Automatic Tuning of Data-Intensive Analytical Workloads

by

Herodotos Herodotou

Department of Computer Science
Duke University

Date: ____________________

Approved:

____
Shivnath Babu, Supervisor

____
Jun Yang

____
Jeffrey Chase

____
Christopher Olston

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

Abstract

Automatic Tuning of Data-Intensive Analytical Workloads

by

Herodotos Herodotou

Department of Computer Science
Duke University

Date: ____________________
Approved:

____
Shivnath Babu, Supervisor

____
Jun Yang

____
Jeffrey Chase

____
Christopher Olston

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

Copyright © 2012 by Herodotos Herodotou
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

Modern industrial, government, and academic organizations are collecting massive amounts of data ("*Big Data*") at an unprecedented scale and pace. The ability to perform timely and cost-effective analytical processing of such large datasets in order to extract deep insights is now a key ingredient for success. These insights can drive automated processes for advertisement placement, improve customer relationship management, and lead to major scientific breakthroughs.

Existing database systems are adapting to the new status quo while large-scale dataflow systems (like Dryad and MapReduce) are becoming popular for executing analytical workloads on Big Data. Ensuring good and robust performance automatically on such systems poses several challenges. First, workloads often analyze a hybrid mix of structured and unstructured datasets stored in nontraditional data layouts. The structure and properties of the data may not be known upfront, and will evolve over time. Complex analysis techniques and rapid development needs necessitate the use of both declarative and procedural programming languages for workload specification. Finally, the space of workload tuning choices is very large and high-dimensional, spanning configuration parameter settings, cluster resource provisioning (spurred by recent innovations in cloud computing), and data layouts.

We have developed a novel dynamic optimization approach that can form the basis for tuning workload performance automatically across different tuning scenarios and systems. Our solution is based on (i) collecting monitoring information in

order to learn the run-time behavior of workloads, (ii) deploying appropriate models to predict the impact of hypothetical tuning choices on workload behavior, and (iii) using efficient search strategies to find tuning choices that give good workload performance. The dynamic nature enables our solution to overcome the new challenges posed by Big Data, and also makes our solution applicable to both MapReduce and Database systems. We have developed the first cost-based optimization framework for MapReduce systems for determining the cluster resources and configuration parameter settings to meet desired requirements on execution time and cost for a given analytic workload. We have also developed a novel tuning-based optimizer in Database systems to collect targeted run-time information, perform optimization, and repeat as needed to perform fine-grained tuning of SQL queries.

Contents

Abstract **iv**

List of Tables **x**

List of Figures **xii**

Acknowledgements **xvii**

1 Analytical Processing in the Big Data Era **1**

1.1 MADDER Principles in Big Data Analytics 2

1.2 Two Approaches to Big Data Analytics 3

1.3 Big Data Analytics Systems are Becoming MADDER 6

1.4 Challenges in Tuning MADDER Systems 8

1.5 Contributions 9

2 A Tuning Approach for MADDER Systems **14**

2.1 Current Approaches to Optimization and Tuning 14

2.1.1 Self-tuning Database Systems 15

2.1.2 Optimizing Dataflow Systems 17

2.2 Overview of a MADDER Tuning Approach 19

2.2.1 Tuning MapReduce Workloads with Starfish 20

2.2.2 Tuning SQL Queries with Xplus 26

3 Primer on Tuning MapReduce Workloads **28**

3.1 MapReduce Job Execution 29

3.2 Impact of Configuration Parameter Settings 35

3.3 MapReduce on the Cloud 39

3.4 Use Cases for Tuning MapReduce Workloads 40

4 Dynamic Profiling of MapReduce Workloads 46

4.1 Job and Workflow Profiles 47

4.2 Using Profiles to Analyze Execution Behavior 52

4.3 Generating Profiles via Measurement 54

4.4 Task-level Sampling to Generate Approximate Profiles 58

5 A Declarative Query Interface to Access Performance Predictors and Optimizers 61

5.1 Declarative Interface to Express Workload Tuning Queries 63

5.2 Overview of How Starfish Answers a Workload Tuning Query 66

5.3 Starfish Visualizer 70

6 Predicting MapReduce Workload Performance 78

6.1 Overview for Predicting MapReduce Workload Performance 82

6.2 Cardinality Models to Estimate Dataflow Statistics Fields 83

6.3 Relative Black-box Models to Estimate Cost Statistics Fields 84

6.4 Analytical Models to Estimate Dataflow and Cost Fields 88

6.4.1 Modeling the Read and Map Phases in the Map Task 91

6.4.2 Modeling the Collect and Spill Phases in the Map Task 92

6.4.3 Modeling the Merge Phase in the Map Task 94

6.4.4 Modeling the Shuffle Phase in the Reduce Task 98

6.4.5 Modeling the Merge Phase in the Reduce Task 103

6.4.6 Modeling the Reduce and Write Phases in the Reduce Task . . 108

6.5 Simulating the Execution of a MapReduce Workload 109

6.6 Estimating Derived Data Properties and Workflow Performance . . . 110

6.7 Evaluating the Predictive Power of the What-if Engine 111
6.7.1 Accuracy of What-if Analysis 113
6.7.2 Tuning the Cluster Size 115
6.7.3 Transitioning from Development to Production 116
6.7.4 Evaluating the Training Benchmarks 117
7 Cost-based Optimization for MapReduce Workloads 121
7.1 Current Approaches to MapReduce Optimization 124
7.2 Cost-based Optimization of MapReduce Jobs 129
7.2.1 Subspace Enumeration 131
7.2.2 Search Strategy within a Subspace 132
7.2.3 Evaluating Cost-based Job Optimization 134
7.3 Cost-based Optimization of MapReduce Workflows 143
7.3.1 Dataflow and Resource Dependencies in Workflows 144
7.3.2 MapReduce Workflow Optimizers 150
7.3.3 Evaluating Cost-based Workflow Optimization 159
7.4 Cost-based Optimization of Cluster Resources 170
7.4.1 Cluster Resource Optimizer 171
7.4.2 Evaluating Cost-based Cluster Provisioning 172
8 An Experiment-driven Approach to Tuning Analytical Queries 175
8.1 New Representation of the Physical Plan Space 181
8.2 New Search Strategy over the Physical Plan Space 185
8.2.1 Enumerating Neighborhoods and Plans 185
8.2.2 Picking the Neighborhoods to Cover 188
8.2.3 Picking the Plan to Run in a Neighborhood 192
8.3 Implementation of Xplus 193

8.3.1 Architecture . . . 193
8.3.2 Extensibility Features . . . 194
8.3.3 Efficiency Features . . . 196
8.4 Comparing Xplus to Other SQL-tuning Approaches . . . 198
8.5 Experimental Evaluation . . . 202
8.5.1 Overall Performance of Xplus . . . 204
8.5.2 Comparison with Other SQL-tuning Approaches . . . 205
8.5.3 Internal Comparisons for Xplus . . . 207
9 Increasing Partition-awareness in Cost-based Query Optimization 210
9.1 Optimization Opportunities for Partitioned Tables . . . 212
9.2 Related Work on Table Partitioning . . . 217
9.3 Query Optimization Techniques for Partitioned Tables . . . 219
9.3.1 Matching Phase . . . 222
9.3.2 Clustering Phase . . . 227
9.3.3 Path Creation and Selection . . . 229
9.3.4 Extending our Techniques to Parallel Database Systems . . . 235
9.4 Experimental Evaluation . . . 236
9.4.1 Results for Different Partitioning Schemes . . . 237
9.4.2 Studying Optimization Factors on Table Partitioning . . . 241
9.4.3 Impact on Cardinality Estimation . . . 245
10 The Future of Big Data Analytics 247
10.1 Starfish: Present and Future . . . 248
10.2 Xplus: Present and Future . . . 251
Bibliography 253
Biography 265

List of Tables

3.1 A subset of important job configuration parameters in Hadoop. . . . 33

3.2 Five representative Amazon EC2 node types, along with resources and monetary costs. 40

4.1 Dataflow fields in the job profile. 48

4.2 Cost fields in the job profile. 49

4.3 Dataflow statistics fields in the job profile. 50

4.4 Cost statistics fields in the job profile. 51

4.5 A subset of job profile fields for two Word Co-occurrence jobs run with different settings for *io.sort.mb*. 52

6.1 Example questions the What-if Engine can answer. 79

6.2 A subset of cluster-wide and job-level Hadoop parameters. 89

6.3 Cluster-wide Hadoop parameter settings for five EC2 node types. . . 111

6.4 MapReduce programs and corresponding datasets for the evaluation of the What-if Engine. 111

7.1 MapReduce programs and corresponding datasets for the evaluation of the Job Optimizer. 134

7.2 MapReduce job configuration settings in Hadoop suggested by Rules-of-Thumb and the cost-based Job Optimizer for the Word Co-occurrence program. 135

7.3 Number of reduce tasks chosen and speedup over Rules-of-Thumb settings by the Workflow Optimizer for a workflow as we vary the total input size. 150

7.4 MapReduce workflows and corresponding dataset sizes on two clusters for the evaluation of the Workflow Optimizer. 160

8.1 Properties of the current experts in Xplus. 189

8.2 Comparison of Xplus, Leo, Pay-As-You-Go, and ATO. 199

8.3 Tuning scenarios created with TPC-H queries. 203

8.4 Overall tuning results of Xplus for TPC-H queries. 204

8.5 Tuning results of Xplus, Leo controller, and ATO controller when asked to find a 5x better plan. 207

9.1 Uses of table partitioning in Database systems. 211

9.2 Optimizer categories considered in the experimental evaluation. . . . 237

9.3 Partitioning schemes for TPC-H databases. 238

List of Figures

1.1 Typical architecture for Database systems. 4

1.2 Typical architecture for Dataflow systems. 5

1.3 Summary of contributions. 11

2.1 Starfish in the Hadoop ecosystem. 21

2.2 Components in the Starfish architecture. 23

3.1 Execution of a MapReduce job. 29

3.2 Execution of a map task showing the map-side phases. 30

3.3 Execution of a reduce task showing the reduce-side phases. 30

3.4 An example MapReduce workflow. 34

3.5 Response surfaces of WordCount MapReduce jobs in Hadoop. 36

3.6 Response surfaces of TeraSort MapReduce jobs in Hadoop. 36

3.7 Performance Vs. pay-as-you-go costs for a workload that is run on different EC2 cluster resource configurations. 43

3.8 Pay-as-you-go costs for a workload run on Hadoop clusters using auction-based EC2 spot instances. 44

4.1 Map and reduce time breakdown for two Word Co-occurrence jobs run with different settings for *io.sort.mb*. 53

4.2 Total map execution time, Spill time, and Merge time for a representative Word Co-occurrence map task as we vary the setting of *io.sort.mb*. 54

4.3 (a) Overhead to measure the (approximate) profile, and (b) corresponding speedup given by Starfish as the percentage of profiled tasks is varied for Word Co-occurrence and TeraSort MapReduce jobs. . . . 59

5.1 Screenshot from the Starfish Visualizer showing the execution timeline of the map and reduce tasks of a MapReduce job running on a Hadoop cluster. 71

5.2 Screenshot from the Starfish Visualizer showing a histogram of the map output data size per map task. 72

5.3 Screenshot from the Starfish Visualizer showing a visual representation of the data flow among the Hadoop nodes during a MapReduce job execution. 73

5.4 Screenshot from the Starfish Visualizer showing the map and reduce time breakdown from the virtual profile of a MapReduce job. 74

5.5 Screenshot from the Starfish Visualizer showing the optimal configuration parameter settings found by the Job Optimizer, as well as cluster and input data properties. 75

6.1 Overall process used by the What-if Engine to predict the performance of a given MapReduce workflow. 82

6.2 Overall process used by the What-if Engine to estimate a virtual job profile. 83

6.3 Map and reduce time breakdown for Word Co-occurrence jobs from (A) an actual run and (B) as predicted by the What-if Engine. 113

6.4 Actual Vs. predicted running times for (a) Word Co-occurrence, (b) WordCount, and (c) TeraSort jobs running with different configuration parameter settings. 114

6.5 Actual and predicted running times for MapReduce jobs as the number of nodes in the cluster is varied. 115

6.6 Actual and predicted running times for MapReduce jobs when run on the production cluster. 116

6.7 Total running time for each training benchmark. 118

6.8 Relative prediction error for the Fixed and Custom benchmarks over the Apriori benchmark when asked to predict cost statistics for a test workload. 119

6.9 Relative prediction error for the Fixed and Custom benchmarks over the Apriori benchmark without TF-IDF when asked to predict cost statistics for the TF-IDF job. 120

7.1 Overall process for optimizing a MapReduce job. 131

7.2 Map and reduce time breakdown for two Word Co-occurrence jobs run with configuration settings suggested by Rules-of-Thumb and the cost-based Job Optimizer. 136

7.3 Running times for MapReduce jobs running with Hadoop's Default, Rules-of-Thumb, and CBO-suggested settings. 138

7.4 Optimization time for the six Cost-based Optimizers for various MapReduce jobs. 139

7.5 Number of what-if calls made (unique configuration settings considered) by the six Cost-based Optimizers for various MapReduce jobs. . 139

7.6 The job execution times for TeraSort when run with (a) Rules-of-Thumb settings, (b) CBO-suggested settings using a job profile obtained from running the job on the corresponding data size, and (c) CBO-suggested settings using a job profile obtained from running the job on 5GB of data. 140

7.7 The job execution times for MapReduce programs when run with (a) Rules-of-Thumb settings, (b) CBO-suggested settings using a job profile obtained from running the job on the production cluster, and (c) CBO-suggested settings using a job profile obtained from running the job on the development cluster. 141

7.8 Percentage overhead of profiling on the execution time of MapReduce jobs as the percentage of profiled tasks in a job is varied. 142

7.9 Speedup over the job run with Rules-of-Thumb settings as the percentage of profiled tasks used to generate the job profile is varied. . . 143

7.10 MapReduce workflows for (a) Query H4 from the Hive Performance Benchmark; (b) Queries P1 and P7 from the PigMix Benchmark run as one workflow; (c) Our custom example. 144

7.11 Execution times for jobs in a workflow when run with settings suggested by (a) popular Rules of Thumb; (b) a Job-level Workflow Optimizer; (c) an Interaction-aware Workflow Optimizer. 146

7.12 Execution timeline for jobs in a workflow when run with settings suggested by (a) popular Rules of Thumb; (b) an Interaction-aware Workflow Optimizer. 148

7.13 The optimization units (denoted with dotted boxes) for our example MapReduce workflow for the three Workflow Optimizers. 153

7.14 Speedup achieved over the Rules-of-Thumb settings for workflows running on the Amazon cluster from (a) the TPC-H Benchmark, (b) the PigMix Benchmark, and (c) the Hive Performance Benchmark. 162

7.15 Speedup achieved over the Rules-of-Thumb settings for workflows running on the Yahoo! cluster from (a) the TPC-H Benchmark, (b) the PigMix Benchmark, and (c) the Hive Performance Benchmark. 163

7.16 Optimization overhead for all MapReduce workflows. 164

7.17 Running times of queries with settings based on Rules-of-Thumb, the Job-level Workflow Optimizer, and the Interaction-aware Workflow Optimizer. 165

7.18 Optimization times for the Job-level Workflow Optimizer and the Interaction-aware Workflow Optimizer. 166

7.19 Running times of queries with settings based on Rules-of-Thumb, the Single-configuration Workflow Optimizer, and the Interaction-aware Workflow Optimizer. 167

7.20 Optimization times for the Single-configuration Workflow Optimizer and the Interaction-aware Workflow Optimizer. 168

7.21 Running times of queries with settings based on Rules-of-Thumb, and the Static and Dynamic Interaction-aware Workflow Optimizers. . . . 169

7.22 Running times of queries with settings based on Rules-of-Thumb, and the Static and Dynamic Interaction-aware Workflow Optimizers, as we vary the actual filter ratio of job j_1. 170

7.23 Running time and monetary cost of the workload when run with (a) Rules-of-Thumb settings and (b) Starfish-suggested settings, while varying the number of nodes and node types in the clusters. 173

8.1 Neighborhoods and physical plans for our example star-join query. . . 179

8.2 Neighborhood and Cardinality Tables for our example star-join query. 183

8.3 System architecture of Xplus. 193

8.4 Progress of the execution time of the best plan in the covered space as Xplus tunes TPC-H Query 7. 205

8.5 Speedup from Xplus, Leo, and ATO controllers. 206

8.6 (a) Convergence times and (b) completion times for the expert-selection policies. 208

8.7 Impact of the efficiency features in Xplus. 209

9.1 Partitioning of tables R, S, T, and U. Dotted lines show partitions with potentially joining records. 212

9.2 P_1 is a plan generated by current optimizers for the running example query Q. P_2 is a plan generated by our partition-aware optimizer. . . 214

9.3 A partition index tree containing intervals for all child tables (partitions) of T from Figure 9.1. 223

9.4 Matching algorithm. 224

9.5 Clustering algorithm applied to the running example query Q. 227

9.6 Clustering algorithm. 228

9.7 Logical relations (with child relations) enumerated for query Q by our partition-aware bottom-up optimizer. 233

9.8 (a) Execution times and (b) optimization times for TPC-H queries over partitioning scheme PS-J. 239

9.9 (a) Execution times and (b) optimization times for TPC-H queries 5 and 8 over three partitioning schemes. 240

9.10 (a) Execution times and (b) optimization times for TPC-H queries over partitioning scheme PS-C with partition size 128MB. 241

9.11 (a) Execution times and (b) optimization times as we vary the partition size for TPC-H queries 5 and 8. 242

9.12 Execution times as we vary the total data size. 243

9.13 (a) Execution times and (b) optimization times for enabling and disabling clustering. 244

9.14 Estimated and actual number of records produced by TPC-H queries over partitioning scheme PS-C. 245

Acknowledgements

I am sincerely and heartily grateful to my advisor, Shivnath Babu, for the support and guidance he showed me throughout my graduate studies at Duke University. I would also like to extend my gratitude to the rest of my committee members—Jun Yang, Jeffrey S. Chase, and Christopher Olston—for their support and advice during this work.

I would like to acknowledge and thank those who have contributed to my personal growth and the growth of my work: Benjamin Reed and Jianjun Chen of Yahoo! Research for many valuable discussion on the workflow optimization problem; John Cieslewicz and Eric Friedman of Aster Data for introducing me to the query optimization problem over partitioned tables; and my colleagues Fei Dong and Nick Bodnar for helping develop the Starfish Visualizer.

I am also grateful to Amazon Web Services for awarding us multiple research grants for the use of resources on the Amazon Elastic Compute Cloud (EC2), and to Yahoo! for establishing a research collaboration for the use of Yahoo! resources.

Lastly, I offer my regards and blessings to my family and friends who supported me during the development of this dissertation.

1

Analytical Processing in the Big Data Era

Modern industrial, government, and academic organizations are collecting massive amounts of data ("*Big Data*") at an unprecedented scale and pace. Companies like Facebook, Yahoo!, and eBay maintain and process petabytes of data, including product sales, customer interactions, web content, software logs, click streams, and other types of information (Thusoo et al., 2009). On the scientific front, powerful telescopes in astronomy, genome sequencers in biology, and particle accelerators in physics are putting massive amounts of data into the hands of scientists. At the same time, many basic and applied science disciplines now have computational subareas, e.g., computational biology, computational economics, and computational journalism, expanding even further the need for versatile Big Data analytics.

Advanced analysis techniques (like data mining, statistical modeling, and inferential statistics) are now applied routinely to Big Data. These techniques drive automated processes for spam and fraud detection, advertisement placement, Website optimization, and customer relationship management; and lead to cost savings and higher revenue. Moreover, key scientific breakthroughs are expected to come from computational analysis of the collected scientific data. However, success in the

Big Data era is about more than the ability to process large amounts of data; it is about getting deep insights and learning from these huge datasets in a timely and cost-effective way (Cohen et al., 2009).

1.1 MADDER Principles in Big Data Analytics

Large-scale data-intensive computation has recently attracted a tremendous amount of attention both in the research community and in industry. Existing systems used for Big Data analytics are constantly improving while new systems are emerging. Regardless of the system used, *data scientists* now expect and need certain important features from these systems. Cohen et al. (2009) recently coined the acronym *MAD*—for *Magnetism*, *Agility*, and *Depth*—to express three such features:

- **Magnetism:** A magnetic system attracts all sources of data irrespective of issues like possible presence of outliers, unknown schema or lack of structure, and missing values that keep many useful data sources out of conventional data warehouses.

- **Agility:** An agile system adapts in sync with rapid data evolution, instead of retaining strict, predefined designs and planning decisions.

- **Depth:** A deep system supports analytics needs that go far beyond conventional rollups and drilldowns to complex statistical and machine-learning analysis.

In addition to MAD, three more features are becoming important in today's analytics systems: *Data-lifecycle-awareness*, *Elasticity*, and *Robustness*. A system with all six features would be *MADDER* than current analytics systems.

- **Data-lifecycle-awareness:** A data-lifecycle-aware system goes beyond query execution to optimize the movement, storage, and processing of data during its

entire lifecycle. This feature can: (i) support terabyte-scale daily data cycles that power the intelligence embedded in sites like Yahoo! and LinkedIn (Kreps, 2009); (ii) eliminate indiscriminate data copying that causes bloated storage needs (Ratzesberger, 2010); and (iii) realize performance gains due to reuse of intermediate data or learned metadata in workflows that are part of the cycle (Gunda et al., 2010).

- **Elasticity:** An elastic system adjusts its resource usage and operational costs to the user and workload requirements. Services like Amazon Elastic Compute Cloud (EC2) have created a market for pay-as-you-go analytics hosted on the cloud. Amazon EC2 provisions and releases clusters on demand, sparing users the hassle of cluster setup and maintenance.

- **Robustness:** A robust system continues to provide service, possibly with graceful degradation, in the face of undesired events like hardware failures, software bugs, and data corruption (Kreps, 2009).

1.2 Two Approaches to Big Data Analytics

Systems for Big Data analytics can be divided into two main categories: *Database systems* and *Dataflow systems*. Traditionally, Enterprise Data Warehouses (EDWs) and Business Intelligence (BI) tools built on top of Database systems have been providing the means for retrieving and analyzing large amounts of data. Massive Parallel Processing (MPP) database management systems that run on a cluster of commodity servers (like Teradata (Teradata, 2012), Aster nCluster (AsterData, 2012), and Greenplum (Greenplum, 2012)) provide support for Big Data analytics in the world of EDWs. On the other hand, large-scale parallel Dataflow systems, like Google's MapReduce (Dean and Ghemawat, 2004), Hadoop (Hadoop, 2012), Pig (Gates et al., 2009), Hive (Thusoo et al., 2009), and Dryad (Isard et al., 2007)), have emerged more

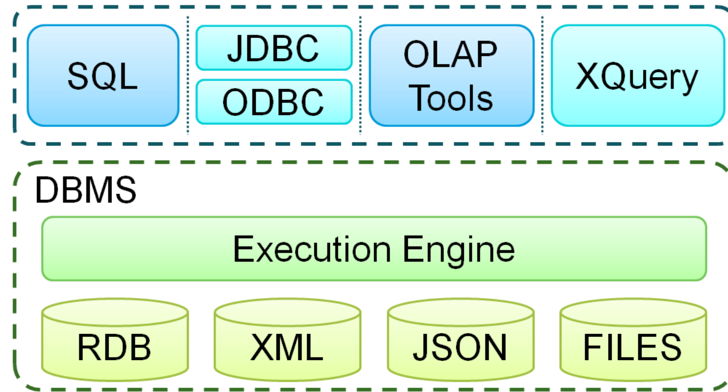


FIGURE 1.1: Typical architecture for Database systems.

recently and are becoming increasingly popular in the enterprise setting as well as in scientific and academic settings.

The typical architectures for Database and Dataflow systems are respectively shown in Figures 1.1 and 1.2. These two classes of systems make different choices in several key areas:

Data storage and structure: Most Database Management Systems (DBMSs) require that data conform to a well-defined schema and are stored in a specialized data store. Typically, the data structure follows the relational model even though other models, like XML or JSON, are supported. This well-defined structure of data allows DBMSs to maintain detailed statistics about the data and to build auxiliary index structures. On the other hand, Dataflow systems typically process data directly from the file system, permitting data to be in any arbitrary format. Hence, Dataflow systems are capable of processing unstructured, semi-structured, and structured data alike.

