, we discuss two resource bricolage problems with constraints that occur in practice. We first formally define the problems and prove their hardness. We then

employ mixed integer programming techniques to efficiently search for the optimal solution which minimizes either (i) workload execution time with a budget constraint, or (ii) cost of money with a time constraint. Finally, we analyze various use cases to illustrate when a simple heuristic solution is effective and when a sophisticated solution is needed for these problems. Through synthetic experiments, we show that a solution that combines both data allocation and resource selection can usually yield significant performance improvement over other alternatives that do not.

4.1 Introduction

In many cases, it has become inevitable for us to select computing resources from a diverse range of such resources to build a cluster or to run an application. For example, we may need to select a set of machines to upgrade a cluster, choose a set of virtual machine instances to run an application, or pick a set of resource containers to execute a workload.

Hardware upgrades. When a cluster is first built, we may start with homogeneous hardware. As time passes by and business grows, we may want to upgrade the hardware or add more capacity to the cluster. By then, we may not be able to find the same hardware that was used to build the cluster at the very beginning, or we may be able to buy different kinds of more powerful machines with the same amount of money. Thus, there are a variety of machines that we can choose for cluster upgrades.

Public clouds. With the prosperity of cloud computing, an increasing number of applications are moving into public clouds. Cloud providers typically adopt a pay-as-you-go model in which tenants can allocate and terminate virtual machine instances at any time and pay for the machine hours they use [1, 4, 6, 9]. However, not all virtual machine instances of the same type are created equal [26, 56]. Due to underlying hardware differences and resource contention, the supposedly identical instances provided by a public

cloud often exhibit measurable performance differences [25, 44, 62, 63]. A striking consequence of such a counterintuitive phenomenon is that a customer may end up paying the same amount of money for instances of the same type with varying performance. On the other hand, the variation between virtual machine instances suggests that cloud customers may optimize performance by carefully selecting the most suitable instances for their workloads.

Shared infrastructure. At the same time, for those systems that remain in private clusters, it is desirable to consolidate different workloads into a shared infrastructure to exploit data locality and multiplex physical resources. In this shared environment, it is hard to guarantee that the resources available on different machines will always be the same for a specific application/query that is about to start. For example, for applications running on Apache Hadoop NextGen MapReduce (YARN) [3] framework, they may request different resource containers on different machines, leaving behind a collection of containers with vastly different capacities for new coming applications. As a result, we need to decide which machines and containers to use for new applications.

4.1.1 Motivation

The consequent performance heterogeneity is of concern to cluster users. Due to performance heterogeneity, if we carefully select which resources to use, we may obtain better performance with the same budget or achieve the same performance with a lower cost. For cluster upgrades and resource multiplexing, since machines or containers may have varying capacities, different choices may lead to different performance. For public clouds, there are a number of differences between running applications on them and on traditional clusters, and the most important difference is *variability* [44]. To help customers deploy applications in the cloud, previous researchers either conduct performance

evaluations or design algorithms to seek virtual machine instances with better performance [26, 56, 57, 70, 71, 72]. 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 [56, 57].

However, previous research has been focused on relatively simple workloads, such as workloads with a single bottleneck resource. For example, the work in [71] aims at only latency-sensitive applications where the response time of a service request largely depends on network connectivity between instances. As we have discussed in the previous chapter, our targeted workloads consist of SQL queries running in a parallel database system, which may be decomposed into a number of steps with different resource 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 of all, 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 (refer to Section 3.2.3 for an example when machines do not collaborate well). Finally, we should allocate data using our resource bricolage technique presented in previous chapter for the minimum workload execution time. Unfortunately, the first two aspects sometimes could be contradictory. For example, 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.

4.1.2 Contributions

For the problem we addressed in the previous chapter, we knew all the machines in a cluster, and the goal was to minimize workload execution time. We did this by dealing with the following two challenges: one is to identify the performance characteristics of the machines, and the other is to assign proper amounts of data to them. For the two problems we are going to solve in this chapter, we do not know which machines are going to be used in the cluster. Our goal is to select the most suitable computing resources due to budget constraints or time constraints. More specifically, in addition to the above two challenges, we also need to either (i) select a set of computing resources that minimize the total execution time given a budget, or (ii) select a set of computing resources that minimize the cost given a performance target.

In this chapter, we tackle these challenges introduced by performance variability and additional constraints when running parallel database systems on heterogeneous environments. Like our previous solution, we first quantify differences among machines and formalize the constrained problems as optimization problems. We then formulate the problems as different Mixed-Integer Programs (MIPs). We finally solve the programs using different standard linear program solvers to obtain both the resource selection decision (e.g., which machines to use) and the data allocation decision (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. 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 this chapter is organized as follows. In the next section, we formalize the resource bricolage problems with constraints and prove the NP-hardness of the problems.

Section 4.3 presents our solutions for solving the problems. Section 4.4 evaluates the performance of different approaches. We conclude the chapter in Section 4.5.

4.2 The Constrained Bricolage Problems

4.2.1 Problem Definitions

The problem setting in this chapter is similar to the one in previous chapter (see Figure 3.4 for more details), and is summarized as below.

Machines: We have a set of n heterogeneous machines denoted as M_1, M_2, ..., 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, ..., 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: Our problems 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 problem in previous chapter, the n machines are the machines in a cluster, and there are no constraints on how we use them. The n machines here are considered as candidates. Due to the additional constraints, we can 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. We 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 through out the chapter. 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, our goal is the 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, 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.*

4.2.2 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 Figure 3.6, 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 ($M_{\frac{n}{2}+1}$, $M_{\frac{n}{2}+2}$, ..., M_n) 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 Figure 3.6, 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 $U = \{e_1, e_2, ..., e_h\}$, a set S of n sets u_1, u_2, ..., 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}, ..., 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) [35]? 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}, ..., 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.

4.3 Solving the Problems

Like the problem in Chapter 3, we first need to estimate all t_{ij} values. We use the approach presented in Section 3.3 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 solution.

4.3.1 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 Section 3.4.2, 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.

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

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 Section 3.4.2. 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 $\sum_{i=1}^{n} m_i \leq b$. We also want to enforce that we will not allocate any data to a machine that is not selected. In other words, when $m_i = 0$ $(1 \leq i \leq n)$, we must also have $p_i = 0$; and when $m_i = 1$, p_i can be greater than 0. We use the constraint function

$m_i \geq p_i$ to enforce this requirement. Since m_i can only take two values, when $m_i = 0$, $m_i \geq p_i$ implies that p_i must be 0 as well. When $m_i = 1$, p_i can be any non-negative 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.

4.3.2 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. We formulate this problem as below.

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

In the objective function, we minimize the total number of selected machines. The first five constraint functions are the same as those in Section 4.3.1. 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 $\sum_{j=1}^{h} x_j \leq T$. 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 [15]. As a result, when meeting a performance goal is critical, we use $\sum_{j=1}^{h} x_j \leq T * \alpha$ instead of $\sum_{j=1}^{h} x_j \leq T$ as a constraint function, where $0 < \alpha \leq 1$. 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.

4.4 Simulation Experiments

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 problems. We first give a summary of the problem 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.

4.4.1 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, ..., res_x, where x is the number of different resource types. For the other type of machine, its amounts of resources must be $res_1 * \gamma$, $res_2 * \gamma$, ..., $res_x * \gamma$, 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 Section 4.1.1, 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 Section 4.2.2, 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 Chapter 3 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 Section 4.3, and it provides the optimal solution by taking into account all the three aspects.

Approaches	Case (1)	Case (2)	Case (3)	Case (4)
Blind	$\surd$	$\times$	$\times$	$\times$
Greedy_U	$\surd$	$\surd$	$\times$	$\times$
Greedy_B	$\surd$	$\surd$	$\surd$	$\times$
Optimal	$\surd$	$\surd$	$\surd$	$\surd$

Table 4.1: Overall performance of different approaches.

For each case listed above, we compare the performance of these four approaches. In Table 4.1, we summarize the overall performance of these approaches. We use $\surd$ to

indicate that the corresponding approach can find the optimal solution for a given case, and we use $\times$ 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, 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 can not 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 Section 4.2.2.

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).

4.4.2 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 hour, 1 day, etc.), while the slow machines need r units of time ($r \geq 1$) to process the same data. Suppose that our budget 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 Figure 4.1, 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, 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.

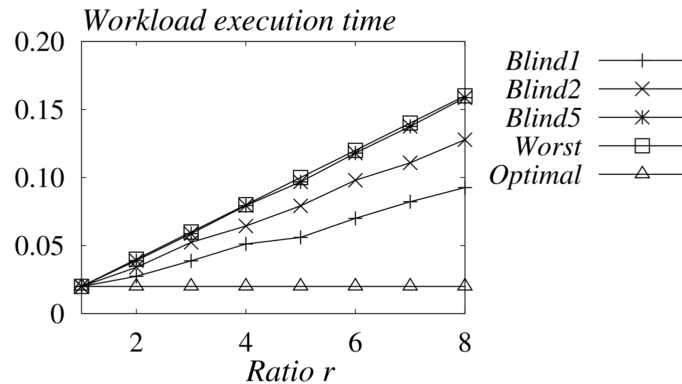


Figure 4.1: Comparison of workload execution times for Case (2).

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.