Programming model and interfaces: Structured Query Language (SQL) is the declarative language most widely used for accessing, managing, and analyzing data in Relational DBMSs. Users can specify an analysis task using an SQL query, and the DBMS will optimize and execute the query. In addition to SQL, most DBMSs

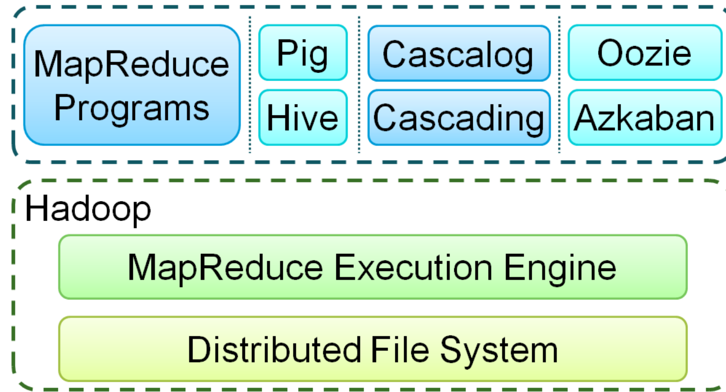


FIGURE 1.2: Typical architecture for Dataflow systems.

support (i) user-defined functions, user-defined aggregates, and stored procedures, (ii) interfaces (e.g., JDBC, ODBC) for accessing data from higher-level programming languages (e.g., Java, C), and (iii) XQuery, a query and functional programming language that is designed to query collections of XML data (see Figure 1.1). Apart from an execution engine, MapReduce is also a programming model inspired by functional programming (presented in detail in Chapter 3). This model, although highly flexible, has been found to be too low-level for routine use by practitioners such as data analysts, statisticians, and scientists (Olston et al., 2008b; Thusoo et al., 2009). As a result, the MapReduce framework has evolved rapidly over the past few years into a *MapReduce stack* (see Figure 1.2) that includes a number of higher-level layers added over the core MapReduce engine. Prominent examples of these higher-level layers include *Hive* (with an SQL-like declarative interface), *Pig* (with an interface that mixes declarative and procedural elements), *Cascading* (with a Java interface for specifying workflows), *Cascalog* (with a Datalog-inspired interface), and *BigSheets* (includes a spreadsheet interface).

Execution strategy: In a DBMS execution engine, input data are accessed and processed using a *query execution plan* that specifies how a given query will actually run in the system. The execution plan is composed of operators—e.g., index scan,

filter, sort, and hash join—from a known and fixed set. The input data are then *pushed* through the execution plan in order to generate the final results. Typically, the number of possible and valid execution plans is very large. It is the responsibility of the *query optimizer* to choose the optimal execution plan based on available data statistics, applicable operators, and cost models. In the Dataflow world, the typical execution engine used is the *MapReduce engine*. MapReduce engines represent a new data-processing framework that has emerged in recent years to deal with data at massive scale (Dean and Ghemawat, 2004). Users specify computations over large datasets in terms of Map and Reduce functions, and the underlying run-time system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disk bandwidth. Similar to DBMSs, there is also a large number of choices to be made in terms of *configuration parameter settings* that can affect the performance of a MapReduce job. However, MapReduce engines are not yet equipped with an optimizer that can make those choices automatically.

Though it may seem that Database and Dataflow systems target different applications, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates) or a set of MapReduce jobs (Pavlo et al., 2009). Following this observation, we will argue that a common approach can be taken for automatically tuning systems from both categories, especially as the systems keep evolving and becoming MADDER.

1.3 Big Data Analytics Systems are Becoming MADDER

Existing systems are changing and new systems are being created to support the MADDER principles. In particular, many DBMSs now offer support for analyzing semi-structured data, for performing Extract-Transform-Load (ETL) operations

from various heterogeneous data sources, and for processing data directly from files; making DBMSs more magnetic and agile. In addition, there is a growing interest among database vendors in providing query interfaces that combine the best features of SQL and MapReduce. SQL/MapReduce from Aster nCluster is a new framework for user-defined functions (UDFs) that leverages ideas from the MapReduce programming paradigm. UDFs in SQL/MapReduce are self-describing, polymorphic, and inherently parallelizable (Friedman et al., 2009). In an attempt to become MAD, Greenplum introduced a three-layered schema architecture as well as SQL extensions for vector arithmetic and data partitioning (Cohen et al., 2009).

Hadoop, the most popular open-source implementation of MapReduce, is becoming popular for Big Data analytics. Hadoop is already a MAD system. Hadoop has two primary components: a MapReduce execution engine and a distributed file-system. Analytics with Hadoop involves loading data as files into the distributed file-system, and then running parallel MapReduce computations to interpret and process the data. Working with (possibly) unstructured files as well as interpreting data (lazily) at processing time instead of (eagerly) at loading time makes Hadoop a magnetic and agile system. Furthermore, MapReduce computations in Hadoop can be expressed directly in general-purpose programming languages like Java or Python, domain-specific languages like R, or generated automatically from higher-level languages like HiveQL and Pig Latin. This coverage of the language spectrum makes Hadoop well suited for deep analytics. Finally, an unheralded aspect of Hadoop is its extensibility, i.e., the ease with which many of Hadoop's core components like the scheduler, storage subsystem, input/output data formats, data partitioner, compression algorithms, caching layer, and monitoring can be customized or replaced.

Hadoop has the core mechanisms to be MADDER than existing analytics systems. However, the use of most of these mechanisms has to be managed manually. Take elasticity as an example. Hadoop supports dynamic node addition as well as

decommissioning of failed or surplus nodes. However, Hadoop lacks control modules to decide (a) when to add new nodes or to drop surplus nodes, and (b) when and how to re-balance the data layout in this process.

1.4 Challenges in Tuning MADDER Systems

The traditional data warehousing philosophy demands a tightly controlled environment that makes it easier to meet performance requirements; a luxury we cannot afford any more (Cohen et al., 2009). The MADDER principles introduce new optimization challenges for ensuring good and robust performance across the different types of data analytics systems:

Less control on data storage and structure: Magnetism and agility—that come with supporting multiple (possibly heterogeneous) data sources and interpreting data only at processing time—imply less control over the data structure and layouts. The structure and properties of the data, like statistics and schema, may not be known initially, and will evolve over time. The data may be generated in different formats and stored as few large files, millions of small files, or anything in between. Such uncontrolled data layouts are a marked contrast to the carefully-planned layouts in Database systems. Hence, newer systems may have limited or no access to data properties that are traditionally used to drive optimization and tuning.

Less control on workload specification: The MADDER principles also imply less control on how workloads are given to the system. The increased need for deep analytics by a very diverse user community—ranging from marketing analysts and sales managers to scientists, statisticians, and systems researchers—fuels the use of programming languages like Java and Python in place of the conventional SQL queries. Even SQL queries are now laced commonly with user-defined functions and aggregates, which are very hard to accurately account for in traditional cost models.

Intertwining optimization and provisioning decisions: The elastic and pay-as-you-go nature of Infrastructure-as-a-Service (IaaS) cloud platforms are particularly attractive for small to medium organizations that need to process large datasets. A user can now provision clusters almost instantaneously and pay only for the resources used and duration of use. These features give tremendous power to the user, but they simultaneously place a major burden on her shoulders. The user is now faced regularly with complex decisions that involve finding the cluster size, the type of resources to use in the cluster from the large number of choices offered by current IaaS cloud platforms, and the job execution plans that best meet the performance needs of her workload. Hence, resource provisioning decisions are now intertwined with job-level optimization decisions, which further increases the size and dimensionality of the space of workload tuning choices.

As indicated by the aforementioned challenges, getting the desired performance from a MADDER system can be a nontrivial exercise. The practitioners of Big Data analytics like data analysts, computational scientists, and systems researchers usually lack the expertise to tune system internals. Such users would rather use a system that can tune itself and provide good performance automatically. However, the high uncertainty regarding data and task specification resulting from the MADDER principles renders traditional static optimization approaches ineffective.

1.5 Contributions

In this dissertation, we propose a novel dynamic optimization approach that can be used for automatically tuning the workload and cluster performance in MADDER systems. The approach is based on (i) collecting monitoring information (termed *dynamic profiling*) in order to learn the run-time behavior of complex workloads noninvasively, (ii) deploying appropriate models to predict the impact of hypothetical

tuning choices on workload behavior, and (iii) using efficient search strategies to find tuning choices that give good workload performance. This *profile-predict-optimize* approach forms the basis for automatically tuning a MADDER Database or Dataflow system, and can be used as needed for a wide spectrum of tuning scenarios.

To illustrate the breadth of possible tuning scenarios, suppose we are given a workload W to tune. Initially, the system has limited knowledge about the workload and the data, and an initial execution plan P_0 is selected. While P_0 is running, the system can profile the plan execution to learn the run-time behavior of W, and then employ the profile-predict-optimize approach to find a better plan P_1. When W gets submitted again for execution, the improved plan P_1 can be used instead of P_0. The system could then repeat the profile-predict-optimize approach to find an even better plan P_2 based on the profiled execution of P_1. Therefore, the system learns over time and executes better and better plans for W.

In another tuning scenario, suppose the user knows that W is an important, repeatedly-run workload for which she is willing to invest some resources upfront to find the optimal plan P_{opt}. The system can employ the profile-predict-optimize approach iteratively; that is, the system can collect some targeted profile information, perform optimization, and iterate as needed to perform fine-grained tuning. The main challenge here is to perform the minimum amount of profiling required to reach the optimal plan as quickly as possible. In yet another tuning scenario, the user can offload the tuning process to a backup or test system in order to minimize the impact on the production system.

Therefore, the profile-predict-optimize approach can be used for a variety of tuning scenarios, as well as for tuning workloads on different systems. (Note that the above discussion applies to both Database and Dataflow systems alike.) Figure 1.3 summarizes the contributions as they relate to profiling, predicting, and optimizing the performance of analytics workloads running on MADDER systems. Overall, our

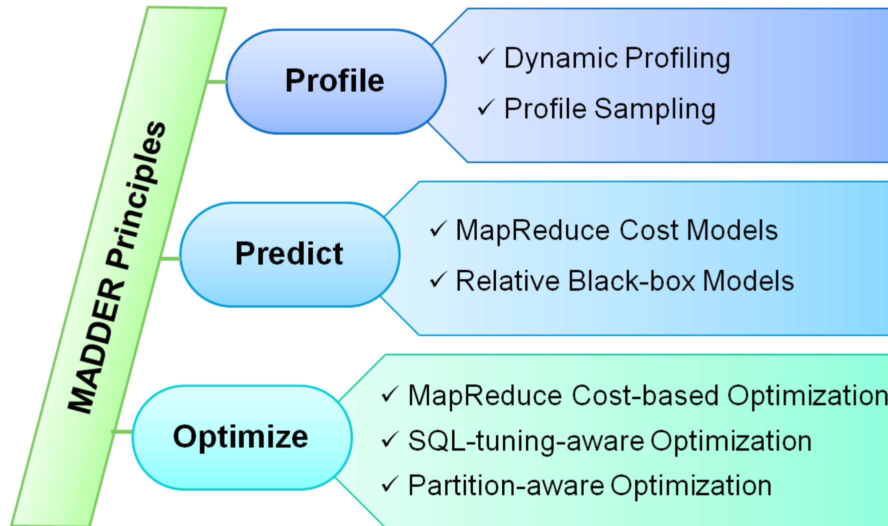


FIGURE 1.3: Summary of contributions.

contributions are as follows:

1. **Dynamic profiling:** We introduce dynamic profiling for collecting fine-grained, run-time monitoring information about a workload, which is needed for performance visualization, prediction, and optimization. Profiling works regardless of the amount of prior information the system has about the data processed or the tasks performed by the workload.

2. **Profile sampling:** Apart from ensuring noninvasive and dynamic profiling, another key challenge was keeping the run-time overhead low. For this purpose, we make use of sampling techniques to collect profiling information quickly and accurately.

3. **MapReduce cost models:** We have designed detailed cost models for predicting the performance of MapReduce workloads running on a cluster. These models take into account complex issues that arise in distributed settings such as task parallelism, scheduling, and interactions among concurrent or sequential tasks.

4. **Relative black-box models:** Common tuning scenarios require predicting and optimizing the performance of workloads across different clusters. For instance, it is typical to test the execution of workloads on a development cluster before staging for execution on the production cluster. We employ relative black-box models to accurately predict the execution behavior of a workload on one cluster, given profiling information for the execution on a different one.

5. **MapReduce cost-based optimization:** Optimization involves enumerating and searching through a large and high-dimensional space of tuning choices. We have developed, to the best of our knowledge, the first cost-based optimizer for MapReduce systems that (given a set of optimization objectives and constraints) can make optimization decisions ranging from finding the optimal cluster size and the type of resources to use in the cluster, to determining good workload configuration settings.

6. **SQL-tuning-aware optimization:** Employing the profile-predict-optimize approach iteratively can be used for fine-grained tuning of workload performance. We have implemented this approach within the realm of Database systems to improve any suboptimal query execution plans picked by the query optimizer for repeatedly-run SQL queries.

7. **Partition-aware optimization:** While Database systems include a cost-based query optimizer, the optimization techniques used have not kept pace with the rapid advances in usage and user control of new data layouts and partitioning strategies introduced by the MADDER principles. We address this gap by developing new techniques to generate efficient plans for queries involving multiway joins over partitioned tables.

Impact: Using the above techniques, we have designed and developed two systems for automatically tuning analytics workloads:

- *Starfish* is a MADDER and self-tuning system for Big Data analytics that employs the profile-predict-optimize approach for tuning MapReduce workloads. Starfish builds on the Hadoop MapReduce platform while adapting to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning knobs available. Starfish includes a declarative interface for specifying optimization and provisioning requests as well as a graphical user interface to simplify the user-interaction with the system. Starfish has been released publicly and has gained several external users in both academia and industry.
- *Xplus* is a novel SQL-tuning-aware query optimizer capable of executing plans proactively, collecting monitoring data from the runs, and iterating, in search for a better query execution plan. Xplus has been prototyped using PostgreSQL and has also been released publicly.

Chapter 2 presents the motivation and overview of our tuning approach, and guides the remaining chapters of this work.

2

A Tuning Approach for MADDER Systems

The problem of optimizing and automatically tuning analytical workloads executing on data-intensive systems has been the focus of a long line of commercial and research work. However, the MADDER principles call for the development of new tuning techniques in order to cope with the new challenges that arise in the era of Big Data. In this chapter, we will first review work on self-tuning Database systems. We will then discuss how the new analysis practices led us to the "profile-predict-optimize" approach for automatic tuning introduced in Section 1.5. As the usage of data-intensive workloads is growing beyond large Web companies to smaller groups with few human tuning experts, automatic tuning has become particularly timely and important.

2.1 Current Approaches to Optimization and Tuning

The *Query Optimizer* in a Database Management System (DBMS) is responsible for ensuring the fast execution of queries in the system. For each query, the optimizer will (a) consider a number of different execution plans, (b) use a cost model to predict the execution time of each plan based on some data statistics (e.g., histograms) and

configuration parameters, and (c) use the plan with the minimum predicted execution time to run the query to completion. Many query optimizers enumerate the execution plans via a dynamic programming algorithm pioneered by IBM's System R DBMS (Astrahan et al., 1976).

The rapid evolution of storage systems, increased use of user-defined functions (UDFs), and complicated data patterns resulting from the MADDER principles are causing estimates from traditional cost models to be increasingly inaccurate, leading to poorly performing execution plans (Babu et al., 2005). Even when the system is well tuned, workloads and business needs change over time; thus, the production database has to be kept in step. New optimizer statistics, configuration parameter changes, software upgrades, and hardware changes are among a large number of factors that stress the need for repeated database tuning.

2.1.1 Self-tuning Database Systems

There has been extensive work on providing database administrators (DBAs) and users the tools to tune a Database system correctly and efficiently. Database system tuning covers a broad area of research that involves problems such as performance monitoring and diagnostics infrastructures, statistics management, and automating physical database design.

Performance Monitoring and Diagnostics Infrastructures: Several tools like HP OpenView (Sheers, 1996) and IBM Tivoli (Karjoth, 2003) provide performance monitoring, whereas tools like DB2 SQL Performance Analyzer (IBM Corp., 2010) and SQL Server Performance Tuning (Agrawal et al., 2005) provide extensive analysis of SQL queries without executing them. Oracle Databases 10g and 11g contain automated tools that enable the database to monitor and diagnose itself on an ongoing basis, and alert the DBA when any problems are found (Dageville et al., 2004; Belknap et al., 2009). In particular, the *Automatic Tuning Optimizer* is a new mode

of the optimizer that is specifically used during designated maintenance sessions for generating additional information that can be used at run-time to speed performance (Belknap et al., 2009). Based on predefined rules, performance tuning is invoked by the *Automatic Diagnostic Monitor*, which is able to analyze the information in its performance data warehouse. The tools mentioned above are designed to facilitate the DBA in tuning and improving the performance of a Database system. Our goal is the same, but our approach is based on run-time monitoring information, tries to fully automate tuning, and generalizes to Dataflow systems.

Statistics Management and Execution Feedback: Query execution feedback is a technique used to improve the quality of plans by correcting cardinality estimation errors made by the query optimizer (Chen and Roussopoulos, 1994; Aboulnaga and Chaudhuri, 1999). LEO's approach (Stillger et al., 2001) extended and generalized this technique to provide a general mechanism for repairing incorrect statistics and cardinality estimates of a query execution plan. The *Pay-as-you-go* framework (Chaudhuri et al., 2008) proposed more proactive monitoring mechanisms and plan modification techniques for gathering the necessary cardinality information from a given query execution plan. Another related research direction focuses on dynamic adjustment of query plans during their execution. Kabra and DeWitt (1998) introduced a new operator to decide whether to continue or stop the execution and re-optimize the remaining plan, based on statistics collected during the query execution. *RIO* (Babu et al., 2005) proposes proactive re-optimization techniques. RIO uses intervals of uncertainty to pick execution plans that are robust to deviations of the estimated values or to defer the choice of execution plan until the uncertainty in estimates can be resolved.

Automated Physical Database Design: The efficiency by which a query is executed on a DBMS is determined by the capabilities of the execution engine and

the optimizer, as well as the *physical database design*. Automated physical database design tuning focuses on identifying the right set of index structures (Bruno and Chaudhuri, 2005; Agrawal et al., 2006), materialized views (Agrawal et al., 2000), data partitioning (Agrawal et al., 2004), and table layouts (Papadomanolakis and Ailamaki, 2004), which are crucial for efficient query execution over large databases. Furthermore, several commercial tools were created to aid the DBAs in database tuning, such as Microsoft's *Database Engine Tuning Advisor* (Agrawal et al., 2005), IBM's *DB2 Design Advisor* (Zilio et al., 2004), and Oracle's *SQL Access Advisor* (Dageville et al., 2004). Since our optimization techniques focus on improving run-time performance, they are complementary to the existing techniques for physical database design tuning.

Traditional query processing techniques based on static query optimization are ineffective in applications where statistics about the data are unavailable at the start of query execution, or where the data characteristics are skewed and change dynamically (Chaudhuri and Narasayya, 2007; Urhan et al., 1998). Even though the aforementioned adaptive query processing techniques can overcome some of the limitations of static query optimizers, they cannot handle the increasing usage of UDFs in analytical workloads or the need for analyzing unstructured and semi-structured data. Therefore, the current tuning approaches are not sufficient for dealing with all the new challenges arising in MADDER Database systems.

2.1.2 Optimizing Dataflow Systems

Being a much newer technology, MapReduce engines significantly lack principled optimization techniques compared to Database systems. Hence, the MapReduce stack (see Figure 1.2) is poorer in performance compared to a Database system running on the same amount of cluster resources (Pavlo et al., 2009). A number of ongoing efforts are addressing this issue through optimization opportunities arising

at different levels of the MapReduce stack. For higher levels of the MapReduce stack that have access to declarative semantics, many optimization opportunities inspired by database query optimization and workload tuning have been proposed. Hive and Pig employ *rule-based* approaches for a variety of optimizations such as filter and projection pushdown, shared scans of input datasets across multiple operators from the same or different analysis tasks (Nykiel et al., 2010), reducing the number of MapReduce jobs in a workflow (Lee et al., 2011), and handling data skew in sorts and joins. The *epiC* system supports System-R-style join ordering (Wu et al., 2011). Improved data layouts inspired by database storage have also been proposed (Jindal et al., 2011).

Lower levels of the MapReduce stack deal with workflows of MapReduce jobs. A MapReduce job may contain *black-box* map and reduce functions expressed in programming languages like Java, Python, and R. Many heavy users of MapReduce, ranging from large companies like Facebook and Yahoo! (Olston et al., 2008b) to small startups (Macbeth, 2011), have observed that MapReduce jobs often contain black-box UDFs to implement complex logic like statistical learning algorithms or entity extraction from unstructured data. One of the optimization techniques proposed for this level—exemplified by *HadoopToSQL* (Iu and Zwaenepoel, 2010) and *Manimal* (Cafarella and Ré, 2010)—does static code analysis of MapReduce programs to extract declarative constructs like filters and projections. These constructs are then used for database-style optimization such as projection pushdown, column-based compression, and use of indexes. Finally, the performance of MapReduce jobs is directly affected by various configuration parameter settings like degree of parallelism and use of compression. Choosing such settings for good job performance is a nontrivial problem and a heavy burden on users (Babu, 2010).

In addition, *cloud platforms* make MapReduce an attractive proposition for small organizations that need to process large datasets, but lack the computing and human

resources of a Google, Microsoft, or Yahoo! to throw at the problem. A nonexpert MapReduce user can now provision a cluster of any size on the cloud within minutes to meet her data-processing needs; and pay (only) for the nodes provisioned to the cluster for the duration of use (Amazon EMR, 2012). This feature of the cloud gives tremendous power to the average user, while placing a major burden on her shoulders. However, there has been very little work on integrating workload tuning with provisioning decisions in the context of cloud platforms.

Overall, MapReduce systems lack cost-based optimization; a feature that has been key to the historical success of Database systems. Hence, a significant portion of this work focuses on building a new cost-based optimization framework for MapReduce that is based on the collection of run-time monitoring information and fine-grained cost models. Data analysis in MapReduce exhibits many of the MADDER challenges as, typically, data resides in flat files and the jobs are specified using languages ranging from declarative to general-purpose programming ones. Finally, our approach deals with performance predictions in a complex space of workloads, data properties, cluster resources, configuration settings, and scheduling policies.

2.2 Overview of a MADDER Tuning Approach

The MADDER principles—and the consequent challenges—motivate a dynamic and automated approach for tuning data-intensive computing systems. Our solution is based on the notion of "profile-predict-optimize":

1. *Profile:* The system observes the actual run-time behavior of a workload executing on the system.

2. *Predict:* The system understands and learns from these observations, and reasons about hypothetical tuning choices.

3. *Optimize:* The system makes the appropriate optimization decisions to improve workload performance along one or more dimensions (e.g., completion time, resource utilization, pay-as-you-go monetary costs).

We have fully developed this approach for different tuning scenarios for both Dataflow and Database systems. In particular, *Starfish* employs the profile-predict-optimize approach for automatically tuning a MapReduce workload and cluster resources, after observing the run-time behavior of the workload from a single execution. A similar approach can be developed for Database systems, especially since several mechanisms are already available (see Section 2.1). Instead, we employ the profile-predict-optimize approach for the tuning scenario where the user or DBA is willing to invest some resources upfront for tuning important, repeatedly-run SQL queries. The *Xplus* optimizer profiles the execution of some query (sub)plans proactively, optimizes the plan based on the collected monitoring data, and iterates, until it finds the optimal query execution plan. We elaborate on these contributions below.