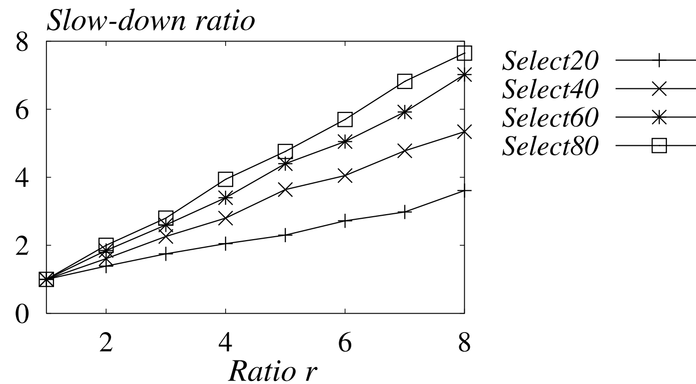


Figure 4.2: Slow-down ratio for Case (2).

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 Figure 4.2. 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.

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 4.3 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.

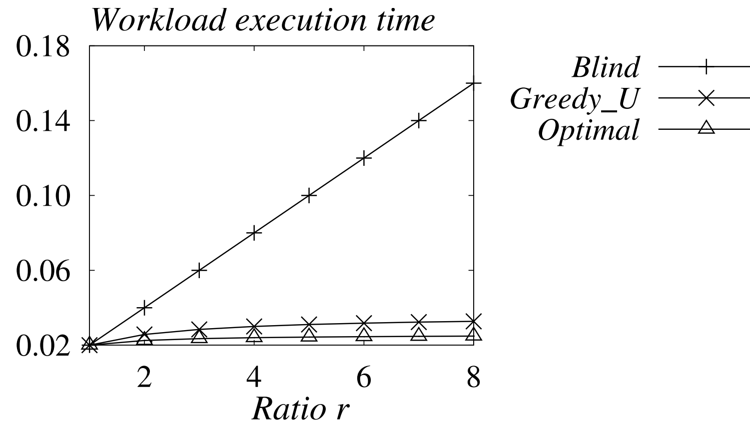


Figure 4.3: Comparison of workload execution times for Case (3).

Recall that a slow machine takes $0.02r$ unit of time to process the data, the processing time of a step is 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 Chapter 3 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 Figure 4.4, we show 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

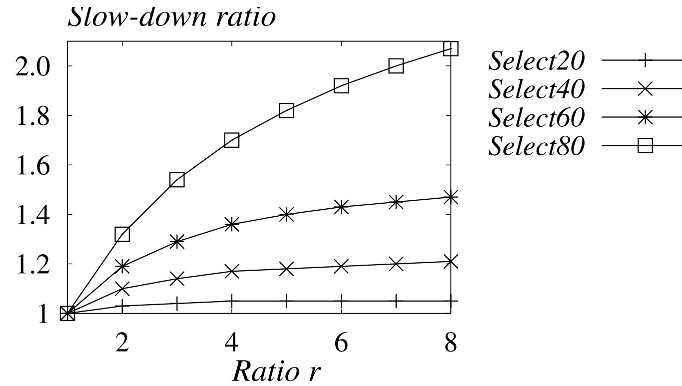


Figure 4.4: Slow-down ratio for Case (3).

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 Figure 4.5.

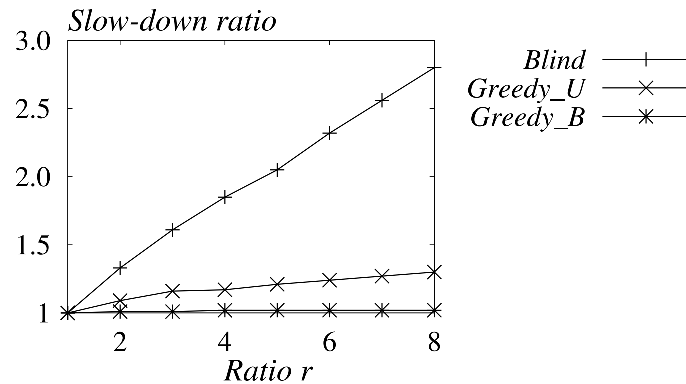


Figure 4.5: Slow-down ratio for Case (4).

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. 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.

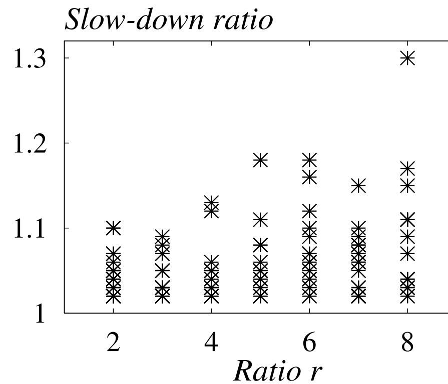


Figure 4.6: Slow-down ratio of Greedy_B for Case (4).

In Figure 4.6, 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. There is no guarantee of its performance.

4.4.3 Experiments for Minimum Cost Resource Selection

The machines and workloads we employ here for the experiments are the same as those we use in Section 4.4.2 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 Figure 4.7, 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.

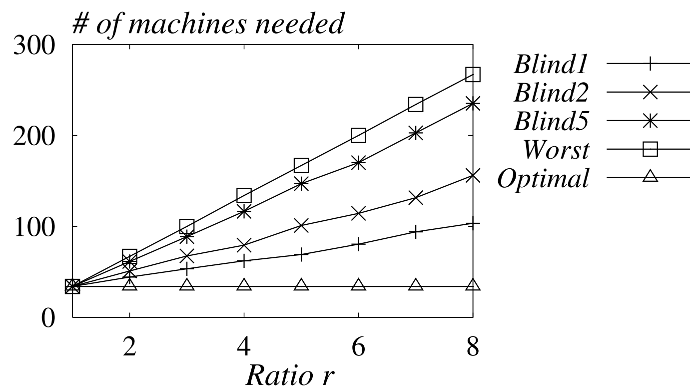


Figure 4.7: Comparison of number of machines needed for Case (2).

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 Figure 4.8. 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.

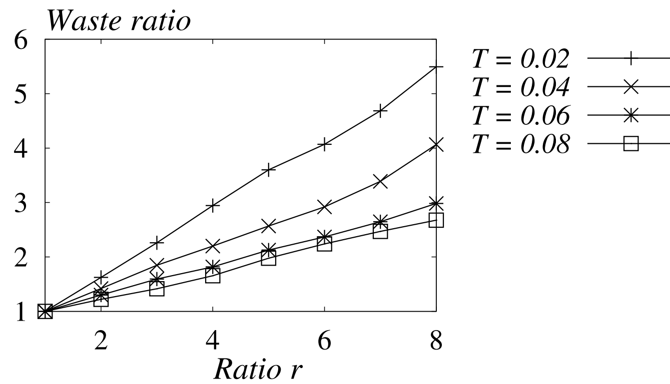


Figure 4.8: Waste ratio for Case (2).

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 Figure 4.9. 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 can not achieve the performance goal even if it uses all the 100 machines. When r is greater than two, it is impossible for Blind to achieve this goal either. For these cases, we plot their data points with the value of 101 in Figure 4.9 to indicate the target performance is unachievable for the corresponding approaches.

More results for Greedy_U are shown in Figure 4.10, where we vary T from 0.02 to 0.08. We omit the results when the performance goal can not be achieved from the graph. As we mentioned before, Greedy_U can not 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 Section 4.4.2, the achievable performance goal varies with

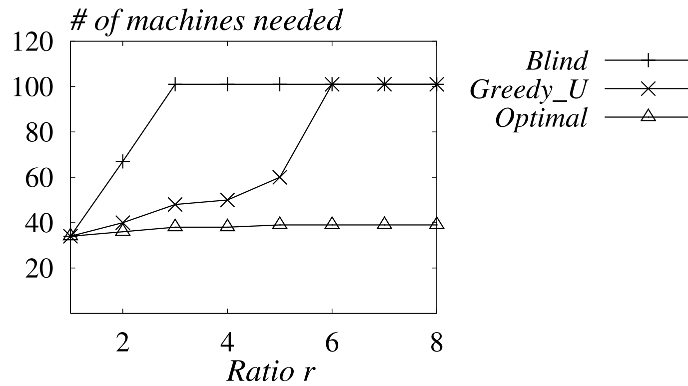


Figure 4.9: Comparison of number of machines needed for Case (3).

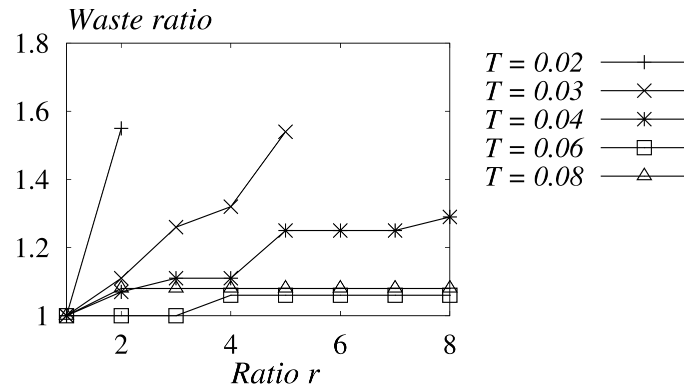


Figure 4.10: Waste ratio for Case (3).

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 Figure 4.11. Note that the numbers we show here are averaged from 100 times of repeated experiments.

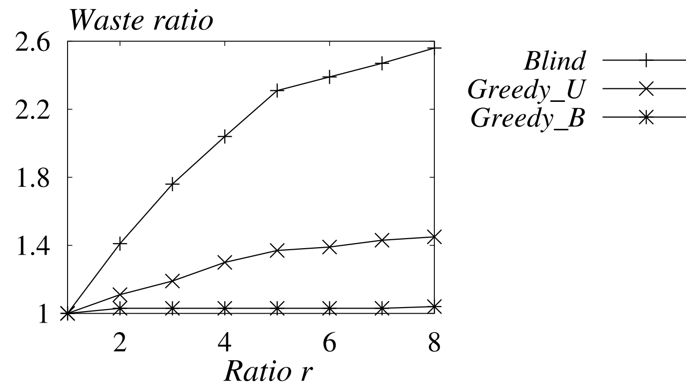


Figure 4.11: Waste ratio for Case (4).