2.2.1 Tuning MapReduce Workloads with Starfish

Starfish is a MADDER and self-tuning system for analytics on Big Data (Herodotou et al., 2011d). An important design decision we made was to build Starfish on the Hadoop stack as shown in Figure 2.1. Hadoop, as observed in Chapter 1, has useful primitives to help meet the new requirements of Big Data analytics (Hadoop, 2012). In addition, Hadoop's adoption by academic, government, and industrial organizations is growing at a fast pace (Gantz and Reinsel, 2011).

A number of ongoing projects aim to improve Hadoop's peak performance, especially to match the query performance of parallel Database systems (Abouzeid et al., 2009; Dittrich et al., 2010; Jiang et al., 2010). Starfish has a different goal. The peak performance that a manually-tuned system can achieve is not our primary concern, especially if this performance is for one of the many phases in a complete

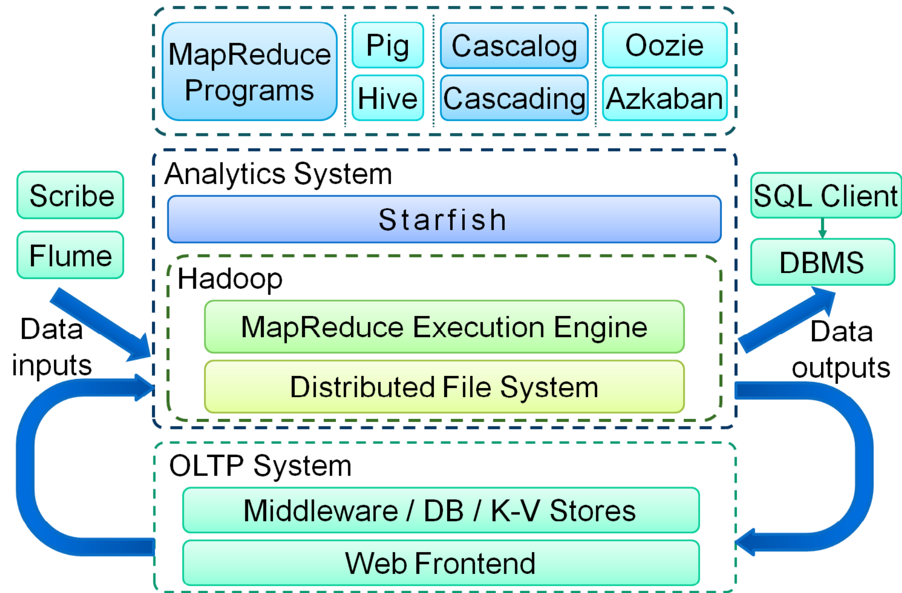


FIGURE 2.1: Starfish in the Hadoop ecosystem.

data lifecycle that includes data loading, processing ad-hoc queries, running workflows repeatedly on newly arrived data, and data archival. Starfish's goal is to enable Hadoop users and applications to get good performance automatically throughout the data lifecycle in analytics; without any need on their part to understand and manipulate the many tuning knobs available.

The *workload* that a Hadoop deployment runs can be considered at different levels. At the lowest level, Hadoop runs MapReduce *jobs*. A job can be generated directly from a program written in a programming language like Java or Python; or generated from a query in a higher-level language like HiveQL or Pig Latin; or submitted as part of a MapReduce job *workflow* (i.e., a directed acyclic graph of MapReduce jobs) by systems like Azkaban, Cascading, and Oozie (Azkaban, 2011; Cascading, 2011; Oozie, 2010). The execution plan generated for a HiveQL or Pig Latin query is usually a workflow of MapReduce jobs (Thusoo et al., 2009; Olston et al., 2008b). Workflows may be ad-hoc, time-driven (e.g., run every hour), or data-driven. Yahoo! uses data-driven workflows to generate a reconfigured preference

model and an updated home-page for any user within seven minutes of a home-page click by the user.

Hadoop itself is typically run on a large cluster built of commodity hardware. Clusters can now be easily provisioned by several cloud platforms like Amazon, Rackspace, and Skytap. *Elastic MapReduce*, for example, is a hosted service on the Amazon cloud platform where a user can instantly provision a Hadoop cluster running on any number of *Elastic Compute Cloud (EC2)* nodes (Amazon EMR, 2012). The cluster can be used to run data-intensive MapReduce jobs, and then terminated after use. The user has to pay (only) for the nodes provisioned to the cluster for the duration of use.

Suppose a user wants to execute a given MapReduce workload on a Hadoop cluster provisioned by the Amazon cloud platform. The user could have multiple preferences and constraints for the workload. For example, the goal may be to minimize the monetary cost incurred to run the workload, subject to a maximum tolerable workload execution time of two hours. In order to satisfy these requirements, the user must make a wide range of *decisions*. First, the user must decide the cluster size and the type of resources to use in the cluster from the several choices offered by the Amazon cloud platform. Next, the user must specify a large number of cluster-wide Hadoop configuration parameters like the maximum number of map and reduce tasks to execute per node and the maximum available memory per task execution. To complicate the space of decisions even further, the user has to also specify what values to use for a number of job-level Hadoop configuration parameters like the number of reduce tasks and whether to compress job outputs. Chapter 3 includes a primer on MapReduce execution and tuning, as well as multiple tuning scenarios that arise routinely in practice.

As the above scenario illustrates, users are now faced with complex *workload tuning problems* that involve determining the cluster resources as well as configuration

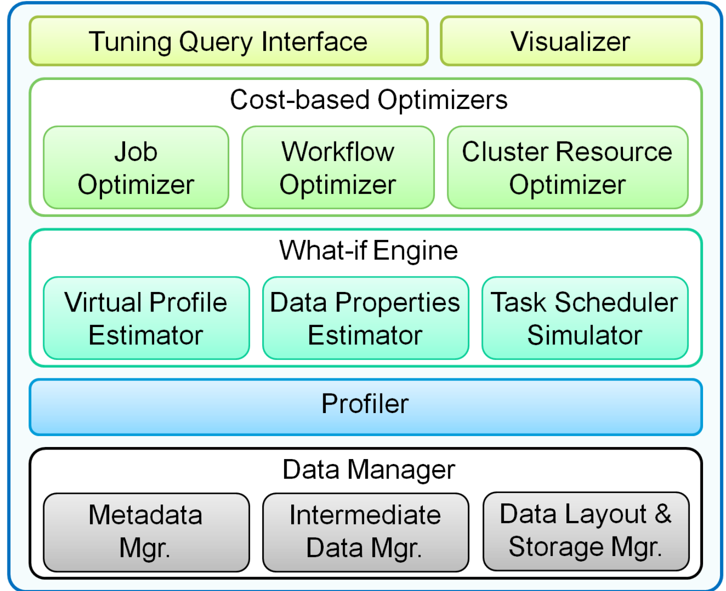


FIGURE 2.2: Components in the Starfish architecture.

settings to meet desired requirements on execution time and cost for a given analytic workload. Starfish is a novel system to which users can express their tuning problems as queries in a declarative fashion. Starfish can provide reliable answers to these queries using an automated technique, and provide nonexpert users with a good combination of cluster resource and configuration settings to meet their needs. The automated technique is based on a careful mix of profiling, estimation using black-box and white-box models, and simulation.

The Starfish architecture, shown in Figure 2.2, is motivated from the profile-predict-optimize approach. Starfish includes a *Profiler* to collect detailed statistical information from unmodified MapReduce programs, and a *What-if Engine* for fine-grained cost estimation. The capabilities of the What-if Engine are utilized by a number of *Cost-based Optimizers* that are responsible for enumerating and searching efficiently through various spaces of tuning choices, in order to find the best choices that meet the user requirements. Finally, the Tuning Query Interface and the Visualizer provide the interfaces by which users interact with the Starfish system.

Profiler: The Profiler instruments unmodified MapReduce programs dynamically to generate concise statistical summaries of MapReduce job execution, called *job profiles*. A job profile consists of dataflow and cost estimates for a MapReduce job j: dataflow estimates represent information regarding the number of bytes and key-value pairs processed during j's execution, while cost estimates represent resource usage and execution time.

The Profiler makes two important contributions. First, job profiles capture information at the fine granularity of phases within the map and reduce tasks of a MapReduce job execution. This feature is crucial to the accuracy of decisions made by the What-if Engine and the Cost-based Optimizers. Second, the Profiler uses *dynamic instrumentation* to collect run-time monitoring information from unmodified MapReduce programs. The dynamic nature means that monitoring can be turned on or off on demand; an appealing property in production deployments. By supporting unmodified MapReduce programs, we free users from any additional burden on their part to collect monitoring information. Dynamic profiling and the Profiler are discussed in detail in Chapter 4.

What-if Engine: The What-if Engine[1] is the heart of our approach to cost-based optimization and automated tuning. Apart from being invoked by the Cost-based Optimizers during program optimization, the What-if Engine can be invoked in standalone mode by users or applications to answer questions regarding the impact of configuration parameter settings, as well as data and cluster resource properties, on MapReduce workload performance.

The What-if Engine's novelty and accuracy come from how it uses a mix of simulation and model-based estimation at the phase level of MapReduce job execution.

[1] The term "what-if" also appears in the context of automated physical design in Database systems (Chaudhuri and Narasayya, 2007), where the scope of the what-if questions consists of physical design choices (e.g., indexes, materialized views) rather than tuning choices.

The What-if Engine uses a four-step process. First, a *virtual job profile* is estimated for each hypothetical job j' specified by the what-if question. The virtual profile is then used to simulate the execution of j' on the (perhaps hypothetical) cluster, as well as to estimate the data properties for the derived dataset(s) produced by j'. Finally, the answer to the what-if question is computed based on the estimated execution of j'. All performance models and components of the What-if Engine are presented in Chapter 6.

Cost-based Optimizers: For a given MapReduce workflow, input data, and cluster resources, an optimizer's role is to enumerate and search through the high-dimensional space of tuning choices efficiently, making appropriate calls to the What-if Engine, in order to find the (near) optimal choice. The space of possible tuning choices consists of (i) the subspace of cluster resource settings—which includes the number of nodes in the cluster as well as the type of each node in the cluster—and (ii) the high-dimensional subspace of configuration parameter settings—which includes parameters such as the degree of parallelism, memory settings, use of map-side and reduce-side compression, and many others.

The search space is enumerated and traversed using three Optimizers (see Figure 2.2). The *Job Optimizer* is responsible for finding good configuration settings for individual MapReduce jobs (Herodotou and Babu, 2011). The jobs in a workflow exhibit dataflow dependencies because of producer-consumer relationships as well as cluster resource dependencies because of concurrent scheduling. The *Workflow Optimizer* carefully optimizes the workflow execution within and across jobs, while accounting for these dependencies (Herodotou et al., 2012). Finally, the *Cluster Resource Optimizer* is responsible for the subspace of cluster resource settings and can help with making cluster provisioning decisions (Herodotou et al., 2011b).

The number of calls to the What-if Engine has to be minimized for efficiency,

without sacrificing the ability to find good tuning settings. Towards this end, all optimizers divide the full space of tuning choices into lower-dimensional subspaces such that the globally-optimal choices in the high-dimensional space can be generated by composing the optimal choices found for the subspaces. The overall cost-based optimization approach is discussed in Chapter 7.

Tuning Query Interface and Visualizer: A general tuning problem involves determining the cluster resources and MapReduce job-level configuration settings to meet desired performance requirements on execution time and cost for a given analytic workload. Starfish provides a declarative interface to express a range of tuning problems as queries in a declarative fashion. A query expressed using this interface will specify (i) the MapReduce workload, (ii) the search space for cluster resources, (iii) the search space for job configurations, and (iv) the performance requirements in terms of time and monetary cost. Applications and users can also interact with this interface using a programmatic API, or using a graphical interface that forms part of the Starfish system's Visualizer (Herodotou et al., 2011a). The Tuning Query Interface and the Starfish Visualizer are presented in Chapter 5.

2.2.2 Tuning SQL Queries with Xplus

The profile-predict-optimize approach used currently in Starfish is just one cycle of a more general self-tuning approach that learns repeatedly over time and re-optimizes as needed. In this spirit, we propose experiment-driven tuning of important, repeatedly-run SQL queries in Database systems. The need to improve a suboptimal execution plan picked by the query optimizer for a repeatedly-run SQL query (e.g., by a business intelligence or report generation application) arises routinely in MADDER settings. Unknown or stale statistics, complex expressions, and changing conditions can cause the optimizer to make mistakes. In Chapter 8, we present a novel SQL-tuning-aware query optimizer, called Xplus (Herodotou and

Babu, 2010), that is capable of executing plans proactively, collecting monitoring data from the runs, and iterating, in search for a better execution plan.

Finally, despite the recent advances in query optimization techniques, Database and Dataflow systems still struggle with the decreased control over data storage and data structure mandated by the MADDER principles. Careful data layouts and partitioning strategies are powerful mechanisms for improving query performance and system manageability in these systems. SQL extensions and MapReduce frameworks now enable applications and user queries to specify how their results should be partitioned for further use, decreasing the control that database administrators had previously over partitioning. However, query optimization techniques have not kept pace with the rapid advances in usage and user control of table partitioning. We address this gap by developing new techniques to generate efficient plans for SQL queries involving multiway joins over partitioned tables (Herodotou et al., 2011c). These techniques are presented in Chapter 9.

3

Primer on Tuning MapReduce Workloads

MapReduce is a relatively young framework—both a programming model and an associated run-time system—for large-scale data processing (Dean and Ghemawat, 2008). *Hadoop* is the most popular open-source implementation of a MapReduce framework that follows the design laid out in the original paper (Dean and Ghemawat, 2004). A number of companies use Hadoop in production deployments for applications such as Web indexing, data mining, report generation, log file analysis, machine learning, financial analysis, scientific simulation, and bioinformatics research. Infrastructure-as-a-Service cloud platforms like Amazon and Rackspace have made it easier than ever to run Hadoop workloads by allowing users to instantly provision clusters and pay only for the time and resources used. A combination of features contributes to Hadoop's increasing popularity, including fault tolerance, data-local scheduling, ability to operate in a heterogeneous environment, handling of straggler tasks[1], as well as a modular and customizable architecture.

In this chapter, we provide an overview of the MapReduce programming model and describe how MapReduce programs execute on a Hadoop cluster. The behavior

[1] A straggler is a task that performs poorly typically due to faulty hardware or misconfiguration.

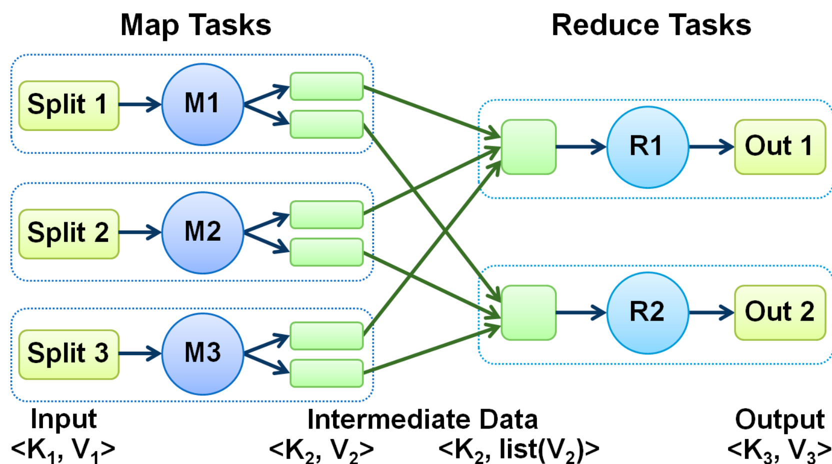


FIGURE 3.1: Execution of a MapReduce job.

of MapReduce job execution is affected by a large number of configuration parameter settings. We will provide empirical evidence of the significant impact that parameter settings can have on the performance of a MapReduce job. Finally, we will list various optimization and tuning scenarios that arise routinely in practice.

3.1 MapReduce Job Execution

The MapReduce programming model consists of two functions: $map(k_1, v_1)$ and $reduce(k_2, list(v_2))$. Users can implement their own processing logic by specifying a customized $map()$ and $reduce()$ function written in a general-purpose language like Java or Python. The $map(k_1, v_1)$ function is invoked for every *key-value* pair $\langle k_1, v_1 \rangle$ in the input data to output zero or more key-value pairs of the form $\langle k_2, v_2 \rangle$ (see Figure 3.1). The $reduce(k_2, list(v_2))$ function is invoked for every unique key k_2 and corresponding values $list(v_2)$ in the map output. $reduce(k_2, list(v_2))$ outputs zero or more key-value pairs of the form $\langle k_3, v_3 \rangle$. The MapReduce programming model also allows other functions such as (i) $partition(k_2)$, for controlling how the map output key-value pairs are partitioned among the reduce tasks, and (ii) $combine(k_2, list(v_2))$, for performing partial aggregation on the map side. The keys k_1, k_2, and k_3 as well

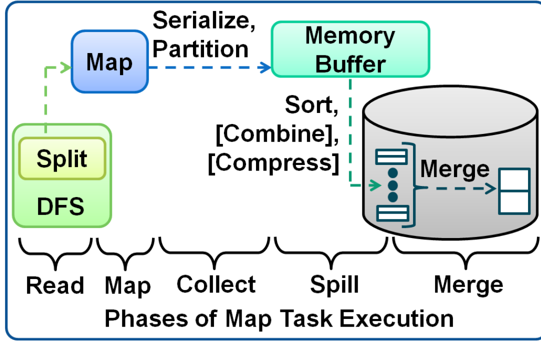


FIGURE 3.2: Execution of a map task showing the map-side phases.

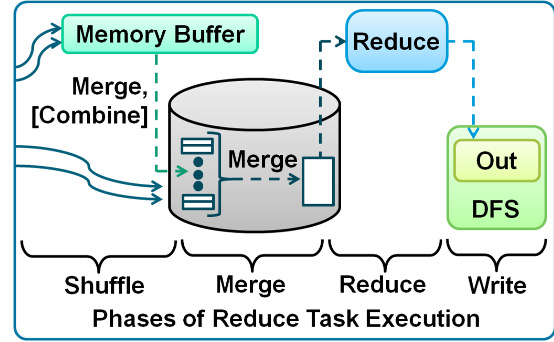


FIGURE 3.3: Execution of a reduce task showing the reduce-side phases.

as the values v_1, v_2, and v_3 can be of different and arbitrary types.

A Hadoop MapReduce cluster employs a master-slave architecture where one master node (called *JobTracker*) manages a number of slave nodes (called *TaskTrackers*). Figure 3.1 shows how a MapReduce job is executed on the cluster. Hadoop launches a MapReduce job by first splitting (logically) the input dataset into data *splits*. Each data split is then scheduled to one TaskTracker node and is processed by a map task. A *Task Scheduler* is responsible for scheduling the execution of map tasks while taking data locality into account. Each TaskTracker has a predefined number of task execution *slots* for running map (reduce) tasks. If the job will execute more map (reduce) tasks than there are slots, then the map (reduce) tasks will run in multiple *waves*. When map tasks complete, the run-time system groups all intermediate key-value pairs using an external sort-merge algorithm. The intermediate data is then *shuffled* (i.e., transferred) to the TaskTrackers scheduled to run the reduce tasks. Finally, the reduce tasks will process the intermediate data to produce the results of the job.

The MapReduce job execution can be decomposed further into *phases* within map and reduce tasks. As illustrated in Figure 3.2, map task execution consists of the following phases: *Read* (reading map inputs), *Map* (map function processing), *Collect*

(partitioning and buffering map outputs before spilling), *Spill* (sorting, combining, compressing, and writing map outputs to local disk), and *Merge* (merging sorted spill files). As illustrated in Figure 3.3, reduce task execution consists of the following phases: *Shuffle* (transferring map outputs to reduce tasks, with decompression if needed), *Merge* (merging sorted map outputs), *Reduce* (reduce function processing), and *Write* (writing reduce outputs to the distributed file-system). Additionally, both map and reduce tasks have *Setup* and *Cleanup* phases.

A MapReduce workload consists of MapReduce jobs of the form $j = \langle p, d, r, c \rangle$. Here, p represents the MapReduce program that is run as part of j to process input data d on cluster resources r using configuration parameter settings c.

Program: A given MapReduce program p may be expressed in one among a variety of programming languages like Java, C++, Python, or Ruby; and then connected to form a workflow using a workflow scheduler like Oozie (Oozie, 2010). Alternatively, the MapReduce jobs can be generated automatically using compilers for higher-level languages like Pig Latin (Olston et al., 2008b), HiveQL (Thusoo et al., 2009), and Cascading (Cascading, 2011).

Data: The properties of the input data d processed by a MapReduce job j include d's size, the block layout of files that comprise d in the distributed file-system, and whether d is stored compressed or not. Since the MapReduce methodology is to interpret data (lazily) at processing time, and not (eagerly) at loading time, other properties such as the schema and data-level distributions of d are unavailable by default.

Cluster resources: The properties of the cluster resources r that are available for a job execution include the number of nodes in the cluster, the machine specifications (or the node type when the cluster is provisioned by a cloud platform like Amazon EC2), the cluster's network topology, the number of map and reduce task execution

slots per node, and the maximum memory available per task execution slot.

Configuration parameter settings: A number of choices have to be made in order to fully specify how the job should execute. These choices, represented by c in $\langle p, d, r, c \rangle$, come from a large and high-dimensional space of configuration parameter settings that includes (but is not limited to):

1. The number of map tasks in job j. Each task processes one partition (*split*) of the input data d. These tasks may run in multiple waves depending on the total number of map task execution slots in r.

2. The number of reduce tasks in j (which may also run in waves).

3. The amount of memory to allocate to each map (reduce) task to buffer its output (input) data.

4. The settings for the multiphase external sorting used by most MapReduce frameworks to group map-output values by key.

5. Whether the output data from the map (reduce) tasks should be compressed before being written to disk (and if so, then how).

6. Whether the program-specified combine function should be used to preaggregate map outputs before their transfer to reduce tasks.

Hadoop has more than 190 configuration parameters out of which Starfish currently considers 14 parameters whose settings can have significant impact on job performance (Herodotou et al., 2011d). These parameters are listed on Table 3.1. If the user does not specify parameter settings during job submission, then default values—shipped with the system or specified by the system administrator—are used. Good settings for these parameters depend on job, data, and cluster characteristics. While

Table 3.1: A subset of important job configuration parameters in Hadoop.

Parameter Name	Brief Description and Use	Default Value
io.sort.mb	Size (in MB) of map-side buffer for storing and sorting key-value pairs produced by the map function	100
io.sort.record.percent	Fraction of io.sort.mb dedicated to metadata storage for every key-value pair stored in the map-side buffer	0.05
io.sort.spill.percent	Usage threshold of map-side memory buffer to trigger a sort and spill of the stored key-value pairs	0.8
io.sort.factor	Number of sorted streams to merge at once during the multiphase external sorting	10
mapreduce.combine .class	The (optional) combine function to preaggregate map outputs before transferring to the reduce tasks	null
min.num.spills.for.combine	Minimum number of spill files at which to use the combine function during the merging of map output data	3
mapred.compress.map .output	Boolean flag to turn on the compression of map output data	false
mapred.reduce.slowstart .completed.maps	Proportion of map tasks that need to be completed before any reduce tasks are scheduled	0.05
mapred.reduce.tasks	Number of reduce tasks	1
mapred.job.shuffle.input .buffer.percent	% of reduce task's heap memory used to buffer output data copied from map tasks during the shuffle	0.7
mapred.job.shuffle .merge.percent	Usage threshold of reduce-side memory buffer to trigger reduce-side merging during the shuffle	0.66
mapred.inmem.merge .threshold	Threshold on the number of copied map outputs to trigger reduce-side merging during the shuffle	1000
mapred.job.reduce.input .buffer.percent	% of reduce task's heap memory used to buffer map output data while applying the reduce function	0
mapred.output.compress	Boolean flag to turn on the compression of the job's output	false

only a fraction of the parameters can have significant performance impact, browsing through the Hadoop, Hive, and Pig mailing lists reveals that users often run into performance problems caused by lack of knowledge of these parameters.