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 Figure 4.6.

4.5 Summary

In this chapter, we discussed two problems of resource bricolage with constraints: 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 show 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.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This dissertation makes three contributions to query performance prediction and system performance improvement for database management systems.

For query performance prediction, we conducted experiments to evaluate two state-of-the-art progress indicators, and we found that they could provide inaccurate progress estimates. Most of these errors are due to a simplified uniform future speed assumption that they make. We proposed a new progress indicator, GSLPI, to provide more accurate query progress estimates.

In the second part of the thesis, our goal is to improve parallel database performance on heterogeneous hardware. To achieve the goal, we proposed a resource bricolage technique to come up with the proper amounts of data that should be allocated to the machines. To accomplish this, we first leveraged the techniques proposed in the previous part to estimate query execution times on different machines. We then took the time estimates as input and solved the problem with linear optimization techniques.

In the third part of the thesis, we studied two resource bricolage problems with additional constraints. One problem is to minimize workload execution time given a fixed amount of budget, and the other one is to minimize the cost given a target execution time. We solved both problems using mixed integer programming techniques.

5.2 Future Work

There are a number of interesting directions for further research following this work. In this section, we outline two such directions.

5.2.1 Progress Indicators on Steroids

To date, all of the prior work has primarily focused on making progress estimates as close to the actual running time as possible. The estimates are mainly used as feedback to users. However, there are a number of interesting questions that we want to ask. Does it always worth pushing hard to improve the accuracy? For some tasks, the current progress indicators may have already provided enough useful information. What other information can we infer from the progress estimates? Is it possible that we can make better use of this information? Now, let us take a step back to think about the following two motivating examples.

A simple progress score. Take, for instance, the progress indicator shipped with the Pig system [54] running map-reduce jobs. It monitors a task's progress using a score between 0 and 1. To compute this score, the 7 phases of a map-reduce job are divided into 4 pipelines: {Record reader, Map, Combine}, {Copy}, {Sort}, and {Reduce}. Among them, one is for the map task, and the other three is for reduce task. For a map task, since it only contains 1 pipeline, its score is defined as the percentage of bytes read from the

input data. For a reduce task, since it consists of 3 pipelines, each of these phases accounts for 1/3 of the score. In each pipeline, the score is the fraction of the data that has been processed. The overall score of a map-reduce job is computed as the average of the scores of these pipelines. The progress of a Pig Latin query is then the average score of all jobs in the query. As mentioned by other researchers [51], the progress score provided by the Pig system is not accurate. It assumes that all pipelines in the same query contribute equally to the overall score (e.g., it is assumed that all pipelines perform the same amount of work or take the same amount of time to process), which is rarely the case in practice. Despite the fact that the estimate is not perfect, this simple progress score has been used by Hadoop to select stragglers (i.e., tasks making slow progress compared to others) and to schedule speculative executions of these stragglers on other machines, with the hope of finishing the computation faster. Google has noted that speculative execution can improve a job's response time by 44% [20]. Follow-up work proposed a new scheduler, LATE, which further improved Hadoop response times by a factor of 2 by using a variation of this score [69].

A state-of-the-art progress indicator. Later, ParaTimer [50] was proposed for Pig Latin queries to provide more accurate time-oriented progress estimates. These progress estimates can be used to handle stragglers in a different way. When data skew happens (e.g., some tasks need to process more data than others), tasks with too much input data will be considered as stragglers. Because these tasks run more slowly due to their input data and not the machine where they have been scheduled, speculative executions of these tasks on other machines will not reduce their execution times. An alternative approach is needed. SkewTune [38] handles this problem by utilizing ParaTimer. When a slot becomes available and there are no pending tasks, the task with the longest expected remaining time (determined by ParaTimer) is selected as a straggler. The unprocessed input data of this task is repartitioned, and then they are added to existing partitions to form

new partitions, such that all new partitions complete at the same time. This detection-repartition cycle is repeated until all tasks complete. Their experimental results show that using this technique can significantly reduced job response time in the presence of skew.

For the first example with the simple progress score, although the score is only a rough approximation of the actual query execution time, the information it provides is sufficient for identifying stragglers. Thus, it is helpful for scheduling speculative executions. For the second example, identifying a straggler is not sufficient. To make all new partitions complete at the same time, SkewTune must also know the execution time more precisely. In both cases, rather than using the progress estimates as user feedback, they are used by the system to identify the bottlenecks (e.g., the longest running tasks) and help reduce the response time. Inspired by these examples, it seems that it may not always worth pushing hard on improving estimation accuracy. For some tasks, the current progress indicators are good enough to be useful. Thus, we think the following directions represent tremendous opportunities for future research.