MapReduce workflow: A MapReduce workflow W is a *directed acyclic graph (DAG)* G_W that represents a set of MapReduce jobs and their dataflow dependencies. Each vertex in G_W is either a MapReduce job j or a dataset d. An edge in G_W can only exist between a job (vertex) j and a dataset (vertex) d. A directed edge ($d \rightarrow j$) denotes d as an input dataset of job j; and a directed edge ($j \rightarrow d$) denotes

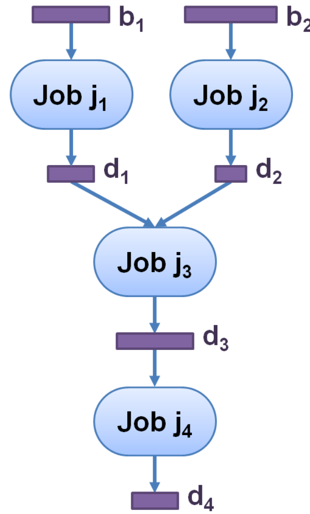


FIGURE 3.4: An example MapReduce workflow with four MapReduce jobs (j_1-j_4), two base datasets (b_1, b_2), and four derived datasets (d_1-d_4).

d as an output dataset of j. The datasets processed by a given workflow W are categorized into *base* and *derived* datasets. Base datasets (*base*(W)) represent the existing data consumed by W, whereas derived datasets (*derived*(W)) represent the data generated by the MapReduce programs in W.

Figure 3.4 shows an example workflow with four MapReduce jobs (j_1-j_4), two base datasets (b_1, b_2), and four derived datasets (d_1-d_4). The distinction between base and derived data will become important in Chapter 6 where we discuss how we can answer hypothetical questions regarding the execution of a MapReduce workflow.

Abstractions in Starfish: In order to support the wide and growing variety of MapReduce programs and the programming languages in which they are expressed, Starfish represents the execution of a MapReduce job j using a *job profile*. This profile is a concise summary of the dataflow and cost information for job j's execution Similar to a job profile, a *workflow profile* is used to represent the execution of a MapReduce workflow W on the cluster. Chapter 4 discusses the content of the job and workflow profiles, as well as how these profiles are generated.

3.2 Impact of Configuration Parameter Settings

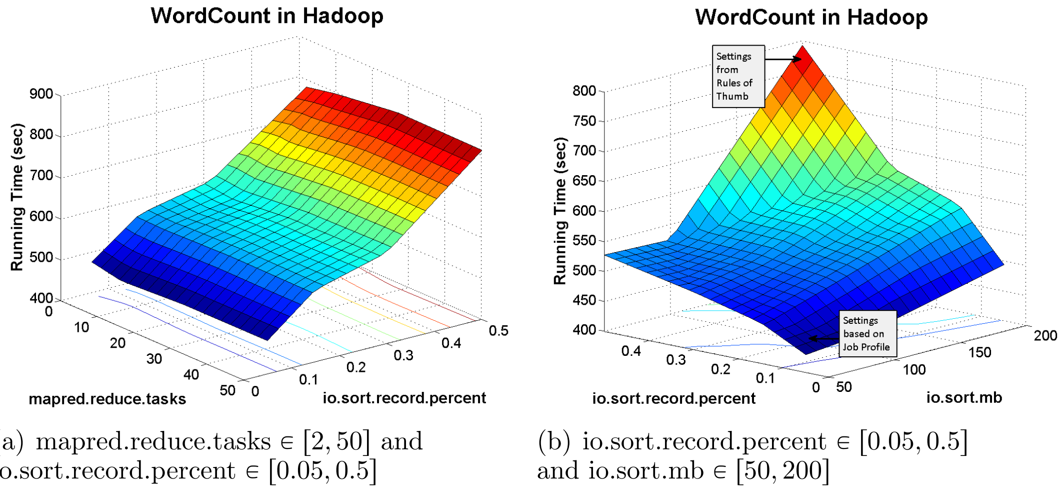
The Hadoop configuration parameters control various aspects of job behavior during execution, such as memory allocation and usage, concurrency, I/O optimization, and network bandwidth usage. To illustrate the impact of job configuration parameters in Hadoop, we study the effects of several parameter settings on the performance of two MapReduce programs. The experimental setup used is a single-rack Hadoop cluster running on 16 nodes, with 1 master and 15 slave nodes, provisioned from Amazon Elastic Compute Cloud (EC2). Each node has 1.7 GB of memory, 5 EC2 compute units, 350 GB of storage, and is set to run at most 3 map tasks and 2 reduce tasks concurrently. Thus, the cluster can run at most 45 map tasks in a concurrent *map wave*, and at most 30 reduce tasks in a concurrent *reduce wave*. Table 3.1 lists the subset of job configuration parameters that we considered in our experiments.

The MapReduce jobs we consider are WordCount and TeraSort[2]; two simple, yet representative, text processing jobs with well-understood characteristics. WordCount processes 30GB of data generated using Hadoop's RandomTextWriter, while TeraSort processes 50GB of data generated using Hadoop's TeraGen. Figures 3.5 and 3.6 show the response surfaces that were generated by measuring the execution time of the WordCount and TeraSort jobs respectively. The three parameters varied in these figures are *io.sort.mb*, *io.sort.record.percent*, and *mapred.reduce.tasks*, while all other job configuration parameters are kept constant.

The effects of the parameter settings on the performance of a MapReduce job depend on job, data, and cluster characteristics:

Effect of job characteristics: Figures 3.5(a) and 3.6(a) show how the setting of the *mapred.reduce.tasks* parameter (i.e., the number of reducers) affect WordCount and TeraSort in different ways. Increasing the number of reducers has no impact on

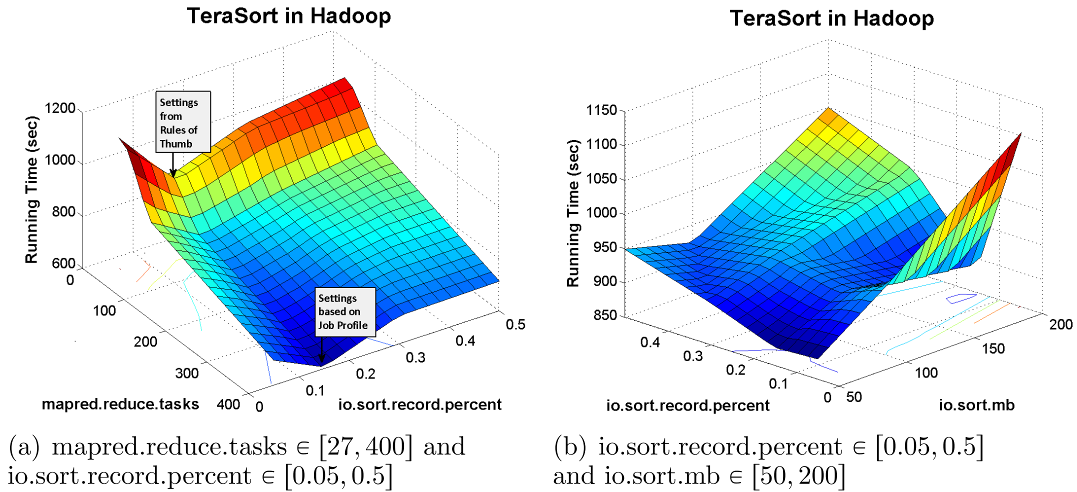
[2] TeraSort was used on a Hadoop cluster at Yahoo! to win the TeraByte Sort Benchmark in 2008.



(a) mapred.reduce.tasks $\in [2, 50]$ and io.sort.record.percent $\in [0.05, 0.5]$

(b) io.sort.record.percent $\in [0.05, 0.5]$ and io.sort.mb $\in [50, 200]$

FIGURE 3.5: Response surfaces of WordCount MapReduce jobs in Hadoop.



(a) mapred.reduce.tasks $\in [27, 400]$ and io.sort.record.percent $\in [0.05, 0.5]$

(b) io.sort.record.percent $\in [0.05, 0.5]$ and io.sort.mb $\in [50, 200]$

FIGURE 3.6: Response surfaces of TeraSort MapReduce jobs in Hadoop.

performance for WordCount across all settings of *io.sort.record.percent*, whereas it improves the performance for TeraSort significantly. The job execution of WordCount is dominated by the execution of the map tasks. Computation in the map tasks includes the parsing of the input files, as well as applying a combine function, whereas the reduce tasks are simply summing the word counts.

On the other hand, the map and reduce functions in TeraSort process the same

amount of data overall and perform the same task of simply writing all input values directly to output. Increasing the number of reduce tasks improves performance due to: (i) the increase in effective concurrency by utilizing more of the reduce slots in the cluster (recall that our cluster has 30 reducer slots across 15 worker nodes), and (ii) the processing of less data by each reduce task, since the overall data size processed is fixed (which in turn can reduce I/O in nonlinear ways).

Effect of data characteristics: Data characteristics—like number of unique key values, key-value distributions, as well as input and intermediate record sizes—can effect the running time of job executions of the same MapReduce program, running with the same parameter settings. The surface area of Figure 3.6(a) contains a "valley" when the value for *io.sort.record.percent* is set to 0.15. *io.sort.record.percent* represents the fraction of the map's heap size that is dedicated to metadata storage for the map's output. Each record produced by the mapper requires 16 bytes of metadata in addition to its serialized size. Given any value for this parameter, the average map output record size will determine whether a spill to disk is caused by exhaustion of the serialization buffer or by exhaustion of the metadata buffer.

Based on the current data characteristics, setting *io.sort.record.percent* to 0.15 maximizes the use of both buffers, leading to good job performance (see Figure 3.6(a)). Suppose we were to run TeraSort with the same parameter settings on a new data set, where the average input record size is half the current size. Then, the metadata buffer would become full when only half the serialization buffer is full, causing a larger number of spills than necessary; thereby increasing the execution time of the map tasks and the job. In other words, the valley in the surface area of Figure 3.6(a) would shift to a larger value for *io.sort.record.percent* compared to the current surface area.

Effect of cluster characteristics: The number of nodes in a cluster, the number

of map and reduce slots per node, the memory available for each task execution, and the network setup are the prime cluster characteristics that can affect the impact of parameter settings in job performance. For example, the number of reduce slots determines the effective concurrency of the reduce computations. When the total number of reduce tasks T is lower than the number of slots S, all reduce tasks will run concurrently. Changing the number of reducers while $T \leqslant S$ will have a significant effect on job performance, assuming the reduce task's execution time is comparable to the map task's execution time. When $T > S$, the reducers will run in waves. In Figure 3.6(a), we observe that as the number of reduce tasks increases, the performance improves but the *rate* of improvement decreases because of the bound on effective concurrency per wave, as well as task setup overheads.

Interaction among parameters: A fairly large subset of the configuration parameters in Hadoop display strong performance *interactions* with one or more other parameters. An interaction exists between parameters p_1 and p_2 when the magnitude of impact that varying p_1 has on job performance depends on the specific setting of p_2. Stated otherwise, the performance impact of varying p_1 is different across different settings of p_2. For example, Figure 3.5(b) shows that for low settings of *io.sort.record.percent*, the job performance is not affected significantly by varying *io.sort.mb*. However, for high settings of *io.sort.record.percent*, the performance changes drastically while varying *io.sort.mb*. Figure 3.6(b) shows stronger and more complicated interactions between *io.sort.record.percent* and *io.sort.mb*. Across different values of *io.sort.record.percent*, even the pattern of change in performance is different as *io.sort.mb* is varied.

3.3 MapReduce on the Cloud

Infrastructure-as-a-Service (IaaS) cloud platforms provide computation, software, data access, and storage resources to a number of users, while the details of the underlying infrastructure are completely transparent. This computing paradigm is attracting increasing interest from both academic researchers and industry data practitioners because it enables MapReduce users to scale their applications up and down seamlessly in a pay-as-you-go manner. *Elastic MapReduce*, for example, is a hosted service on the Amazon cloud platform where a user can instantly provision a Hadoop cluster running on any number of *Elastic Compute Cloud (EC2)* nodes (Amazon EMR, 2012). The cluster can be used to run data-intensive MapReduce jobs, and then terminated after use. The user has to pay (only) for the nodes provisioned to the cluster for the duration of use.

The new and remarkable aspect here is that a nonexpert MapReduce user can provision a cluster of any size on the cloud within minutes to meet her data-processing needs. This feature of the cloud gives tremendous power to the average user, while placing a major burden on her shoulders. Previously, the same user would have had to work with system administrators and management personnel to get a cluster provisioned for her needs. Many days to months would have been needed to complete the provisioning process. Furthermore, making changes to an already-provisioned cluster was a hassle.

Cloud platforms make cluster provisioning almost instantaneous. The elastic and pay-as-you-go nature of these platforms means that, depending on how best to meet her needs, a user can allocate a 10-node cluster today, a 100-node cluster tomorrow, and a 25-node cluster the day after. However, removing the system administrator and the traditional capacity-planning process from the loop shifts the nontrivial responsibility of determining a good cluster configuration to the nonexpert user.

Table 3.2: Five representative Amazon EC2 node types, along with resources and monetary costs.

EC2 Node Type	CPU (# EC2 Units)	Memory (GB)	Storage (GB)	I/O Performance	Cost (U.S. $ per hour)
m1.small	1	1.7	160	moderate	0.085
m1.large	4	7.5	850	high	0.34
m1.xlarge	8	15	1,690	high	0.68
c1.medium	5	1.7	350	moderate	0.17
c1.xlarge	20	7	1,690	high	0.68

As an illustrative example, consider provisioning a Hadoop cluster on Amazon EC2 nodes to run a MapReduce workload on the cloud. Services like Elastic MapReduce and Hadoop On Demand free the user from having to install and maintain the Hadoop cluster. However, the burden of cluster provisioning is still on the user, who is likely not an expert system administrator. In particular, the user has to specify the number of EC2 nodes to use in the cluster, as well as the *node type* to use from among 10+ EC2 node types. Table 3.2 shows the features and renting costs of some representative EC2 node types. Notice that the CPU and I/O resources available on these node types are quoted in abstract terms that an average user will have trouble understanding. To complicate the space of choices even further, the user has to specify what values to use for a number of configuration parameters—e.g., the number of reduce tasks or whether to compress map outputs—at the level of MapReduce job execution on the cluster (Babu, 2010; Herodotou and Babu, 2011).

3.4 Use Cases for Tuning MapReduce Workloads

In this work, we refer to the general problem of determining the cluster resources and MapReduce job-level configuration parameter settings to meet desired requirements on execution time and cost for a given analytic workload as the *tuning problem.* Users can express tuning problems as declarative queries to Starfish, for which Starfish will provide reliable answers in an automated fashion. In order to illustrate how Starfish

benefits users and applications, we begin by discussing some common scenarios where tuning problems arise.

1. Tuning job-level configuration parameter settings: Even to run a single job in a MapReduce framework, a number of configuration parameters have to be set by users or system administrators. Users often run into performance problems because they do not know how to set these parameters, or because they do not even know these parameters exist. In other cases, the performance of a MapReduce job or workflow simply does not meet the *Service Level Objectives (SLOs)* on response time or workload completion time. Hence, the need for understanding the job behavior as well as diagnosing bottlenecks during job execution for the parameter settings used arises frequently. Even when users understand how a program behaved during a specific run, they still cannot predict how the execution of the program will be affected when parameter settings change, or which parameters should they use to improve performance. Nonexpert users can now employ Starfish to (a) get a deep understanding of a MapReduce program's behavior during execution, (b) ask hypothetical questions on how the program's behavior will change when parameter settings, cluster resources, or input data properties change, and (c) ultimately optimize the program.

2. Tuning the cluster size for elastic workloads: Suppose a MapReduce job takes three hours to finish on a 10-node Hadoop cluster of EC2 nodes of the m1.large type. The application or the user who controls the cluster may want to know by how much the execution time of the job will reduce if five more m1.large nodes are added to the cluster. Alternatively, the user might want to know how many m1.large nodes must be added to the cluster to reduce the running time down to two hours. Such questions also arise routinely in practice, and can be answered automatically by Starfish.

3. Planning for workload transition from a development cluster to production: Most enterprises maintain separate (and possibly multiple) clusters for application development compared to the production clusters used for running mission-critical and time-sensitive workloads. Elasticity and pay-as-you-go features have simplified the task of maintaining multiple clusters. For example, Facebook uses a *Platinum* cluster that only runs mission-critical jobs (Bodkin, 2010). Less critical jobs are run on a separate *Gold* or a *Silver* cluster where data from the production cluster is replicated.

A developer will first test a new MapReduce job on the development cluster, possibly using small representative samples of the data from the production cluster. Before the job can be scheduled on the production cluster—usually as part of an analytic workload that is run periodically on new data—the developer will need to identify a MapReduce job-level configuration that will give good performance for the job when run on the production cluster (without actually running the job on this cluster). Starfish helps the developer with this task. Based on how the job performs when run on the development cluster, Starfish can estimate how the job will run under various hypothetical configurations on the production cluster; and recommend a good configuration to use.

4. Cluster provisioning under multiple objectives: Infrastructure-as-a-Service (IaaS) cloud platforms like Amazon EC2 and Rackspace offer multiple choices for the type of node to use in a cluster (see Table 3.2). As the levels of compute, memory, and I/O resources on the nodes increase, so does the cost of renting the nodes per hour. Figure 3.7 shows the execution time as well as total cost incurred for a MapReduce workload running on Hadoop under different cluster configurations on EC2. The clusters in Figures 3.7(a) and (b) use six nodes each of the EC2 node type shown, with a fixed per-hour renting cost, denoted *cost_ph* (shown in Table 3.2). The *pricing*

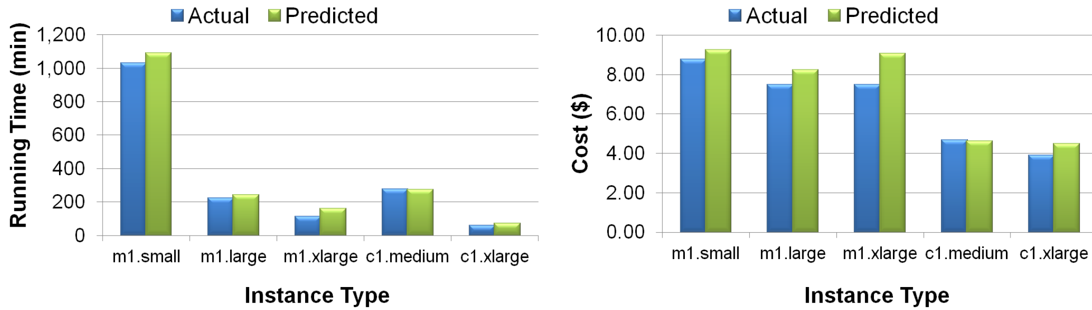


FIGURE 3.7: Performance Vs. pay-as-you-go costs for a workload that is run on different EC2 cluster resource configurations.

model used to compute the corresponding total cost of each workload execution is:

$$total_cost = cost_ph \times num_nodes \times exec_time \tag{3.1}$$

Here, *num_nodes* is the number of nodes in the cluster and *exec_time* is the execution time of the workload rounded up to the nearest hour as done on most cloud platforms. The user could have multiple preferences and constraints for the workload. For example, the goal may be to minimize the monetary cost incurred to run the workload, subject to a maximum tolerable workload execution time. Based on Figure 3.7, if the user wants to minimize cost subject to an execution time of under 45 minutes, then Starfish should recommend a cluster of six c1.xlarge EC2 nodes.

Notice from Figures 3.7(a) and (b) that Starfish is able to capture the execution trends of the workload correctly across the different clusters. Some interesting tradeoffs between execution time and cost can also be seen in Figure 3.7. For example, the cluster of six m1.xlarge nodes runs the workload almost 2x faster than the cluster of six c1.medium nodes; but at 1.7x times the cost.

5. Shifting workloads in time to lower execution costs: The pricing model from Equation 3.1 that was used to compute costs in Figure 3.7(b) charges a flat per-hour price based on the node type used. Such nodes are called *on-demand*

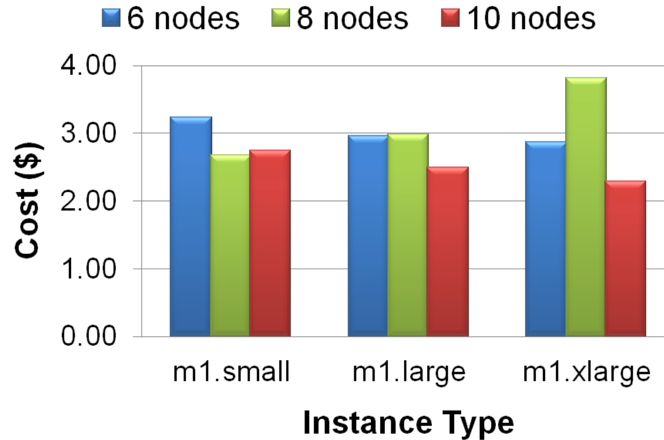


FIGURE 3.8: Pay-as-you-go costs for a workload from Figure 3.7 when run using auction-based EC2 spot instances.

instances on EC2. Amazon EC2 also offers *spot instances* whose prices can vary with time, usually based on the supply and demand for resources on the cloud (Chohan et al., 2010; Hamilton, 2008). Other factors such as temporal and spatial variations in electricity prices can also cause resource usage costs to fluctuate on the cloud (Qureshi et al., 2009).

The vast majority of analytics workloads can tolerate some slack in completion time. For example, data analysts in a company may find it sufficient to have an overnight report-generation workload complete before the company's U.S. East Coast offices reopen for business. (Such workloads are called "batch and non-user synchronous workloads" in Hamilton (2008).) This observation gives rise to an online scheduling problem where the slack is exploited to run a given workload when, ideally, resource prices are the cheapest. Solving this online scheduling problem is beyond the scope of this work since our focus is on the tuning problem. However, Starfish is indispensable in any solution to the scheduling problem since the solution would need estimates of workload execution time and cost for both on-demand and spot instances in various cluster configurations.

As an illustration, Figure 3.8 shows the total cost incurred for the same MapRe-

duce workload from Figure 3.7 when nodes of the EC2 spot instance type shown were used to run the workload around 6.00 AM Eastern Time. The pricing model used to compute the total cost in this case is:

$$total_cost = \sum_{i=0}^{num_hours} cost_ph^{(i)} \times num_nodes \tag{3.2}$$

The summation here is over the number of hours of execution, with $cost_ph^{(i)}$ representing the price charged per node type used in the cluster for the ith hour. By comparing Figure 3.7(b) with Figure 3.8, it is clear that execution costs for the same workload can differ significantly across different choices for the cluster resources used.

4

Dynamic Profiling of MapReduce Workloads

The high-level goal of dynamic profiling in Starfish is to collect run-time monitoring information efficiently during the execution of a MapReduce job (recall Section 2.2.1). Starfish uses dynamic profiling to build a *job profile*, which is a concise representation of the job's execution. There are some key challenges that Starfish needs to address with respect to *what*, *how*, and *when* to profile.

First, we need to collect both dataflow and cost information during job execution. Little knowledge about the input data may be available before the job is submitted. Keys and values are often extracted dynamically from the input data by the map function, so schema and statistics about the data may be unknown. In addition, map and reduce functions are usually written in programming languages like Java, Python, and C++ that are not restrictive or declarative like SQL. Hence, we need to carefully observe the run-time cost of these functions, as well as how they process the data.

Collecting fine-grained monitoring information efficiently and noninvasively is another key challenge. We have chosen to use *dynamic instrumentation* to collect this information from unmodified MapReduce programs. The dynamic nature means

that monitoring can be turned on or off on demand; an appealing property in production deployments. Dynamic instrumentation has the added advantage that no changes are needed to the MapReduce system's source code (the Hadoop system in our specific case). As a result, we leverage all past investments as well as potential future enhancements to the MapReduce system.

Finally, the dynamic nature of profiling creates a need for investing some resources upfront—i.e., before the actual job execution starts—to collect the necessary information. However, many MapReduce programs are written once and run many times over their lifetime (usually on different datasets). Programs for Extract-Transform-Load (ETL) and report generation are good examples. Properties of such programs as well as good configuration settings for them can be learned over time in the spirit of learning optimizers like Leo (Stillger et al., 2001). For ad-hoc MapReduce programs, we employ sampling techniques for collecting approximate profiling information quickly.

4.1 Job and Workflow Profiles

A MapReduce *job profile* is a vector of fields where each field captures some unique aspect of dataflow or cost during a MapReduce job execution at the task level or the phase[1] level within tasks. Including information at the fine granularity of phases within tasks is crucial to the accuracy of decisions made by the What-if Engine and the Cost-based Optimizers. We partition the fields in a profile into four categories, described next. The rationale for this categorization will become clear in Chapter 6 when we describe how a virtual profile is estimated for a hypothetical job (without actually running the job).

[1] Recall the phases of MapReduce job execution discussed in Section 3.1

Table 4.1: Dataflow fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields, unless otherwise stated, represent information at the level of tasks)	Depends On		
		d	r	c
dNumMappers	Number of map tasks in the job	✓		✓
dNumReducers	Number of reduce tasks in the job			✓
dMapInRecs	Map input records	✓		✓
dMapInBytes	Map input bytes	✓		✓
dMapOutRecs	Map output records	✓		✓
dMapOutBytes	Map output bytes	✓		✓
dNumSpills	Number of spills	✓		✓
dSpillBufferRecs	Number of records in buffer per spill	✓		✓
dSpillBufferSize	Total size of records in buffer per spill	✓		✓
dSpillFileRecs	Number of records in spill file	✓		✓
dSpillFileSize	Size of a spill file	✓		✓
dNumRecsSpilled	Total spilled records	✓		✓
dNumMergePasses	Number of merge rounds	✓		✓
dShuffleSize	Total shuffle size	✓		✓
dReduceInGroups	Reduce input groups (unique keys)	✓		✓
dReduceInRecs	Reduce input records	✓		✓
dReduceInBytes	Reduce input bytes	✓		✓
dReduceOutRecs	Reduce output records	✓		✓
dReduceOutBytes	Reduce output bytes	✓		✓
dCombineInRecs	Combine input records	✓		✓
dCombineOutRecs	Combine output records	✓		✓
dLocalBytesRead	Bytes read from local file system	✓		✓
dLocalBytesWritten	Bytes written to local file system	✓		✓
dHdfsBytesRead	Bytes read from HDFS	✓		✓
dHdfsBytesWritten	Bytes written to HDFS	✓		✓

- *Dataflow fields* capture information about the amount of data, both in terms of bytes as well as records (key-value pairs), flowing through the different tasks and phases of a MapReduce job execution. Some example fields are the number of map output records and the amount of bytes shuffled among the map and reduce tasks. Table 4.1 lists all the dataflow fields in a profile.

- *Cost fields* capture information about execution time at the level of tasks and phases within the tasks for a MapReduce job execution. Some example fields

Table 4.2: Cost fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields represent information at the level of tasks)	Depends On		
		d	r	c
cSetupPhaseTime	Setup phase time in a task	✓	✓	✓
cCleanupPhaseTime	Cleanup phase time in a task	✓	✓	✓
cReadPhaseTime	Read phase time in the map task	✓	✓	✓
cMapPhaseTime	Map phase time in the map task	✓	✓	✓
cCollectPhaseTime	Collect phase time in the map task	✓	✓	✓
cSpillPhaseTime	Spill phase time in the map task	✓	✓	✓
cMergePhaseTime	Merge phase time in map/reduce task	✓	✓	✓
cShufflePhaseTime	Shuffle phase time in the reduce task	✓	✓	✓
cReducePhaseTime	Reduce phase time in the reduce task	✓	✓	✓
cWritePhaseTime	Write phase time in the reduce task	✓	✓	✓

are the execution time of the Collect and Spill phases of a map task. Table 4.2 lists all the cost fields in a profile.

- *Dataflow Statistics fields* capture statistical information about the dataflow—e.g., the average number of records output by map tasks per input record (the Map selectivity) or the compression ratio of the map output—that is expected to remain unchanged across different executions of the MapReduce job unless the data distribution in the input dataset changes significantly across these executions. Table 4.3 lists all the dataflow statistics fields in a profile.

- *Cost Statistics fields* capture statistical information about execution time for a MapReduce job—e.g., the average time to read a record from the distributed file-system, or the average time to execute the map function per input record—that is expected to remain unchanged across different executions of the job unless the cluster resources (e.g., CPU, I/O) available per node change. Table 4.4 lists all the cost statistics fields in a profile.

Intuitively, the Dataflow and Cost fields in the profile of a job j help in understanding j's behavior. On the other hand, the Dataflow Statistics and Cost Statistics fields

Table 4.3: Dataflow statistics fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields represent information at the level of tasks)	Depends On		
		d	r	c
dsInputPairWidth	Width of input key-value pairs	✓		
dsRecsPerRedGroup	Number of records per reducer's group	✓		
dsMapSizeSel	Map selectivity in terms of size	✓		
dsMapRecsSel	Map selectivity in terms of records	✓		
dsReduceSizeSel	Reduce selectivity in terms of size	✓		
dsReduceRecsSel	Reduce selectivity in terms of records	✓		
dsCombineSizeSel	Combine selectivity in terms of size	✓		✓
dsCombineRecsSel	Combine selectivity in terms of records	✓		✓
dsInputCompressRatio	Input data compression ratio	✓		
dsIntermCompressRatio	Map output compression ratio	✓		✓
dsOutCompressRatio	Output compression ratio	✓		✓
dsStartupMem	Startup memory per task	✓		
dsSetupMem	Setup memory per task	✓		
dsCleanupMem	Cleanup memory per task	✓		
dsMemPerMapRec	Memory per map's record	✓		
dsMemPerRedRec	Memory per reduce's record	✓		

in j's profile are used by the What-if Engine to predict the behavior of hypothetical jobs that run the same MapReduce program as j.

Workflow Profiles: As discussed in Section 3.1, a MapReduce workflow W is a directed acyclic graph (DAG) G_W that represents a set of MapReduce jobs and their dataflow dependencies. Equivalently, a *workflow profile* is a DAG of job profiles connected based on the dataflow dependencies among the corresponding jobs in the workflow. Each job profile in the workflow profile is generated as soon as a job completes execution. The dataflow dependencies among the jobs are determined in one of two ways. When the workflow is automatically generated by a higher-level system like Pig or Hive, the dataflow dependencies can be obtained directly from the system itself, as it already tracks such dependencies. When this information is not available, Starfish will deduce the dependencies from the paths of the input and

Table 4.4: Cost statistics fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields represent information at the level of tasks)	Depends On		
		d	r	c
csHdfsReadCost	I/O cost for reading from HDFS per byte		✓	
csHdfsWriteCost	I/O cost for writing to HDFS per byte		✓	
csLocalIOReadCost	I/O cost for reading from local disk per byte		✓	
csLocalIOWriteCost	I/O cost for writing to local disk per byte		✓	
csNetworkCost	Cost for network transfer per byte		✓	
csMapCPUCost	CPU cost for executing the Mapper per record		✓	
csReduceCPUCost	CPU cost for executing the Reducer per record		✓	
csCombineCPUCost	CPU cost for executing the Combiner per record		✓	
csPartitionCPUCost	CPU cost for partitioning per record		✓	
csSerdeCPUCost	CPU cost for serializing/deserializing per record		✓	
csSortCPUCost	CPU cost for sorting per record		✓	
csMergeCPUCost	CPU cost for merging per record		✓	
csInUncomprCPUCost	CPU cost for uncompr/ing the input per byte		✓	
csIntermUncomCPUCost	CPU cost for uncompr/ing map output per byte		✓	✓
csIntermComCPUCost	CPU cost for compressing map output per byte		✓	✓
csOutComprCPUCost	CPU cost for compressing the output per byte		✓	✓
csSetupCPUCost	CPU cost of setting up a task		✓	
csCleanupCPUCost	CPU cost of cleaning up a task		✓	

output data processed and generated by each job in the workflow.

Information contained in a workflow profile can be used to reconstruct the entire execution of the MapReduce jobs after their completion, in order to better understand and analyze their overall behavior as well as the various dependencies among them. In addition, as we will see in Chapter 6, the workflow profile is utilized to answer hypothetical what-if questions regarding the behavior of the workflow under different scenarios.

Job and workflow profiles are a very powerful abstraction for representing the execution of any arbitrary MapReduce program or any query expressed in a higher-level language like Pig Latin or HiveQL. Apart from using profiles in answering what-if questions and automatically recommending configuration parameter settings, the profiles also help in understanding the job behavior as well as in diagnosing bottlenecks during job execution.

Table 4.5: A subset of job profile fields for two Word Co-occurrence jobs run with different settings for *io.sort.mb*.

Information in Job Profile	*io.sort.mb*	
	120	200
Number of spills	12	8
Number of merge rounds	2	1
Combine selectivity in terms of size	0.70	0.67
Combine selectivity in terms of records	0.59	0.56
Map output compression ratio	0.39	0.39
File bytes read in map task	133 MB	102 MB
File bytes written in map task	216 MB	185 MB

4.2 Using Profiles to Analyze Execution Behavior

The job profiles allow for an in-depth analysis of the task behavior in terms of resource allocation and usage, concurrency control, I/O optimization, and network bandwidth usage. We will illustrate the benefits of the job profiles and give insights into the complex nature of parameter impact and interactions through an example based on actual experiments.

Suppose a company runs a *Word Co-occurrence* MapReduce job periodically on around 10GB of data. This program is popular in Natural Language Processing (NLP) to compute the word co-occurrence matrix of a large text collection (Lin and Dyer, 2010). A data analyst at the company notices that the job runs in around 1400 seconds on the company's production Hadoop cluster. Based on the standard monitoring information provided by Hadoop, the analyst also notices that map tasks in the job take a large amount of time and do a lot of local I/O. Her natural inclination—which is also what rule-based tools for Hadoop would suggest (discussed later in Section 7.1)—is to increase the map-side buffer size (namely, the *io.sort.mb* parameter in Hadoop). However, when she increases the buffer size from the current 120MB to 200MB, the job's running time degrades by 15% as shown in Table 4.5. The analyst may be puzzled and frustrated.

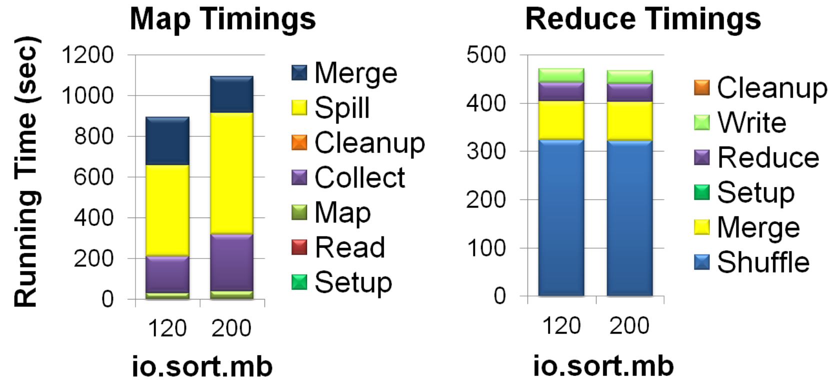


FIGURE 4.1: Map and reduce time breakdown for two Word Co-occurrence jobs run with different settings for *io.sort.mb*.

By using our Profiler to collect job profiles, the data analyst can visualize the task-level and phase-level Cost (timing) fields as shown in Figure 4.1. It is obvious immediately that the performance degradation is due to a change in map performance; and the biggest contributor is the change in the Spill phase's cost. The analyst can drill down to the values of the relevant profile fields, which we show in Figure 4.1. The values shown report the average across all map tasks.

The interesting observation from Figure 4.1 is that changing the map-side buffer size from 120MB to 200MB improves all aspects of local I/O in map task execution: the number of spills reduced from 12 to 8, the number of merges reduced from 2 to 1, and the Combine function became more selective. Overall, the amount of local I/O (reads and writes combined) per map task went down from 349MB to 287MB. However, the overall performance still degraded.

To further understand the job behavior, we run the Word Co-occurrence job with different settings of the map-side buffer size (*io.sort.mb*). Figure 4.2 shows the overall map execution time, and the time spent in the map-side Spill and Merge phases, from our runs. The input data and cluster resources are identical for the runs. Notice the map-side buffer size's nonlinear effect on cost, which comes from an interesting tradeoff: a larger buffer lowers overall I/O size and cost (Figure 4.1), but increases

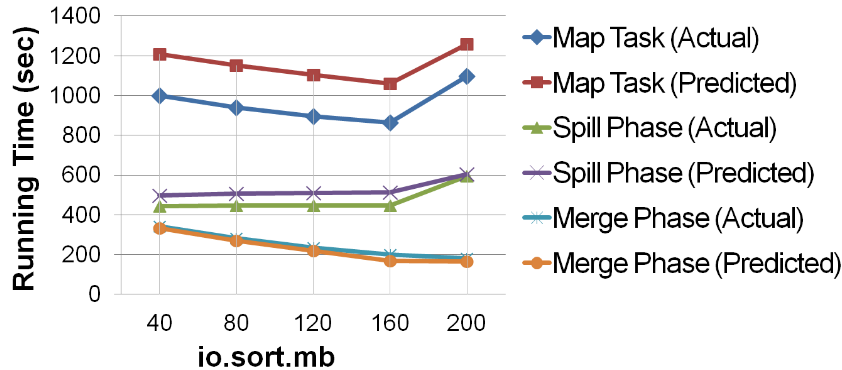


FIGURE 4.2: Total map execution time, Spill time, and Merge time for a representative Word Co-occurrence map task as we vary the setting of *io.sort.mb*.

the computational cost nonlinearly because of sorting.

Figure 4.2 also shows the corresponding execution times as predicted by Starfish's What-if Engine. We observe that the What-if Engine correctly captures the underlying nonlinear effect that caused this performance degradation; enabling the Cost-based Optimizer to find the optimal setting of the map-side buffer size. The fairly uniform gap between the actual and predicted costs is due to overhead added by BTrace (our profiling tool discussed below) while measuring function timings at nanosecond granularities.[2] Because of its uniformity, the gap does not affect the accuracy of what-if analysis which, by design, is about relative changes in performance.

4.3 Generating Profiles via Measurement

A job profile (and, by extension, a workflow profile) is generated by one of two methods. We will first describe how the *Profiler* generates profiles from scratch by collecting monitoring data during full or partial job execution. The second method is by estimation, which does not require the job to be run. Profiles generated by this method are called *virtual profiles*. The What-if Engine is responsible for generating

[2] We expect to close this gap using commercial Java profilers that have demonstrated vastly lower overheads than BTrace (Louth, 2009).

virtual profiles as part of answering a what-if question (discussed in Chapter 6).

Monitoring through dynamic instrumentation: When a user-specified MapReduce program p is run, the MapReduce framework is responsible for invoking the map, reduce, and other functions in p. This property is used by the Profiler to collect run-time monitoring data from unmodified programs running on the MapReduce framework, using *dynamic instrumentation*. Dynamic instrumentation is now a popular technique used to understand, debug, and optimize complex systems (Cantrill et al., 2004). The Profiler applies dynamic instrumentation to the MapReduce framework—not to the MapReduce program p—by specifying a set of *event-condition-action (ECA)* rules.

The space of possible events in the ECA rules corresponds to events arising during program execution, such as entry or exit from functions, memory allocation, and system calls to the operating system. If the condition associated with the event holds when the event fires, then the associated action is invoked. An action can involve, for example, getting the duration of a function call, examining the memory state, or counting the number of bytes transferred.

The current implementation of the Profiler for the Hadoop MapReduce framework uses the *BTrace* dynamic instrumentation tool for Java (BTrace, 2012). To collect raw monitoring data for a program being run by Hadoop, the Profiler uses ECA rules (also specified in Java) to dynamically instrument the execution of selected Java classes internal to Hadoop. Under the covers, dynamic instrumentation intercepts the corresponding Java class bytecodes as they are executed, and injects additional bytecodes to run the associated actions in the ECA rules.

Apart from Java, Hadoop can run a MapReduce program written in various programming languages such as Python, R, or Ruby using *Streaming*, or C++ using *Pipes* (White, 2010). Hadoop executes Streaming and Pipes jobs through special map

and reduce tasks that each communicate with an external process to run the user-specified map and reduce functions (White, 2010). Since the Profiler instruments only the MapReduce framework, not the user-written programs, profiling works irrespective of the programming language in which the program is written.

From raw monitoring data to profile fields: The raw monitoring data collected through dynamic instrumentation of job execution at the task and phase levels includes record and byte counters, timings, and resource usage information. For example, during each spill, the exit point of the sort function is instrumented to collect the sort duration as well as the number of bytes and records sorted. A series of post-processing steps involving aggregation and extraction of statistical properties is applied to the raw data in order to generate the various fields in the job profile (recall Section 4.1).

The raw monitoring data collected from each profiled map or reduce task is first processed to generate the fields in a *task profile*. For example, the raw sort timings are added as part of the overall spill time, whereas the Combine selectivity from each spill is averaged to get the task's Combine selectivity. The map task profiles are further processed to give one representative map task profile for each logical input to the MapReduce program. For example, a Sort program accepts a single logical input (be it a single file, a directory, or a set of files), while a two-way Join accepts two logical inputs. The reduce task profiles are processed into one representative reduce task profile. The representative map task profile(s) and the reduce task profile together constitute the job profile.

The aggregated dataflow and cost fields in the profiles provide a global view of the job execution, whereas the aggregated dataflow and cost statistics fields are essential for estimating the profile fields for hypothetical jobs (discussed in Chapter 6. Apart from point-value fields, the Profiler can potentially be used to collect all

individual key-value flows across tasks to compute *key-value distributions* for input, intermediate, and output data. Such information opens up new tuning possibilities, especially for higher-level systems like Pig and Hive. For example, Pig could use information about intermediate data distributions for automatically selecting the partitioning function or appropriate join algorithm.

Current approaches to profiling in Hadoop: Monitoring facilities in Hadoop—which include *logging*, *counters*, and *metrics*—provide historical data that can be used to monitor whether the cluster is providing the expected level of performance, and to help with debugging and performance tuning (White, 2010).

Hadoop counters are a useful channel for gathering statistics about a job for quality control, application-level statistics, and problem diagnosis. Hadoop contains a set of built-in counters measuring task-level statistics, like the number of input and output records for each task, and the number of bytes read and written to the file-systems. Counters are collected for MapReduce tasks and aggregated for the whole job. In addition, Hadoop offers support for user-defined counters. Counters are similar to the Dataflow fields in a job profile, and can be useful in setting some job configuration parameters. For example, the total number of records spilled to disk may indicate that some memory-related parameters in the map task need adjustment; but the user cannot automatically know which parameters to adjust or how to adjust them to improve the job's performance.

The HDFS and MapReduce daemons in Hadoop expose runtime metrics about events and measurements (White, 2010). For example, HDFS Datanodes[3] collect metrics recording the number of bytes written, the number of blocks replicated, and the number of read requests in a node. Even though metrics have similar uses to counters, they represent cluster-level information and their target audience is

[3] Datanodes are HDFS entities run on each slave node in the cluster and are responsible for storing and retrieving file blocks from each node.

system administrators, not regular users. In addition, Hadoop metrics have a lot of dependencies on third party software and libraries as they have to be analyzed by cluster monitoring systems like Ganglia or Nagios.

Information similar to counters and metrics forms only a fraction of the information in the job profiles collected by the Profiler. Apart from an extensive list of counters, the job profile contains (i) statistical information like map and reduce selectivities, (ii) quantitative costs for executing user-provided functions like maps, reduces, and combiners, and (iii) time spent in the various task phases.

4.4 Task-level Sampling to Generate Approximate Profiles

Another valuable feature of dynamic instrumentation is that it can be turned on or off seamlessly at run-time, incurring zero overhead when turned off. However, it does cause some task slowdown when turned on. We have implemented two techniques that use task-level sampling in order to generate approximate job profiles while keeping the run-time overhead low:

1. If the intent is to profile a job j during a regular run of j on the production cluster, then the Profiler can collect task profiles for only a sample of j's tasks.

2. If the intent is to collect a job profile for j as quickly as possible, then the Profiler can selectively execute (and profile) only a sample of j's tasks.

Consider a job with 100 map tasks. With the first approach and a sampling percentage of 10%, all 100 tasks will be run, but only 10 of them will have dynamic instrumentation turned on. In contrast, the second approach will run (and profile) only 10 of the 100 tasks.

In order to examine the relationship between task-level sampling and the ability to generate job profiles based on which Starfish can make fairly accurate decisions,

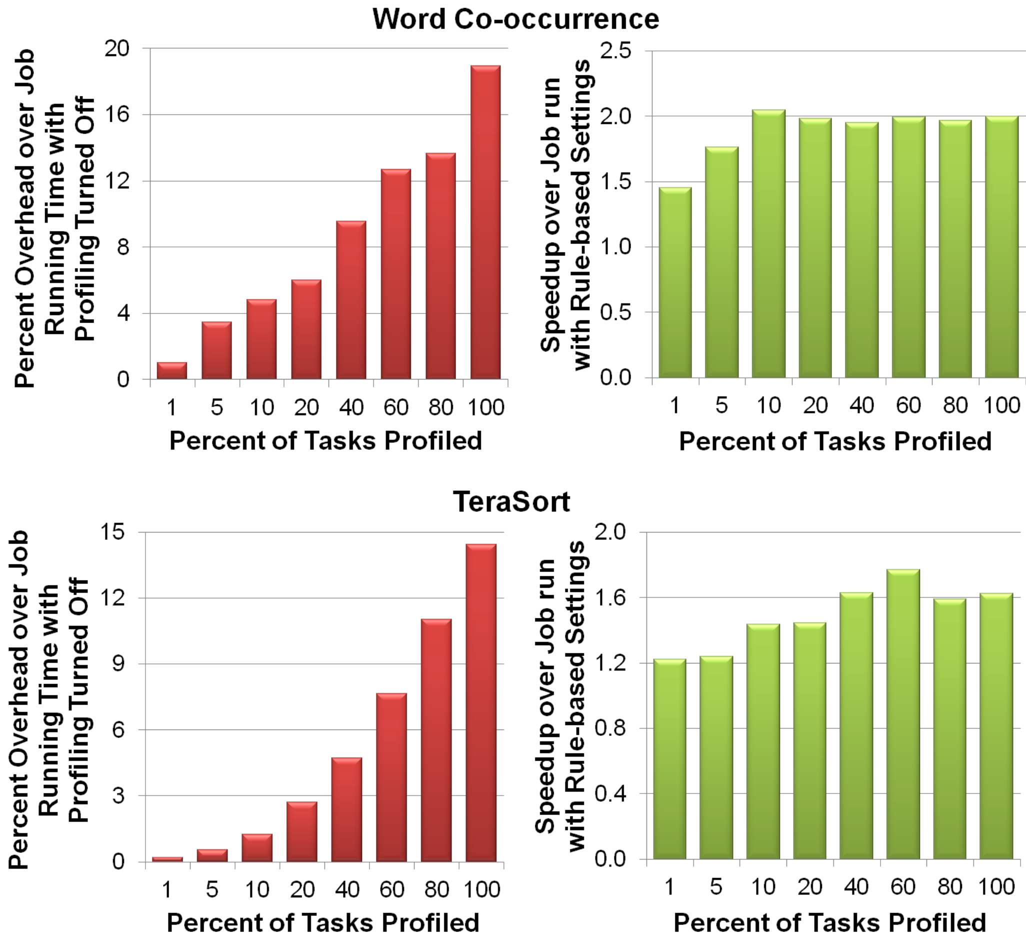


FIGURE 4.3: (a) Overhead to measure the (approximate) profile, and (b) corresponding speedup given by Starfish as the percentage of profiled tasks is varied for Word Co-occurrence and TeraSort MapReduce jobs.

we performed the following experiment. We ran and profiled Word Co-occurrence (a CPU-intensive job) and TeraSort (an I/O-intensive job) on a 16-node Hadoop cluster with c1.medium EC2 nodes, while enabling profiling for only a sample of the tasks in each job. We then used the generated (approximate) job profiles to get recommendations for parameter settings from Starfish. (Understanding the details of how Starfish works at this point is not needed for appreciating the results.)