First, we can change the "lens" through which we view the progress indicator technology in the future. When evaluating progress indicator technology, it would be better to focus on the following questions: (1) Is the progress indicator good enough, so that it is more helpful than not for *specific* tasks? By analogy, query optimizer cost estimates do not have to be perfect to be useful. Similarly, progress indicator estimates do not have to be perfect to be useful for, say, scheduling. (2) Is the progress indicator accurate when nothing in its current universe of knowledge changes? Specifically, it would be silly to start a progress indicator, then add 100 "monster" queries, and say "*wow, that initial prediction was really terrible*!" As we mentioned before, when it comes to big data processing systems, future states are harder to predict. As "monster" queries start running on a system, as skew and failures happen, some tasks will slow down and become stragglers. Predicting the stragglers ahead of time may become too overwhelming for progress in-

dicators. Instead, we should ask, how good is a progress indicator given the information available at the time? (3) Finally, does the progress indicator react to changes quickly? When the situation does change, how long does it take a progress indicator to get to a good prediction given the new information? A progress indicator that takes forever to adjust, but is very accurate when it does, may be less useful than one that adjusts immediately but is somewhat inaccurate. For the two examples we discussed, the sooner we can identify the stragglers, the sooner we can schedule speculative executions and repartitions.

Second, we can expand the kind of information progress indicators can provide by expanding their sources of inputs. So far, progress indicators are mainly used to predict the percent finished or the remaining execution time of a task. This is also the most obvious and direct usage of progress indicators. We believe that there is much more useful information we can derive from the progress indicator technology. For example, ranking a set of tasks by their completion percentage will reveal which ones are stragglers that may need more attention. Furthermore, we can have a deeper analysis of the collected statistics for these stragglers to automatically detect their causes. In addition, the average progress made by tasks running on a specific machine will indicate whether the machine is healthy, so that more tasks or speculative executions of stragglers can be assigned to these machines to make better progress. When a task finishes, its resources are released. Knowing the remaining execution time of the running tasks and the resources reserved for them will give us an idea about resource availability in the future. An intriguing possibility that has not yet been explored to date is the use of changes in estimates over time as a first-class object for analysis (rather than the estimates themselves). For example, while Query 1 is running, Query 2 is started; the difference in the estimates for Query 1's running time before and after Query 2 was started may give useful, actionable information even when both estimates are inaccurate in the absolute sense.

To sum up, we think that, in the future, we can change how we view and evaluate

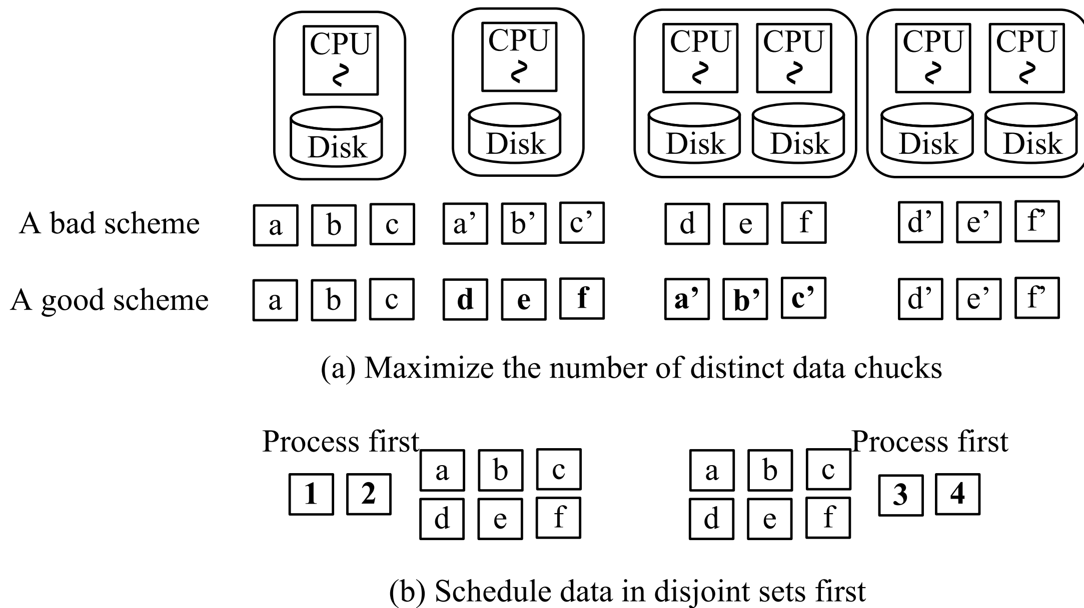
progress indicator technology, and the technology itself can be expanded in many different dimensions. The technology can expand the class of computations it can service, expand the kinds of predictions it can make, broaden the sources of inputs it monitor, and improve the accuracy of its predictions. We term such a greatly expanded progress indicator a "progress indicator on steroids", which we think is a very interesting topic for future work.