As we vary the percentage of profiled tasks in each job, Figure 4.3(a) shows the profiling overhead compared against the same job run with profiling turned off. For

both MapReduce jobs, as the percentage of profiled tasks increases, the overhead added to the job's running time also increases (as expected). Profiling all the map and reduce tasks in each job adds around 15% to 20% overhead to the job's execution time. It is interesting to note that the profiling overhead varies significantly between the two jobs. The magnitude of the profiling overhead depends on whether the job is CPU-bound, uses a combine function, or uses compression, as well as on the job configuration settings.

Figure 4.3(b) shows the speedup achieved in job execution time by the Starfish-suggested settings over rule-based settings as the percentage of profiled tasks used to generate the job profile is varied. In most cases, the settings suggested by Starfish led to nearly the same job performance improvements; showing that the Starfish's effectiveness in finding good configuration settings does not require that all tasks be profiled. In fact, by profiling just 10% of the tasks, Starfish can often achieve the same speedup as by profiling 100% of the tasks.

It is particularly encouraging to note that by profiling just 1% of the tasks—in this specific cases, profiling only one map and one reduce task—with near-zero overhead on job execution, Starfish finds configuration settings that still provide a speedup over the jobs run with rule-based settings. We have repeated this experiment with a large number of different MapReduce programs and, in most cases, the settings suggested by Starfish using sampling led to nearly the same job performance improvements like above. Therefore, by profiling only a small fraction of the tasks, we can keep the overhead low while achieving high degrees of accuracy in the collected information.

5

A Declarative Query Interface to Access Performance Predictors and Optimizers

Starfish is a self-tuning system for running analytics workloads on Big Data. As described in Section 2.2.1 and illustrated in Figure 2.1, Starfish is built on the Hadoop platform. Starfish interposes itself between Hadoop and higher-level clients (e.g., Pig, Hive, Oozie, and command-line interfaces) to submit MapReduce jobs. These Hadoop clients will now submit workloads—which can vary from a single MapReduce job, to a workflow of MapReduce jobs, and to a collection of multiple workflows—directly to Starfish. Starfish provides a wide range of features to the users, including, but not limited to, the following:

1. The user has the option to either execute the MapReduce workload on the Hadoop cluster as is or to enable the collection of profiling information during execution. In the former case, the workload will execute with zero overhead and Starfish will get access to Hadoop's default logs and counters; whereas in the latter case, Starfish will generate fine-grained job and workflow profiles. This

information is organized, stored, and managed by Starfish's Data Manager[1].

2. The information collected by the Profiler helps in understanding the job behavior as well as in diagnosing bottlenecks during job execution. For this purpose, Starfish provides both a command-line and a graphical user interface for exploring and analyzing the collected data. The profiles are also required for Starfish to predict and optimize the performance of MapReduce workloads.

3. The What-if Engine can predict the performance of a MapReduce job j (or workflow W), allowing the user to study the effects of configuration parameters, cluster resources, and input data on the performance of j (or W); without actually running j (or W).

4. The Job and Workflow Optimizers can find the optimal configuration settings for j (or W), and can also help understand why the current settings are (possibly) suboptimal.

5. The Cluster Resource Optimizer can find the optimal number of nodes and node types to use for a MapReduce workload given objectives and constraints on execution time and/or cost.

At the same time, we want Starfish to be usable in environments where workloads are run directly on Hadoop without going through Starfish. For this purpose, Starfish can be run in a special *recommendation mode*. In this mode, Starfish uses its tuning features to only recommend good configurations instead of running the workload with these configurations as Starfish would do in its normal usage mode.

Predicting the performance of MapReduce workloads, finding MapReduce parameter settings, and determining the best cluster resources to use are instances of *workload tuning problems* Starfish can solve. We have developed a declarative query

[1] Recall the components in the Starfish architecture shown in Figure 2.2 in Section 2.2.1.

interface through which the users can express *workload tuning queries*. Starfish will then provide reliable answers to these queries using an automated technique; providing nonexpert users with a good combination of cluster resource and job configuration settings to meet their needs. The automated technique is based on a careful mix of job profiling, estimation using black-box and white-box models, and simulation. Applications and users can also interact with this interface using a programmatic API or using a graphical interface that forms part of the Starfish system's Visualizer.

5.1 Declarative Interface to Express Workload Tuning Queries

A general tuning problem involves determining the cluster resources and MapReduce job-level configuration settings to meet desired performance requirements on execution time and cost for a given analytic workload. Starfish provides a declarative interface to express a range of tuning queries including those arising in the use cases discussed in Section 3.4.

A tuning query specified using Starfish's declarative query interface has four parts. We discuss each of these parts in turn.

1. Specifying the workload: The workload specified in a tuning query consists of MapReduce jobs, including both single jobs as well as jobs from multi-job workflows. Each job j runs some MapReduce program p on input data d. A profile *prof*(p) has to be given for p. *prof*(p) need not, and usually will not, correspond to the actual job $j = \langle p, d, r, c \rangle$ that eventually runs p as part of the workload. If d is different from the input data used while generating *prof*(p), then the properties of d have to be given. For user convenience, when d is actual data on a live cluster, Starfish can collect the properties of d (and r) automatically from the cluster. The cluster resources r and job configuration c to use in the actual execution of j are part of separate specifications that involve search spaces, discussed next.

2. Specifying the search space for cluster resources r: Recall from Section 3.1 that the properties used to represent any cluster resources r include the number of nodes, node type(s), and network topology of r, the number of map and reduce task execution slots per node, and the maximum memory available per task execution slot. A search space over the number of nodes and the node type to use in the cluster is specified for r. This search space is specified as a nonempty set. It is the responsibility of Starfish to find a suitable cluster resource configuration from this search space that meets all other requirements in the query.

The search space for r will be a singleton set if the user is asking for performance estimates for a specific (hypothetical or real) cluster that she has in mind. Use cases 1, 2, and 3 from Section 3.4 have this nature. For example, in use case 2, the user currently has a cluster containing 10 EC2 nodes of the m1.large type. She wants Starfish to estimate what the job's execution time will be on a hypothetical cluster containing 15 nodes of the same type.

The search space is nonsingleton—i.e., it specifies a space that contains more than one candidate resource configuration—when the user wants Starfish to search for a good configuration that meets her needs. In such cases, Starfish's declarative query interface gives the user considerable flexibility to define the search space for cluster resources r. For example:

- An unsophisticated user can use the special "*" symbol to ask Starfish to use its default search space for one or both of the node type and the number of nodes.

- The user can restrict the search space to nodes of one or more selected types, but ask Starfish to find the best number of nodes in the cluster.

Our current implementation of the Cluster Resource Optimizer does not include the cluster's network topology in the search space. This limitation, which can be

removed in the future, is driven partly by a practical consideration: most current cloud providers hide the underlying network topology from clients. The Hadoop clusters that we run on EC2 are configured as per the single-rack network topology used by default in Hadoop.

The Hadoop cluster-wide configuration parameters—namely, the number of map and reduce task execution slots per node, and the maximum memory available per task execution slot—are also not included in the search space. Our empirical studies indicate that good settings for these parameters are determined primarily by the CPU (number of cores) and memory resources available per node in the cluster; so we use empirically-determined fixed values per node type (these values are shown in Table 6.3).[2]

3. Specifying the search space for job configurations c: Recall the space of configuration parameter settings for MapReduce jobs presented in Section 3.1 and Table 3.1. A tuning query needs to specify the search space for configuration parameters c that Starfish should consider for the given workload of MapReduce jobs. Similar to the search space for cluster resources, the search space for c will be a singleton set if the user is asking for performance estimates for a specific configuration that she has in mind. Use case 2 from Section 3.4 has this nature.

It is much more common to specify a larger search space for c. The best setting of configuration parameters depends strongly on the cluster resource configuration. For the convenience of nonexpert users who often have no idea about the configuration parameters, the special "*" symbol can be specified to ask Starfish to use its default search space for c.

The Job and Workflow Optimizers in Starfish are responsible for searching efficiently through the specified space of configuration parameter settings. Starfish uses

[2] Anecdotal evidence from the industry suggests that memory-related misconfigurations are a major cause of failures in Hadoop.

the Job and Workflow Optimizers in tandem with the Cluster Resource Optimizer since the best job configuration will invariably change if the cluster resources change.

4. Specifying performance requirements: Execution time and cost are the two performance metrics supported by Starfish. As part of a tuning query, a user can choose to:

- Have estimated values output for one or both of these metrics.
- Optimize one of these metrics, possibly subject to a constraint on the other metric. For example, optimizing monetary cost subject to an execution time under 30 minutes.

5.2 Overview of How Starfish Answers a Workload Tuning Query

This section gives an overview of how Starfish answers a tuning query posed by a user or application. Chapters 6 and 7 will describe the individual steps in more detail. To simplify the discussion, we will use concrete examples arising from the use cases described in Section 3.4.

1. Tuning job-level configuration parameter settings: Consider a MapReduce job $j = \langle p, d, r, c \rangle$ that run on a 10-node Hadoop cluster using 20 reduce tasks. The user is interested in the affect of the number of reduce tasks (one of the configuration parameters) on the performance of the job. In this scenario, the user can ask for an estimate of the execution time of job $j' = \langle p, d, r, c' \rangle$ whose configuration c' is the same as c except that c' specifies using 40 reduce tasks. The MapReduce program p, input data d, and cluster resources r remain unchanged. A user can express this use case as a tuning query Q_1 using Starfish's query interface described in Section 5.1. The user will specify the following:

- The profile for the run of job j on the current Hadoop cluster. Chapter 4

described how the profile can be generated by measurement as part of an actual job execution.

- The search space for cluster resources r is a singleton set that specifies the current Hadoop cluster.
- The search space for the configuration parameter settings is also a singleton set that specifies c'.
- The performance metric of interest is execution time.

Query Q_1 maps directly to a *what-if question* that can be answered by the What-if Engine. The What-if Engine will first estimate a virtual job profile for the hypothetical job j'. This step uses a careful mix of white-box and black-box models. The virtual profile is then used in a simulation step to estimate how the hypothetical job j' will execute. The answer to the what-if question is computed based on the estimated execution. Note that job j' is never run during this process. Chapter 6 explains the overall prediction process in detail.

A what-if question can involve multiple jobs in a workload. In this case, all the virtual job profiles are generated, and then input to the simulation step.

2. Tuning the cluster size for elastic workloads: Suppose a MapReduce job j takes three hours to finish on a 10-node Hadoop cluster of EC2 nodes of the m1.large type. The user who controls the cluster wants to know by how much the execution time of the job will reduce if five more m1.large nodes are added to the cluster. This use case is expressed as a tuning query Q_2 by specifying the following:

- The profile for the run of j on 10 m1.large EC2 nodes.
- The search space for cluster resources r is a singleton set that specifies 15 EC2 nodes of the (same) m1.large type.

- The search space for the configuration parameter settings c is also a singleton set that specifies the same job configuration as for the 10-node cluster.

- The performance metric of interest is execution time.

The above specification of query Q_2 gives Starfish a profile for a job $j = \langle p, d, r_1, c\rangle$. The desired answer is the estimate of execution time for a hypothetical job $j' = \langle p, d, r_2, c\rangle$. Job j' runs the same program p on the same data d and MapReduce configuration c as job j. However, the cluster resources used are different between j and j', i.e., $r_1 \neq r_2$. Similar to Query Q_1, Query Q_2 maps directly to a what-if question. However, the answer to Q_2 involves estimating the performance of j' on a hypothetical cluster instead of a real one.

3. Planning for workload transition from a development cluster to production: To express this use case as a tuning query Q_3, a user will specify the following:

- A job profile collected on the development cluster for the program p of interest. This profile may have been collected by direct measurement when p was run on a (scaled down) sample of the production data on which the program has to be run on the production cluster. In this case, the properties of the production data d_{prod} will have to be provided.

- The search space for cluster resources r is a singleton set that gives the properties of the production cluster, denoted r_{prod}.

- Note that the developer wants Starfish to find the best job configuration to run the job on the production cluster. Thus, the search space for the configuration parameter settings c is specified as "*" so that Starfish will consider the full space in an efficient fashion.

- The performance metric of interest is execution time.

To process query Q_3, Starfish will invoke the cost-based Job Optimizer to efficiently enumerate and search through the high-dimensional space of configuration parameter settings. The Job Optimizer will consider hypothetical configurations $c^{(1)}$, $c^{(2)}$, ... $c^{(i)}$, ..., $c^{(n)}$, making calls to the What-if Engine to get the estimated execution time of each of the corresponding hypothetical jobs $j^{(i)} = \langle p, d_{prod}, r_{prod}, c^{(i)} \rangle$. The configuration $c^{(opt)}$ found in this process with the least execution time will be output as the query result; the details of the efficient search process are given in Chapter 7.

4. Cluster provisioning under multiple objectives: This use case differs from the earlier use cases in two main ways. First, the search space for cluster resources is not a singleton set any more. Second, the performance requirements demand optimization of monetary cost, while specifying a constraint on the execution time. (Note that optimizing the execution time with a constraint on monetary cost is also supported.) A user will specify the following:

- A job profile collected on a Hadoop cluster with 6 nodes of m1.large EC2 type for the program p of interest.
- The search space for cluster resources r specifies "*" so that Starfish will consider all available instance types and cluster sizes.
- The search space for the configuration parameter settings c is also specified as "*" so that Starfish will consider the full space in an efficient fashion.
- The objective is to minimize the monetary cost while the job runs in less than 2 hours.

The Cluster Resource Optimizer and the Job Optimizer will be used in tandem to enumerate and search over the space of cluster resources and job configurations;

while making calls to the What-if Engine to get estimates of execution time and cost for hypothetical jobs $j^{(i,j)} = \langle p, d, r^{(i)}, c^{(j)} \rangle$. The combination $\langle r,c \rangle^{(opt)}$ found in this process that gives the least monetory cost while meeting the constraint on execution time will be output as the query result.

5.3 Starfish Visualizer

The *Visualizer* forms the graphical user interface to Starfish. Users employ the Visualizer to (a) get a deep understanding of a MapReduce job's behavior during execution, (b) ask hypothetical questions on how the job behavior will change when parameter settings, cluster resources, or input data properties change, and (c) ultimately optimize the job. Hence, we categorize the core functionalities of the Visualizer into *Job Analysis*, *What-if Analysis*, and *Job Optimization*. For each functionality, the Visualizer offers five different views:

1. *Timeline views*, used to visualize the progress of job execution at the task level.
2. *Data Skew views*, used to identify the presence of data skew in the input and output data of map and reduce tasks.
3. *Data Flow views*, used to visualize the flow of data among the nodes of a Hadoop cluster, and between the map and reduce tasks of a job.
4. *Profile views*, used to show the detailed information exposed by the job profiles, including the phase timings within the tasks.
5. *Settings views*, used to list the configuration parameter settings, cluster resources, and the input data properties during job execution.

We will demonstrate the Visualizer's functionalities in order, and show how the user can obtain deep insights into a job's performance from each view in each case.

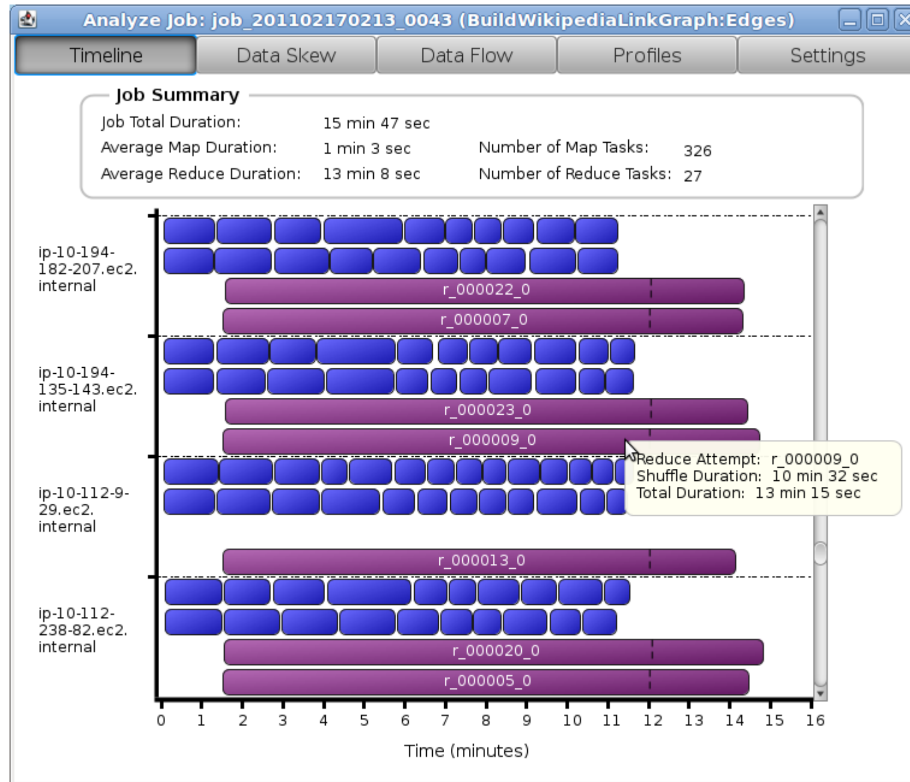


FIGURE 5.1: Screenshot from the Starfish Visualizer showing the execution timeline of the map and reduce tasks of a MapReduce job running on a Hadoop cluster. A user can quickly get useful information such as the number of map and reduce waves or the presence of variance in the task execution times.

Job Analysis: Recall from Section 4.3 that when a MapReduce job executes on a Hadoop cluster, the Profiler collects a wealth of information including logs, counters, and profiling data. Figure 5.1[3] shows the execution timeline of map and reduce tasks that ran during a MapReduce job execution. The user can get information such as how many tasks were running at any point in time on each node, when each task started and ended, or how many map or reduce waves occurred during job execution. The user is able to quickly spot any high variance in the task execution times, and discover potential load-balancing issues. Moreover, Timeline views can be used to compare different executions of the same job run at different times or with different

[3] All figures are actual screenshots from the Starfish Visualizer.

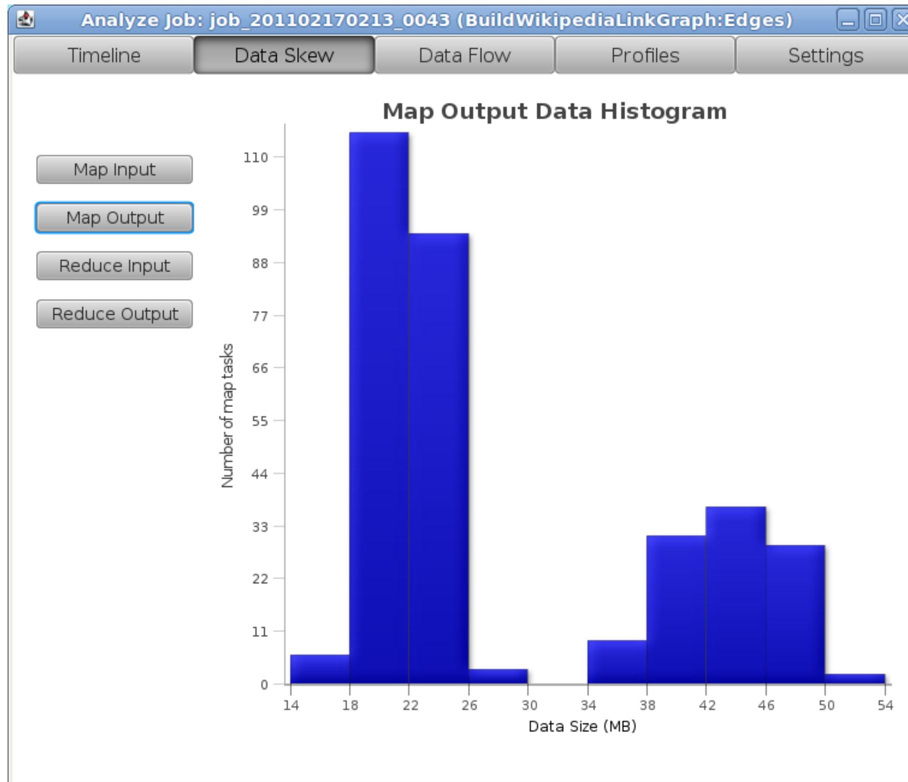


FIGURE 5.2: Screenshot from the Starfish Visualizer showing a histogram of the map output data size per map task that can be used to identify data skew. This particular histogram shows that 33% of the map tasks output about 2x more data than the other map tasks.

parameter settings. Comparison of timelines shows whether the job behavior changed over time and helps understand the impact of changing parameter settings on job execution.

While the Timeline views are useful in identifying computational skew, the Data Skew views (shown in Figure 5.2) can readily help identify the presence of skew in the data consumed and produced by the map and reduce tasks. Data skew in the reduce tasks usually indicates a strong need for a better partitioning function in the current MapReduce job. Data skew in the map tasks corresponds to properties of the input data, and may indicate the need for a better partitioning function in the producer job that generates the input data.

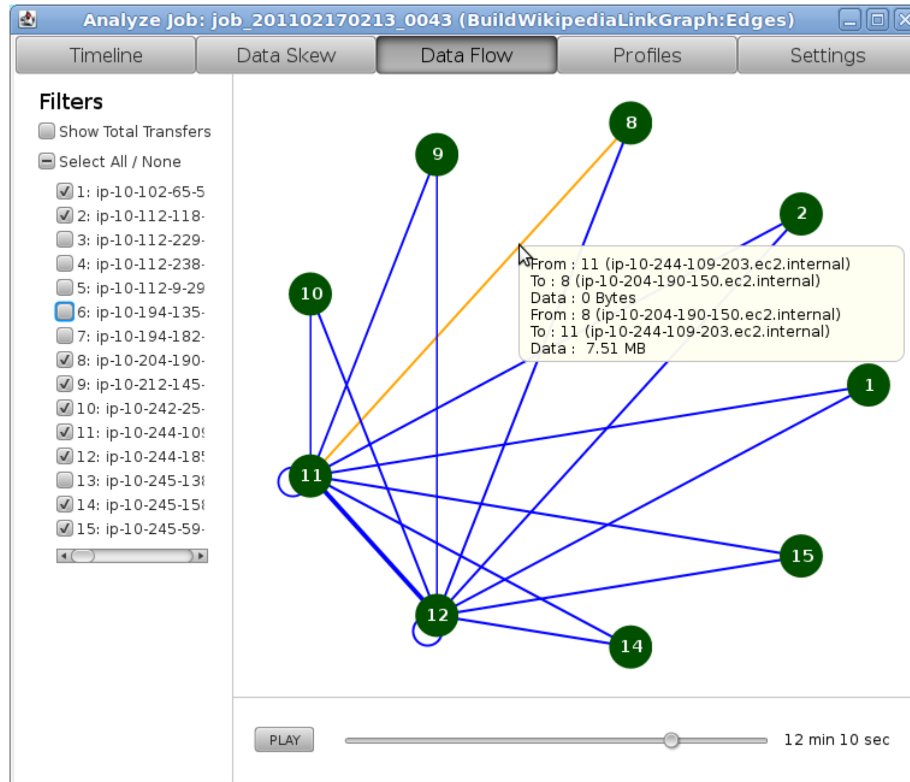


FIGURE 5.3: Screenshot from the Starfish Visualizer showing a visual representation of the data flow among the Hadoop nodes during a MapReduce job execution. The provided Video mode allows a user to inspect how data was transfered among the nodes of the cluster as time went by.