5.2.2 Improving MapReduce Performance in Heterogeneous Clusters

One focus of this thesis has been on improving database performance in heterogeneous clusters. Another possible direction to work on is to improve MapReduce performance running on heterogeneous hardware. MapReduce is an important programming model for processing large-scale applications. Previous research has shown that the disparity in MapReduce performance between homogeneous and heterogeneous clusters could be very high [12, 29].

Recently, an increasing number of algorithms and optimizations have been proposed to make MapReduce perform well on clusters with heterogeneous properties [69, 66, 12, 29]. The work in [69] proposed a scheduling algorithm, which uses estimated execution times to speculatively execute the tasks that hurt the response time the most. The work in [66] pointed out that ignoring data locality in heterogeneous environments may lead to remote tasks, which move data from the machines where they sit on to the machines where they are processed. These remote tasks can noticeably degrade the MapReduce performance by increasing network traffic significantly. The paper addressed the problem by placing data across machines in a way that each machine has a balanced data processing load. The underlying idea is similar to ours where high-performance machines

are expected to store and process more data. Later, the authors in [29] used 11 different benchmarks to test MapReduce performance on different clusters. They found that on a heterogeneous cluster of 10 Xeon-based servers and 80 Atom-based servers, Hadoop performs 20-75% worse than 10-node Xeon-only or 80-node Atom-only homogeneous sub-clusters for 6 out of 11 benchmarks. The performance degradation is mostly caused by remote tasks as well. A suite of optimizations called Tarazu are proposed by the authors to boost MapReduce performance. The follow-up work [29] developed a load rebalancing scheme to significantly improve Tarazu's performance over Hadoop.



(a) Maximize the number of distinct data chucks

(b) Schedule data in disjoint sets first

Figure 5.1: Strategies for improving MapReduce performance.

We observe that a lot of the previous work improves MapReduce performance by dealing with remote tasks using different strategies. Motivated by the previous work, we come up with two additional strategies that can further improve the performance by reducing the number of remote tasks. We will use the examples shown in Figure 5.1 to illustrate our idea. In our running example, we have four machines in the cluster,

where two of them are more powerful than the other two. We use a, b, ..., f to represent the data chunks that are allocated to the machines, and we use a', b', ..., f' to represent the replicated copies of the data chunks, respectively. In Figure 5.1(a), we show two possible data allocation schemes, and we believe that the second allocation scheme is better since it can potentially reduce the number of remote tasks. For the first scheme, the low-performance machines get data a, b, and c (a', b', and c' are duplicates), and the high-performance machines get data d, e, and f. While for the second scheme, the low-performance machines get data a, b, c, d, e, and f, and the high-performance machines also get data a, b, c, d, e, and f. Since the high-performance machines may be able to process data faster than the low-performance machines, when they are running out of data, they need to "borrow" data from the low-performance machines to continue working. When we allocate data using the second scheme, the high-performance machines get more distinct data chunks of their own. As a result, they need to borrow less data from others. In Figure 5.1(b), we are trying to show that a better scheduling may reduce remote tasks as well. Suppose that the low-performance machines have data a, b, c, d, e, f, 1 and 2, and the high-performance machines have data a, b, c, d, e, f, 3 and 4. For different types of machines, we think that a good strategy is to process data in their disjoint sets first. Since for the data that are commonly owned by different types of machines, they can be processed by any of them without data movement. Thus, we can reduce possible remote accesses.

MapReduce is a widely adopted framework for the distributed processing of large data sets, and our motivating examples seem very promising. Thus we think this could be an interesting direction for future work.

Bibliography

[1] Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing/.

[2] Amazon's Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[3] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] Google cloud platform pricing. https://cloud.google.com/pricing/.

[5] Hadoop. http://hadoop.apache.org/.

[6] Hp cloud pricing. http://www.hpcloud.com/pricing.

[7] IBM what is big data? - bringing big data to the enterprise. http://www-01.ibm.com/software/data/bigdata/.

[8] The Next Generation of Apache Hadoop MapReduce. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/.

[9] Rackspace cloud servers pricing. http://www.rackspace.com/cloud/servers/pricing/.

[10] SQL Server 2012 Parallel Data Warehouse. http://www.microsoft.com/en-us/sqlserver/solutions-technologies/data-warehousing/pdw.aspx.

[11] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. SIGMOD, 2004.

[12] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. ASPLOS, 2012.

[13] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. Learning-based query performance modeling and prediction. ICDE, 2012.

[14] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. ICDE, 2011.

[15] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. When can we trust progress estimators for SQL queries? In *SIGMOD*, 2005.

[16] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD*, pages 803–814, 2004.

[17] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 2010.

[18] George B. Dantzig and Mukund N. Thapa. *Linear Programming 1: Introduction*. Springer-Verlag, 1997.

[19] DB2. IBM DB2 query monitor for z/OS. ftp://ftp.software.ibm.com/software/data/db2imstools/ whitepapers/db2querymon-wp05.pdf, 2005.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. OSDI, 2004.

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

[22] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. VLDB, 1992.

[23] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[24] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. SIGMOD, 2011.

[25] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the performance fluctuations of HPC workloads on clouds. CLOUDCOM, 2010.

[26] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: exploiting performance heterogeneity in public clouds. SoCC, 2012.

[27] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[28] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. ICDE, 2009.

[29] Rohan Gandhi, Di Xie, and Y. Charlie Hu. PIKACHU: How to rebalance load in optimizing MapReduce on heterogeneous clusters. USENIX ATC, 2013.

[30] Greenplum. Database performance monitor. http://www.greenplum.com/pdf/Greenplum-Performance-Monitor.pdf.

[31] TPC Homepage. TPC-H benchmark. http://www.tpc.org.

[32] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. *SIGMOD Rec.*, 24, May 1995.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys, 2007.

[34] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16, June 1984.

[35] David S Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*.

[36] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Rec.*, 27, June 1998.

[37] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A statistical approach towards robust progress estimation. *PVLDB*, 2012.

[38] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: mitigating skew in MapReduce applications. SIGMOD, 2012.

[39] Jiexing Li, Rimma V. Nehme, and Jeffrey F. Naughton. GSLPI: A cost-based query progress indicator. In *ICDE*, 2012.

[40] Jiexing Li, Rimma V. Nehme, and Jeffrey F. Naughton. Toward progress indicators on steroids for big data systems. In *CIDR*, 2013.

[41] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Toward a progress indicator for database queries. In *SIGMOD*, 2004.

[42] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Increasing the accuracy and coverage of SQL progress indicators. In *ICDE*, 2005.

[43] Gang Luo, Jeffrey F. Naughton, and Philip S. Yu. Multi-query SQL progress indicators. In *EDBT*, 2006.

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

[45] Microsoft. Logical and physical operators reference. http://msdn.microsoft.com/en-us/library/ms191158.aspx.

[46] Microsoft. SQL Server. http://www.microsoft.com/sqlserver.

[47] Microsoft. Execution related dynamic management views and functions. http://msdn.microsoft.com/en-us/library/ms188068.aspx, 2010.

[48] Chaitanya Mishra and Nick Koudas. A lightweight online framework for query progress indicators. *ICDE*, 2007.

[49] Chaitanya Mishra and Nick Koudas. The design of a query monitoring system. *ACM Trans. Database Syst.*, 34:1:1–1:51, 2009.

[50] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[51] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *ICDE*, 2010.

[52] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. SIGMOD, 2011.

[53] Kenneth W. Ng, Zhenghao Wang, Richard R. Muntz, and Silvia Nittel. Dynamic query re-optimization. In *SSDBM*, 1999.

[54] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. SIGMOD, 2008.

[55] Oracle. Oracle database data warehousing guide. http://download.oracle.com/docs/cd/B19306_01/server.102/ b14223/bi.htm, 2005.

[56] Zhonghong Ou, Hao Zhuang, Andrey Lukyanenko, Jukka K. Nurminen, Pan Hui, Vladimir Mazalov, and Antti Yla-Jaaski. Is the same instance type created equal? Exploiting heterogeneity of public clouds. *IEEE Transactions on Cloud Computing*, pages 201–214, 2013.

[57] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. HotCloud, 2012.

[58] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. SIGMOD, 2012.

[59] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25, June 1996.

[60] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. SIGMOD, 2002.

[61] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. SoCC, 2012.

[62] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *PVLDB*, 2010.

[63] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. INFOCOM, 2010.

[64] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 2013.

[65] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, 2013.

[66] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. In *IPDPS Workshops*, 2010.

[67] Yu Xu and Pekka Kostamaa. Efficient outer join data skew handling in parallel DBMS. *PVLDB*, 2009.

[68] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. SIGMOD, 2008.

[69] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. OSDI, 2008.

[70] Hao Zhuang, Xin Liu, Zhonghong Ou, and Karl Aberer. Impact of instance seeking strategies on resource allocation in cloud data centers. CLOUD, 2013.

[71] Tao Zou, Ronan Le Bras, Marcos Vaz Salles, Alan Demers, and Johannes Gehrke. ClouDiA: A deployment advisor for public clouds. In *PVLDB*, 2013.

[72] Tao Zou, Guozhang Wang, Marcos Vaz Salles, David Bindel, Alan Demers, Johannes Gehrke, and Walker White. Making time-stepped applications tick in the cloud. In *SOCC*, 2011.