The Data Skew views are complemented by the Data Flow views used to identify data skew across the Hadoop nodes caused during the Shuffle phase of the MapReduce job execution. Figure 5.3 presents the data flow among some cluster nodes during the execution of a MapReduce job. The thickness of each line is proportional to the amount of data that was shuffled between the corresponding nodes. The user also has the ability to specify a set of filter conditions (see the left side of Figure 5.3) that allows her to zoom in on a subset of nodes or on the large data transfers. An important feature of the Visualizer is the *Video mode* that allows users to play back a job execution from the past. Using the Video mode, the user can inspect how data was processed and transfered between the map and reduce tasks of the job, and

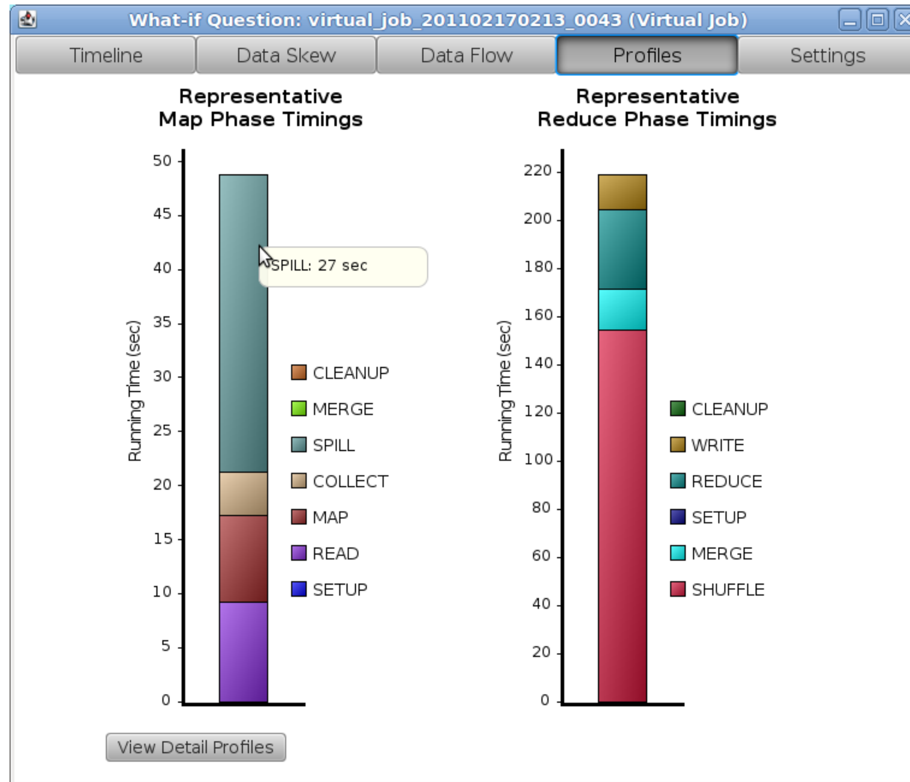


FIGURE 5.4: Screenshot from the Starfish Visualizer showing the map and reduce time breakdown from the virtual profile of a MapReduce job. A user here can quickly spot that the time spent shuffling the map output data to the reduce tasks contributes the most to the total execution time.

among nodes and racks of the cluster, as time went by.

The Profile views help visualize the job profiles, namely, the information exposed by the profile fields at the fine granularity of phases within the map and reduce tasks of a job; allowing for an in-depth analysis of task behavior during execution. For example, Figure 5.4 displays the breakdown of time spent on average in each map and reduce task. The Profile views also form an excellent way of diagnosing bottlenecks during task execution. From the visualization shown in Figure 5.4, even a nonexpert user can spot that the time spent shuffling the map output data to the reduce tasks contributes the most to the total execution time; indicating that the corresponding configuration parameters (e.g., *mapred.job.shuffle.input.buffer.percent* and

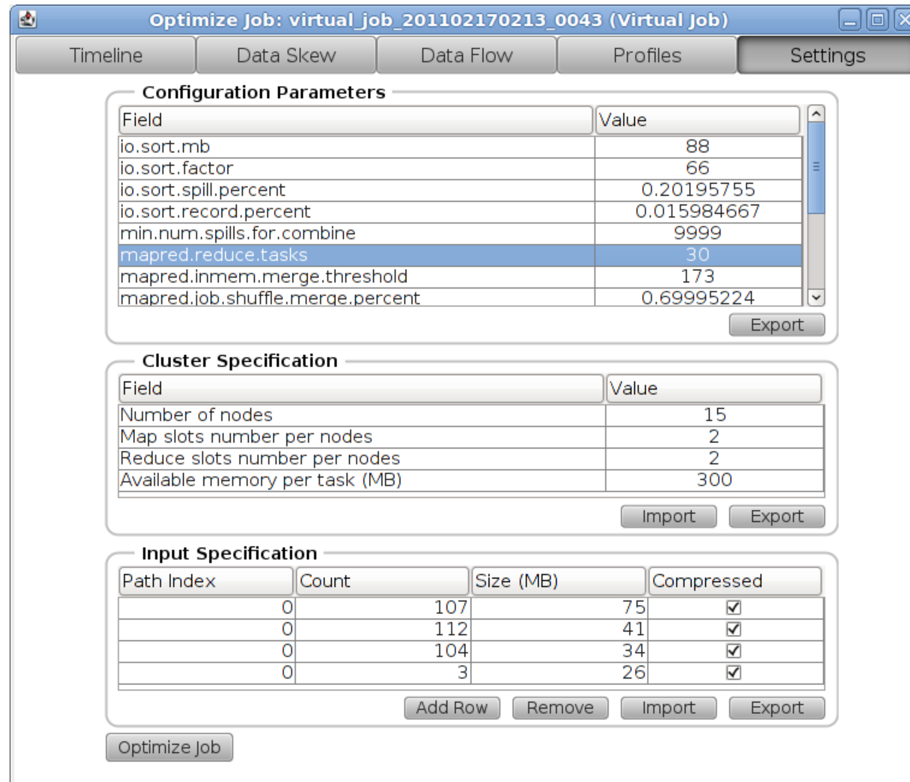


FIGURE 5.5: Screenshot from the Starfish Visualizer showing the optimal configuration parameter settings found by the Job Optimizer, as well as cluster and input data properties. The user also has the option of importing or exporting any of the above settings and properties in XML format.

mapred.job.shuffle.merge.percent from Table 3.1) have settings that are potentially suboptimal.

Finally, the Settings view (see Figure 5.5) lists the most important Hadoop configuration parameters used during the execution of a MapReduce job, as well as the cluster setup and input data properties. The cluster setup is summarized as the number of nodes, the average number of map and reduce slots per node, and the memory available for each task execution. The input data properties include the size and compression of each input split processed by a single map task. The user also has the option of importing or exporting any of the above settings in XML format.

What-if Analysis: The second core functionality provided by the Starfish Visualizer is the ability to answer hypothetical questions about the behavior of a MapReduce job when run under different settings. This functionality allows users to study and understand the impact of configuration parameter settings, cluster resources, and input data properties on the performance of a MapReduce job.

For instance, the user can ask a what-if question of the form: "How will the execution time of a job change if the number of reduce tasks is doubled?" The user can then use the Timeline view to visualize what the execution of the job will look like under the new settings, and compare it to the current job execution. By varying the number of reducers (or any other configuration parameter), the user can determine the impact of changing that parameter on the job execution. Under the hood, the Visualizer invokes the What-if Engine to generate a virtual job profile for the job in the hypothetical setting.

Furthermore, the user can investigate the behavior of MapReduce jobs when changing the cluster setup or the input specification. This functionality is useful in two scenarios. First, many organizations run the same MapReduce programs over multiple datasets with similar data distribution, but different sizes. For example, the same report-generation MapReduce program may be used to generate daily, weekly, and monthly reports. Or, the daily log data collected and processed may be larger for a weekday than the data for the weekend. By modifying the input specification, the user can ask what-if questions on the job behavior when the job is run using datasets of different sizes.

Another common use-case is the presence of a development cluster for generating job profiles. In many companies, developers use a small development cluster for testing and debugging MapReduce programs over small (representative) datasets before running the programs, possibly multiple times, on the production cluster. Again, the user can modify the cluster setup in order to determine in advance how

the jobs will behave on the production cluster. These novel capabilities are immensely useful in Hadoop deployments.

Job Optimization: Perhaps the most important functionality of the Visualizer comes from how it enables a user to invoke the cost-based Job Optimizer to find good configuration settings for executing a MapReduce job on a (possibly hypothetical) Hadoop cluster. The user can then export the configuration settings as an XML file that is used when the same program has to be run in future. At the same time, the user can examine the behavior of the optimal job through the other views provided by the Visualizer.

Similar to the What-if Analysis functionality, the user can modify the cluster and input specifications before optimizing a MapReduce job. Hence, the user can obtain good configuration settings for the same MapReduce program executed over different input datasets and different clusters (per the two usage scenarios presented above). In addition, the user can take advantage of the sampling capabilities of the Profiler to quickly collect a job profile on a sample of the input data. The user can then modify the input specifications in the Settings View of the Visualizer (see Figure 5.5), and find the optimal settings to use when executing the MapReduce program over the full (or a different) dataset.

6

Predicting MapReduce Workload Performance

The ability to *accurately* predict the performance of a MapReduce workload running on a MapReduce cluster is key for answering questions regarding the impact of configuration parameter settings, as well as data and cluster resource properties, on MapReduce workload performance. This chapter will present the overall approach and the detailed performance models we use to overcome the many challenges that arise in distributed settings, such as task parallelism, scheduling, and interactions among tasks. To simplify the presentation, we first focus the discussion on how to predict the performance of individual MapReduce jobs. We then extend the discussion to MapReduce workflows and overall workloads.

MapReduce jobs: Consider a MapReduce job $j = \langle p, d, r, c \rangle$ that runs program p on input data d and cluster resources r using configuration parameter settings c. Job j's performance can be represented as:

$$perf = F(p, d, r, c) \tag{6.1}$$

Table 6.1: Example questions the What-if Engine can answer.

	What-if Questions on MapReduce Job Execution
WIF_1	How will the execution time of job j change if I increase the number of reduce tasks from the current value of 20 to 40?
WIF_2	What is the new estimated execution time of job j if 5 more nodes are added to the cluster, bringing the total to 20 nodes?
WIF_3	How will the execution time of job j change when I execute j on the production cluster instead of the development cluster, and the input data size increases by 60%?
WIF_4	What is the estimated execution time of job j if I execute j on a new cluster with 10 EC2 nodes of type m1.large rather than the current in-house cluster?

Here, *perf* is some performance metric (e.g., execution time) of interest for jobs that is captured by the *performance model* F. The response surfaces shown in Figures 3.5 and 3.6 in Section 3.2 are partial projections of Equation 6.1 for the WordCount and TeraSort MapReduce programs, respectively, when run on a Hadoop cluster. In Starfish, function F from Equation 6.1 is implemented by the *What-if Engine* using a careful mix of analytical, black-box, and simulation models.

Section 3.4 presented various tuning use cases that arise routinely in practice. These use cases give rise to several interesting *what-if questions* that users or applications can express directly to the What-if Engine using Starfish's declarative query interface (recall Chapter 5). For example, consider question WIF_1 from Table 6.1. Here, the performance of a MapReduce job $j = \langle p, d, r, c \rangle$ is known when 20 reduce tasks are used. The number of reduce tasks is one of the job configuration parameters. WIF_1 asks for an estimate of the execution time of job $j' = \langle p, d, r, c' \rangle$ whose configuration c' is the same as c except that c' specifies using 40 reduce tasks. The MapReduce program p, input data d, and cluster resources r remain unchanged.

The What-if Engine can answer any what-if question of the following general form:

Given the profile of a job $j = \langle p, d_1, r_1, c_1 \rangle$ that runs a MapReduce pro-

gram p over input data d_1 and cluster resources r_1 using configuration c_1, what will the performance of program p be if p is run over input data d_2 and cluster resources r_2 using configuration c_2? That is, how will job j' $= \langle p, d_2, r_2, c_2 \rangle$ perform?

To estimate the performance of j' and answer the what-if question, the What-if Engine requires four inputs:

- The job profile generated for p by the Profiler, which captures various aspects of the job's map and reduce task executions (see Chapter 4).
- Information regarding the input dataset d_2, which includes d_2's size, the block layout of files that comprise d_2 in the distributed file-system, and whether d_2 is stored compressed. Note that d_2 may be different from the dataset used while generating the job profile.
- Information regarding the cluster resources r_2, which includes the number of nodes and node types of r_2, the number of map and reduce task execution slots per node, and the maximum memory available per task slot.
- The new settings of the job configuration parameters c_2 to run the job with. The parameters may include the number of map and reduce tasks, the partitioning function, settings for memory-related parameters, the use of compression, and the use of the combine function.

As indicated in Table 6.1, the What-if Engine can answer questions on real and hypothetical input data as well as cluster resources. For questions involving real data and a live Hadoop cluster, the user does not need to provide the information for d_2 and r_2; the What-if Engine can collect this information automatically from the live cluster.

MapReduce workflows: Predicting the performance of individual MapReduce jobs naturally extends to MapReduce workflows. Recall from Section 3.1 that a MapReduce workflow W is a directed acyclic graph (DAG) G_W that represents a set of MapReduce jobs and their dataflow dependencies. Consider a MapReduce workflow W that runs the programs $\{p_i\}$ from the corresponding jobs j_i in G_W on input base datasets $\{b_i\}$ and cluster resources r using configuration parameter settings $\{c_i\}$. W's performance can be represented as:

$$perf = F(\{p_i\}, \{b_i\}, r, \{c_i\}) \tag{6.2}$$

In order to support workflows, the What-if Engine has the ability to simulate the execution of multiple jobs that exhibit producer-consumer relationships or are run concurrently, as well as to estimate properties of derived datasets. Predicting the performance of a single MapReduce job is simply the limiting case for predicting the performance of a MapReduce workflow that consists of a single job. Hence, in reality, the What-if Engine implements function F from Equation 6.2.

A What-if Engine is an indispensable component of any optimizer, just like a costing engine is for a query optimizer in a Database system. However, the uses of a What-if Engine go beyond optimization: it may be used by physical design tools for deciding data layouts and partitioning schemes; it may be used as part of a simulator that helps make critical design decisions during a Hadoop setup—like the size of the cluster, the network topology, and the node compute capacity; or it may help make administrative decisions—like how to allocate resources effectively or which scheduling algorithm to use.

Algorithm for predicting MapReduce workflow performance

Input: Workflow profile, Cluster resources, Base dataset properties,
Configuration settings

Output: Prediction for the MapReduce workflow performance

```
For each (job profile in workflow profile in topological sort order) {
    Estimate the virtual job profile for the hypothetical job (Sections 6.2, 6.3, and 6.4);
    Simulate the job execution on the cluster resources (Section 6.5);
    Estimate the data properties of the hypothetical derived dataset(s) and the overall
      job performance (Section 6.6);
}
```

Figure 6.1: Overall process used by the What-if Engine to predict the performance of a given MapReduce workflow.

6.1 Overview for Predicting MapReduce Workload Performance

Figure 6.1 shows the overall process for predicting the performance of a MapReduce workflow. The DAG of job profiles in the workflow profile is traversed in topological sort order to ensure that the What-if Engine respects the dataflow dependencies among the jobs. For each job profile, the *virtual job profile* is estimated for the new hypothetical job j' based on the new configuration settings, the cluster resources, and the properties of the data processed by j' (Sections 6.2, 6.3, and 6.4). The virtual profile is then used to simulate the execution of j' on the (perhaps hypothetical) cluster (Section 6.5). Finally, the simulated execution is used to estimate the data properties for the derived dataset(s) produced by j' as well as the overall performance of j' (Section 6.6). The overall workflow performance is predicted by combining the performance predictions for each job in the workflow.

The virtual job profile contains the predicted timings and data-flow information of the job when run with the new parameter settings, similar to what a job profile generated by the Profiler contains. The purpose of the virtual profile is to provide the user with more insights on how each job will behave when using the new parameter settings, as well as to expand the uses of the What-if Engine.

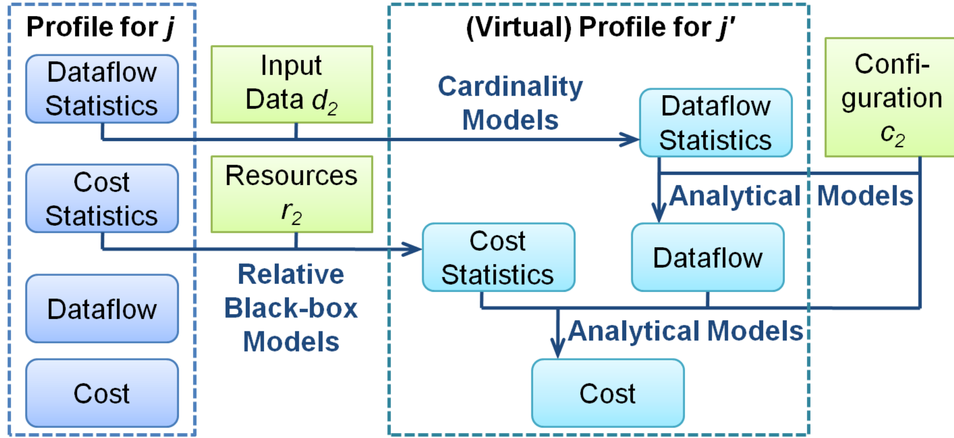


FIGURE 6.2: Overall process used by the What-if Engine to estimate a virtual job profile.

The process of virtual profile estimation forms the foundation on which Starfish's ability to answer tuning queries is based. Specifically, given the profile of a job $j = \langle p, d_1, r_1, c_1 \rangle$ and the properties of input data d_2, cluster resources r_2, and configuration parameter settings c_2 of a hypothetical job $j' = \langle p, d_2, r_2, c_2 \rangle$, the virtual profile of j' has to be estimated. Our solution for virtual profile estimation is based on a mix of black-box and white-box models. The overall estimation process has been broken down into smaller steps as shown in Figure 6.2, and a suitable modeling technique was picked for each step. These smaller steps correspond to the four categories of fields in a job profile. We use conventional cardinality (white-box) models from database query optimization to estimate dataflow statistics fields (described in Section 6.2), relative black-box models to estimate the cost statistics fields (described in Section 6.3), and new analytical (white-box) models that we developed to estimate the dataflow fields, and in turn, the cost fields (described in Section 6.4).

6.2 Cardinality Models to Estimate Dataflow Statistics Fields

Database query optimizers keep fine-grained, data-level statistics such as histograms to estimate the dataflow in execution plans for declarative queries. However, MapRe-

duce frameworks lack the declarative query semantics and structured data representations of Database systems. Thus, the common case in the What-if Engine is to not have detailed statistical information about the input data d_2 in the hypothetical job j'. By default, the What-if Engine makes a *dataflow proportionality assumption* which says that the logical dataflow sizes through the job's phases are proportional to the input data size. It follows from this assumption that the dataflow statistics fields (Table 4.3) in the virtual profile of j' will be the same as those in the profile of job j given as input.

When additional information is available, the What-if Engine allows the default assumption to be overridden by providing dataflow statistics fields of the virtual profile directly as input. For example, when higher semantic layers like Hive and Pig submit a MapReduce job for a computation like filtering or join, they can estimate dataflow statistics fields like Map and Reduce selectivity using conventional statistics like histograms. Researchers are also developing tools to extract detailed information from MapReduce programs through program analysis (Iu and Zwaenepoel, 2010; Cafarella and Ré, 2010).

6.3 Relative Black-box Models to Estimate Cost Statistics Fields

Consider the cost statistics fields shown in Table 4.4. Clusters with identical resources will have the same CPU and I/O (local and remote) costs. Thus, if the cluster resource properties of r_1 are the same as those of r_2, then the cost statistics fields in the hypothetical job j' virtual profile can be copied directly from the profile of job j given as input. This copying cannot be used when $r_1 \neq r_2$, in particular, when job j' will run on a *target cluster* containing nodes with a different type from the *source cluster* where job j was run to collect the profile that was given as input.

The technique we use when $r_1 \neq r_2$ is based on a *relative* black-box model $M_{src \rightarrow tgt}$ that can predict the cost statistics fields CS_{tgt} in the virtual profile for the target

cluster from the cost statistics fields CS_{src} in the job profile for the source cluster.

$$CS_{tgt} = M_{src \to tgt}(CS_{src}) \tag{6.3}$$

Generating training samples for the $M_{src \to tgt}$ model: Let r_{src} and r_{tgt} denote the cluster resources respectively for the source and target clusters. Suppose the MapReduce program p is run on input data d and configuration parameter settings c on both the source and target clusters. That is, we run the two jobs $j_{src} = \langle p, d, r_{src}, c \rangle$ and $j_{tgt} = \langle p, d, r_{tgt}, c \rangle$. From these runs, we can generate the profiles for these two jobs by direct measurement. Even further, recall from Section 4.3 that we can generate a separate task profile for each task run in each of these two jobs. Therefore, from the ith task in these two jobs,[1] we get a training sample $\langle \text{CS}^{(i)}_{src}, \text{CS}^{(i)}_{tgt}, \rangle$ for the $M_{src \to tgt}$ model.

The above training samples were generated by running a related pair of jobs j_{src} and j_{tgt} that have the same MapReduce program p, input data d, and configuration parameter settings c. We can generate a complete set of training samples by using a *training benchmark* containing jobs with different $\langle p, d, c \rangle$ combinations. Selecting an appropriate training benchmark is nontrivial because the two main requirements of effective black-box modeling have to be satisfied. First, for accurate prediction, the training samples must have good coverage of the prediction space. Second, for efficiency, the time to generate the complete set of training samples must be small.

Based on the coverage and efficiency considerations, we came up with three methods to select the training benchmark in Starfish: *Apriori*, *Fixed*, and *Custom*. We will evaluate these three benchmarks empirically in Section 6.7.

Apriori: This method assumes that the full workload of jobs that will run on the provisioned clusters is known at model training time. A sample of jobs is selected

[1] Ensuring that the tasks have the same input data d and configuration parameter settings c ensures that there is a one-to-one correspondence between the tasks in these two jobs.

from this workload, either randomly or from the top-k longest-running jobs. A similar approach of sampling the SQL query workload is used by the index selection and other wizards in most commercial Database systems (Chaudhuri et al., 2003). To improve the training efficiency, it may be possible to run the jobs on a scaled-down sample of their input data. However, this step requires domain knowledge or user assistance. Apriori gives good coverage of the prediction space as long as the assumption on the workload holds. However, Apriori's running time grows with the size of the workload and the input data.

Fixed: This method assumes that a good coverage of the prediction space can be obtained by selecting a predetermined set of existing MapReduce jobs (e.g., Sort, WordCount) and executing them using different configuration settings that will give different degrees of resource usage. For example, the benchmark can consist of CPU-intensive, CPU-light, I/O-intensive, and I/O-light jobs. The running time of Fixed is independent of the size of the actual workload and the input data.

Custom: The goal of this method is to execute a small, synthetic workload to generate training samples for cost statistics efficiently such that these samples will give good coverage of the prediction space. It is because of our abstraction of any MapReduce job execution as a job profile—where a profile can be represented as a point in a high-dimensional space (see Chapter 4)—that we are able to consider such a unique approach that is independent of the actual MapReduce workload run on the cluster.

Our Custom training benchmark is composed of just two synthetic MapReduce *job templates*: a data-generation template and a data-processing template. These two templates are instantiated in different ways for a total of six MapReduce job executions. Unlike the Fixed benchmark that consists of existing MapReduce jobs, the jobs generated by Custom have been designed such that the different tasks within these jobs behave differently in terms of their CPU, I/O, memory, and network usage.

While this approach may sound counterintuitive because the map (reduce) tasks in a job are expected to behave similarly, it produces more diverse training samples per job execution than Apriori or Fixed. Custom provides two additional advantages: (i) lower and more predictable running time for the training benchmark; and (ii) no knowledge or use of actual workloads and input data is needed during the training phase.

Learning all the $M_{src \to tgt}$ models needed: It is important to note that the training benchmark has to be run only once (or with a few repetitions) per target cluster resource; giving only a linear number of benchmark runs, and not quadratic as one might expect from the relative nature of the $M_{src \to tgt}$ models. The training samples for each source-to-target cluster pair is available from these runs. For example, to address use case 3 from Section 3.4 (i.e., planning for workload transition from a development cluster to production), one run each of the training benchmark on the development and the production cluster will suffice. For a more complex scenario like use case 4 (i.e., provisioning a cluster under multiple objectives) that involves different types of Amazon EC2 nodes, one benchmark run for each distinct node type and a representative number of cluster nodes is usually sufficient. If the workload or data size is expected to vary widely, then benchmark runs over a few different numbers of nodes in the cluster can improve prediction accuracy.

Once the training samples are generated, there are many *supervised learning* techniques available for generating the black-box model in Equation 6.3. Since cost statistics are real-valued, we selected the *M5 Tree Model* (Quinlan, 1992). An M5 Tree Model first builds a regression-tree using a typical decision-tree induction algorithm. Then, the tree goes through pruning and smoothing phases to generate a linear regression model for each leaf of the tree.

In summary, given the cost statistics fields CS_{src} in the job profile for the source

cluster r_{src}, the relative black-box model $M_{src \to tgt}$ is used to predict the cost statistics fields CS_{tgt} in the virtual profile for the target cluster r_{tgt} when $r_{src} \neq r_{tgt}$.

6.4 Analytical Models to Estimate Dataflow and Cost Fields

The What-if Engine uses a detailed set of analytical (white-box) models that describes the execution of a MapReduce job on a Hadoop cluster. The models calculate the dataflow fields (Table 4.1) and cost fields (Table 4.2) in a virtual job profile. The inputs required by the models are (i) the estimated dataflow statistics fields in the virtual job profile (Table 4.3), (ii) the estimated cost statistics fields in the virtual job profile (Table 4.4), and (iii) cluster-wide and job-level configuration parameter settings (Table 6.2). These models give good accuracy by capturing the subtleties of MapReduce job execution at the fine granularity of phases within map and reduce tasks. The current models were developed for Hadoop, but the overall approach applies to any MapReduce implementation.

Preliminaries: To simplify the notation, we use the abbreviations contained in Tables 4.1, 4.2, 4.3, 4.4, and 6.2. Note the prefixes in all abbreviations used to distinguish where each abbreviation belongs to: *d* for dataset fields, *c* for cost fields, *ds* for data statistics fields, *cs* for cost statistics fields, *p* for Hadoop parameters, and *t* for temporary information not stored in the profile.

In an effort to present concise formulas and avoid the use of conditionals as much as possible, we make the following definitions and initializations:

$$\text{Identity Function} \quad I(x) = \begin{cases} 1 \text{ , if x exists or equals true} \\ 0 \text{ , otherwise} \end{cases} \tag{6.4}$$

Table 6.2: A subset of cluster-wide and job-level Hadoop parameters.

Abbreviation	Hadoop Parameter	Default Value
pNumNodes	Number of Nodes	
pTaskMem	mapred.child.java.opts	-Xmx200m
pMaxMapsPerNode	mapred.tasktracker.map.tasks.max	2
pMaxRedsPerNode	mapred.tasktracker.reduce.tasks.max	2
pNumMappers	mapred.map.tasks	
pSortMB	io.sort.mb	100 MB
pSpillPerc	io.sort.spill.percent	0.8
pSortRecPerc	io.sort.record.percent	0.05
pSortFactor	io.sort.factor	10
pNumSpillsForComb	min.num.spills.for.combine	3
pNumReducers	mapred.reduce.tasks	
pReduceSlowstart	mapred.reduce.slowstart.completed.maps	0.05
pInMemMergeThr	mapred.inmem.merge.threshold	1000
pShuffleInBufPerc	mapred.job.shuffle.input.buffer.percent	0.7
pShuffleMergePerc	mapred.job.shuffle.merge.percent	0.66
pReducerInBufPerc	mapred.job.reduce.input.buffer.percent	0
pUseCombine	mapred.combine.class or mapreduce.combine.class	null
pIsIntermCompressed	mapred.compress.map.output	false
pIsOutCompressed	mapred.output.compress	false
pIsInCompressed	Whether the input is compressed or not	
pSplitSize	The size of the input split	

If ($pUseCombine$ == FALSE)

$dsCombineSizeSel = 1$

$dsCombineRecsSel = 1$

$csCombineCPUCost = 0$

If ($pIsInCompressed$ == FALSE)

$dsInputCompressRatio = 1$

$csInUncomprCPUCost = 0$

If ($pIsIntermCompressed$ == FALSE)

$dsIntermCompressRatio = 1$

$csIntermUncomCPUCost = 0$

$csIntermComCPUCost = 0$

If ($pIsOutCompressed$ == FALSE)

$dsOutCompressRatio = 1$

$csOutComprCPUCost = 0$

MapReduce job execution phases: The Map task execution is divided into five phases:

1. *Read*: Reading the input split from HDFS and creating the input key-value pairs (records).

2. *Map*: Executing the user-defined map function to generate the map-output data.

3. *Collect*: Partitioning and collecting the intermediate (map-output) data into a buffer before spilling.

4. *Spill*: Sorting, using the combine function if any, performing compression if specified, and finally writing to local disk to create *file spills*.

5. *Merge*: Merging the file spills into a single map output file. Merging might be performed in multiple rounds.

The Reduce Task is divided into four phases:

1. *Shuffle*: Transferring the intermediate data from the mapper nodes to a reducer's node and decompressing if needed. Partial merging may also occur during this phase.

2. *Merge*: Merging the sorted fragments from the different mappers to form the input to the reduce function.

3. *Reduce*: Executing the user-defined reduce function to produce the final output data.

4. *Write*: Compressing, if specified, and writing the final output to HDFS.

We model all task phases in order to accurately calculate the dataflow and cost information for the new (hypothetical) MapReduce job execution. For a map task, we model the Read and Map phases in Section 6.4.1, the Collect and Spill phases in Section 6.4.2, and the Merge phase in Section 6.4.3. For a reduce task, we model the Shuffle phase in Section 6.4.4, the Merge phase in Section 6.4.5, and the Reduce and Write phases in Section 6.4.6.

6.4.1 Modeling the Read and Map Phases in the Map Task

During this phase, the input split is read (and uncompressed if necessary) and the key-value pairs are created and passed as input to the user-defined map function.

$$dMapInBytes = \frac{pSplitSize}{dsInputCompressRatio} \tag{6.5}$$

$$dMapInRecs = \frac{dMapInBytes}{dsInputPairWidth} \tag{6.6}$$

The cost of the Map Read phase is:

$$\begin{aligned} cReadPhaseTime = {} & pSplitSize \times csHdfsReadCost \\ & + pSplitSize \times csInUncomprCPUCost \end{aligned} \tag{6.7}$$

The cost of the Map phase is:

$$cMapPhaseTime = dMapInRecs \times csMapCPUCost \tag{6.8}$$

If the MapReduce job consists only of mappers (i.e., $pNumReducers = 0$), then the spilling and merging phases will not be executed and the map output will be written directly to HDFS.

$$dMapOutBytes = dMapInBytes \times dsMapSizeSel \tag{6.9}$$

$$dMapOutRecs = dMapInRecs \times dsMapRecsSel \tag{6.10}$$

The cost of the Map Write phase is:

$$\begin{aligned} cWritePhaseTime = & \\ & dMapOutBytes \times csOutComprCPUCost \\ & + dMapOutBytes \times dsOutCompressRatio \times csHdfsWriteCost \end{aligned} \tag{6.11}$$

6.4.2 Modeling the Collect and Spill Phases in the Map Task

The map function generates output key-value pairs (records) that are placed in the map-side memory buffer of size *pSortMB*. The amount of data output by the map function is calculated as follows:

$$dMapOutBytes = dMapInBytes \times dsMapSizeSel \tag{6.12}$$

$$dMapOutRecs = dMapInRecs \times dsMapRecsSel \tag{6.13}$$

$$tMapOutRecWidth = \frac{dMapOutBytes}{dMapOutRecs} \tag{6.14}$$

The map-side buffer consists of two disjoint parts: the *serialization* part that stores the serialized map-output records, and the *accounting* part that stores 16 bytes of metadata per record. When either of these two parts fills up to the threshold determined by *pSpillPerc*, the spill process begins. The maximum number of records in the serialization buffer before a spill is triggered is:

$$tMaxSerRecs = \left\lfloor \frac{pSortMB \times 2^{20} \times (1 - pSortRecPerc) \times pSpillPerc}{tMapOutRecWidth} \right\rfloor \quad (6.15)$$

The maximum number of records in the accounting buffer before a spill is triggered is:

$$tMaxAccRecs = \left\lfloor \frac{pSortMB \times 2^{20} \times pSortRecPerc \times pSpillPerc}{16} \right\rfloor \quad (6.16)$$

Hence, the number of records in the buffer before a spill is:

$$dSpillBufferRecs = Min\{\ tMaxSerRecs,\ tMaxAccRecs\ ,\ dMapOutRecs\ \} \quad (6.17)$$

The size of the buffer included in a spill is:

$$dSpillBufferSize = dSpillBufferRecs \times tMapOutRecWidth \quad (6.18)$$

The overall number of spills is:

$$dNumSpills = \left\lceil \frac{dMapOutRecs}{dSpillBufferRecs} \right\rceil \quad (6.19)$$

The number of pairs and size of each spill file (i.e., the amount of data that will be written to disk) depend on the width of each record, the possible use of the Combiner, and the possible use of compression. The Combiner's pair and size selectivities as well as the compression ratio are part of the Dataflow Statistics fields of the job profile. If a Combiner is not used, then the corresponding selectivities are set to 1 by default. If map output compression is disabled, then the compression ratio is set to 1.

Hence, the number of records and size of a spill file are:

$$dSpillFileRecs = dSpillBufferRecs \times dsCombineRecsSel \quad (6.20)$$

$$\begin{aligned} dSpillFileSize = & dSpillBufferSize \times dsCombineSizeSel \\ & \times dsIntermCompressRatio \end{aligned} \tag{6.21}$$

The total cost of the Map's Collect and Spill phases are:

$$\begin{aligned} cCollectPhaseTime = & dMapOutRecs \times csPartitionCPUCost \\ & + dMapOutRecs \times csSerdeCPUCost \end{aligned} \tag{6.22}$$

$$\begin{aligned} cSpillPhaseTime = & \ dNumSpills \times \\ & [\ dSpillBufferRecs \times \log_2(\frac{dSpillBufferRecs}{pNumReducers}) \times csSortCPUCost \\ & + dSpillBufferRecs \times csCombineCPUCost \\ & + dSpillBufferSize \times dsCombineSizeSel \times csIntermComCPUCost \\ & + dSpillFileSize \times csLocalIOWriteCost\ ] \end{aligned} \tag{6.23}$$

6.4.3 Modeling the Merge Phase in the Map Task

The goal of the Merge phase is to merge all the spill files into a single output file, which is written to local disk. The Merge phase will occur only if more than one spill file is created. Multiple merge passes might occur, depending on the *pSortFactor* parameter. *pSortFactor* defines the maximum number of spill files that can be merged together to form a new single file. We define a *merge pass* to be the merging of at most *pSortFactor* spill files. We define a *merge round* to be one or more merge passes that merge only spills produced by the spill phase or a previous merge round. For example, suppose $dNumSpills = 28$ and $pSortFactor = 10$. Then, 2 merge passes will be performed (merging 10 files each) to create 2 new files. This constitutes the first merge round. Then, the 2 new files will be merged together with the 8 original spill files to create the final output file, forming the 2nd and final merge round.

The first merge pass is unique because Hadoop will calculate the optimal number of spill files to merge so that all other merge passes will merge exactly *pSortFactor*

files. Notice how, in the example above, the final merge round merged exactly 10 files.

The final merge pass is also unique in the sense that if the number of spills to be merged is greater than or equal to *pNumSpillsForComb*, the combiner will be used again. Hence, we treat the intermediate merge rounds and the final merge round separately. For the intermediate merge passes, we calculate how many times (on average) a single spill will be read.

Note that the remaining section assumes $numSpils \leqslant pSortFactor^2$. In the opposite case, we must use a simulation-based approach in order to calculate the number of spill files merged during the intermediate merge rounds as well as the total number of merge passes. Since the Reduce task also contains a similar Merge Phase, we define the following three methods to reuse later:

$$calcNumSpillsFirstPass(N, F) =$$

$$\begin{cases} N & \text{, if } N \leqslant F \\ F & \text{, if } (N-1)\ MOD\ (F-1) = 0 \\ (N-1)\ MOD\ (F-1) + 1 & \text{, otherwise} \end{cases} \quad (6.24)$$

$$calcNumSpillsIntermMerge(N, F) =$$

$$\begin{cases} 0 & \text{, if } N \leqslant F \\ P + \left\lfloor \frac{N-P}{F} \right\rfloor * F & \text{, if } N \leqslant F^2 \end{cases}$$

$$\text{, where } P = calcNumSpillsFirstPass(N, F) \quad (6.25)$$

$$calcNumSpillsFinalMerge(N, F) =$$

$$\begin{cases} N & \text{, if } N \leqslant F \\ 1 + \left\lfloor \frac{N-P}{F} \right\rfloor + (N-S) & \text{, if } N \leqslant F^2 \end{cases}$$

$$\text{, where } P = calcNumSpillsFirstPass(N, F)$$

$$\text{, where } S = calcNumSpillsIntermMerge(N, F) \quad (6.26)$$

The number of spills read during the first merge pass is:

$$tNumSpillsFirstPass = calcNumSpillsFirstPass(dNumSpills, pSortFactor) \quad (6.27)$$

The number of spills read during intermediate merging is:

$$tNumSpillsIntermMerge = calcNumSpillsIntermMerge(dNumSpills, pSortFactor) \quad (6.28)$$

The total number of merge passes is:

$$dNumMergePasses = \begin{cases} 0 & \text{, if } dNumSpills = 1 \\ 1 & \text{, if } dNumSpills \leqslant pSortFactor \\ 2 + \left\lfloor \frac{dNumSpills - tNumSpillsFirstPass}{pSortFactor} \right\rfloor & \text{, if } dNumSpills \leqslant pSortFactor^2 \end{cases} \quad (6.29)$$

The number of spill files for the final merge round is:

$$tNumSpillsFinalMerge = calcNumSpillsFinalMerge(dNumSpills, pSortFactor) \quad (6.30)$$

As discussed earlier, the Combiner might be used during the final merge round. In this case, the size and record Combiner selectivities are:

$$tUseCombInMerge = (dNumSpills > 1) \text{ AND } (pUseCombine) \text{ AND } (tNumSpillsFinalMerge \geqslant pNumSpillsForComb) \quad (6.31)$$

$$tMergeCombSizeSel = \begin{cases} dsCombineSizeSel & \text{, if } tUseCombInMerge \\ 1 & \text{, otherwise} \end{cases} \quad (6.32)$$

$$tMergeCombRecsSel = \begin{cases} dsCombineRecsSel & \text{, if } tUseCombInMerge \\ 1 & \text{, otherwise} \end{cases} \quad (6.33)$$

The total number of records spilled equals the sum of (i) the records spilled during the Spill phase, (ii) the number of records that participated in the intermediate merge rounds, and (iii) the number of records spilled during the final merge round.

$$\begin{aligned} dNumRecsSpilled = & dSpillFileRecs \times dNumSpills \\ & + dSpillFileRecs \times tNumSpillsIntermMerge \\ & + dSpillFileRecs \times dNumSpills \times tMergeCombRecsSel \end{aligned} \quad (6.34)$$

The final size and number of records for the final map output data are:

$$tIntermDataSize = dNumSpills \times dSpillFileSize \times tMergeCombSizeSel \quad (6.35)$$

$$tIntermDataRecs = dNumSpills \times dSpillFileRecs \times tMergeCombRecsSel \quad (6.36)$$

The total cost of the Merge phase is divided into the cost for performing the intermediate merge rounds and the cost for performing the final merge round.

$$\begin{aligned} tIntermMergeTime = & tNumSpillsIntermMerge \times \\ & [\ dSpillFileSize \times csLocalIOReadCost \\ & + dSpillFileSize \times csIntermUncomCPUCost \\ & + dSpillFileRecs \times csMergeCPUCost \\ & + \frac{dSpillFileSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\ & + dSpillFileSize \times csLocalIOWriteCost\ ] \end{aligned} \quad (6.37)$$

$$
\begin{aligned}
tFinalMergeTime = & dNumSpills \times \\
& [\ dSpillFileSize \times csLocalIOReadCost \\
& \quad + dSpillFileSize \times csIntermUncomCPUCost \\
& \quad + dSpillFileRecs \times csMergeCPUCost \\
& \quad + dSpillFileRecs \times csCombineCPUCost\ ] \\
& + \frac{tIntermDataSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\
& + tIntermDataSize \times csLocalIOWriteCost
\end{aligned} \tag{6.38}
$$

$$
cMergePhaseTime = tIntermMergeTime + tFinalMergeTime \tag{6.39}
$$

6.4.4 Modeling the Shuffle Phase in the Reduce Task

In the Shuffle phase, the framework fetches the relevant map output partition from each mapper (called a *map segment*) and copies it to the reducer's node. If the map output is compressed, Hadoop will uncompress it after the transfer as part of the shuffling process. Assuming a uniform distribution of the map output to all reducers, the size and number of records for each map segment that reaches the reduce side are:

$$
tSegmentComprSize = \frac{tIntermDataSize}{pNumReducers} \tag{6.40}
$$

$$
tSegmentUncomprSize = \frac{tSegmentComprSize}{dsIntermCompressRatio} \tag{6.41}
$$

$$
tSegmentRecs = \frac{tIntermDataRecs}{pNumReducers} \tag{6.42}
$$

where *tIntermDataSize* and *tIntermDataRecs* are the size and number of records produced as intermediate output by a single mapper (see Section 6.4.3). A more complex model can be used to account for the presence of skew. The data fetched to a single reducer will be:

$$dShuffleSize = pNumMappers * tSegmentComprSize \tag{6.43}$$

$$dShuffleRecs = pNumMappers * tSegmentRecs \tag{6.44}$$

The intermediate data is transfered and placed in an in-memory *shuffle buffer* with a size proportional to the parameter *pShuffleInBufPerc*:

$$tShuffleBufferSize = pShuffleInBufPerc \times pTaskMem \tag{6.45}$$

However, when the segment size is greater than 25% times the *tShuffleBufferSize*, the segment will get copied directly to local disk instead of the in-memory shuffle buffer. We consider these two cases separately.

Case 1: $tSegmentUncomprSize < 0.25 \times tShuffleBufferSize$

The map segments are transfered, uncompressed if needed, and placed into the shuffle buffer. When either (a) the amount of data placed in the shuffle buffer reaches a threshold size determined by the *pShuffleMergePerc* parameter or (b) the number of segments becomes greater than the *pInMemMergeThr* parameter, the segments are merged and spilled to disk creating a new local file (called *shuffle file*). The size threshold to begin merging is:

$$tMergeSizeThr = pShuffleMergePerc \times tShuffleBufferSize \tag{6.46}$$

The number of map segments merged into a single shuffle file is:

$$tNumSegInShuffleFile = \frac{tMergeSizeThr}{tSegmentUncomprSize} \tag{6.47}$$

If $(\lceil tNumSegInShuffleFile \rceil \times tSegmentUncomprSize \leqslant tShuffleBufferSize)$

$tNumSegInShuffleFile = \lceil tNumSegInShuffleFile \rceil$

else

$tNumSegInShuffleFile = \lfloor tNumSegInShuffleFile \rfloor$

If $(tNumSegInShuffleFile > pInMemMergeThr)$

$$tNumSegInShuffleFile = pInMemMergeThr \tag{6.48}$$

If a Combiner is specified, then it is applied during the merging. If compression is enabled, then the (uncompressed) map segments are compressed after merging and before written to disk. Note also that if $numMappers < tNumSegInShuffleFile$, then merging will not happen. The size and number of records in a single shuffle file is:

$$tShuffleFileSize = tNumSegInShuffleFile \times tSegmentComprSize \times dsCombineSizeSel \tag{6.49}$$

$$tShuffleFileRecs = tNumSegInShuffleFile \times tSegmentRecs \times dsCombineRecsSel \tag{6.50}$$

$$tNumShuffleFiles = \left\lfloor \frac{pNumMappers}{tNumSegInShuffleFile} \right\rfloor \tag{6.51}$$

At the end of the merging process, some segments might remain in memory.

$$tNumSegmentsInMem = pNumMappers \text{ MOD } tNumSegInShuffleFile \tag{6.52}$$

Case 2: $tSegmentUncomprSize \geqslant 0.25 \times tShuffleBufferSize$

When a map segment is transfered directly to local disk, it becomes equivalent to a shuffle file. Hence, the corresponding temporary variables introduced in Case 1 above are:

$$tNumSegInShuffleFile = 1 \tag{6.53}$$

$$tShuffleFileSize = tSegmentComprSize \tag{6.54}$$

$$tShuffleFileRecs = tSegmentRecs \tag{6.55}$$

$$tNumShuffleFiles = pNumMappers \tag{6.56}$$

$$tNumSegmentsInMem = 0 \tag{6.57}$$

Either case can create a set of shuffle files on disk. When the number of shuffle files on disk increases above a certain threshold (which equals $2 \times pSortFactor - 1$), a new merge thread is triggered and *pSortFactor* shuffle files are merged into a new and larger sorted shuffle file. The Combiner is not used during this so-called *disk merging*. The total number of such disk merges are:

$$tNumShuffleMerges =$$

$$\begin{cases} 0 \text{ , if } tNumShuffleFiles < 2 \times pSortFactor - 1 \\ \left\lfloor \frac{tNumShuffleFiles - 2 \times pSortFactor + 1}{pSortFactor} \right\rfloor + 1 \text{ , otherwise} \end{cases} \tag{6.58}$$

At the end of the Shuffle phase, a set of "merged" and "unmerged" shuffle files will exist on disk.

$$tNumMergShufFiles = tNumShuffleMerges \tag{6.59}$$

$$tMergShufFileSize = pSortFactor \times tShuffleFileSize \tag{6.60}$$

$$tMergShufFileRecs = pSortFactor \times tShuffleFileRecs \tag{6.61}$$

$$\begin{aligned} tNumUnmergShufFiles = & tNumShuffleFiles \\ & -(pSortFactor \times tNumShuffleMerges) \end{aligned} \tag{6.62}$$

$$tUnmergShufFileSize = tShuffleFileSize \tag{6.63}$$

$$tUnmergShufFileRecs = tShuffleFileRecs \tag{6.64}$$

The total cost of the Shuffle phase includes cost for the network transfer, cost for any in-memory merging, and cost for any on-disk merging, as described above.

$$\begin{aligned} & tInMemMergeTime = \\ & \quad I(tSegmentUncomprSize < 0.25 \times tShuffleBufferSize) \times \\ & \quad [\ dShuffleSize \times csIntermUncomCPUCost \\ & \quad\ +tNumShuffleFiles \times tShuffleFileRecs \times csMergeCPUCost \\ & \quad\ +tNumShuffleFiles \times tShuffleFileRecs \times csCombineCPUCost \\ & \quad\ +tNumShuffleFiles \times \frac{tShuffleFileSize}{dsIntermCompressRatio} \times csIntermComCPUCost\ ] \\ & \quad +tNumShuffleFiles \times tShuffleFileSize \times csLocalIOWriteCost \end{aligned} \tag{6.65}$$

$$\begin{aligned} & tOnDiskMergeTime = tNumMergShufFiles\ \times \\ & \quad [\ tMergShufFileSize \times csLocalIOReadCost \\ & \quad\ +tMergShufFileSize \times csIntermUncomCPUCost \\ & \quad\ +tMergShufFileRecs \times csMergeCPUCost \\ & \quad\ +\frac{tMergShufFileSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\ & \quad\ +tMergShufFileSize \times csLocalIOWriteCost\ ] \end{aligned} \tag{6.66}$$

$$
\begin{aligned}
cShufflePhaseTime = & dShuffleSize \times csNetworkCost \\
& + tInMemMergeTime \\
& + tOnDiskMergeTime
\end{aligned} \tag{6.67}
$$

6.4.5 Modeling the Merge Phase in the Reduce Task

After all map output data has been successful transfered to the Reduce node, the Merge phase[2] begins. During this phase, the map output data is merged into a single stream that is fed to the reduce function for processing. Similar to the Map's Merge phase (see Section 6.4.3), the Reduce's Merge phase may occur it multiple rounds. However, instead of creating a single output file during the final merge round, the data is sent directly to the reduce function.

The Shuffle phase may produce (i) a set of merged shuffle files on disk, (ii) a set of unmerged shuffle files on disk, and (iii) a set of map segments in memory. The total number of shuffle files on disk is:

$$
tNumShufFilesOnDisk = tNumMergShufFiles + tNumUnmergShufFiles \tag{6.68}
$$

The merging in this phase is done in three steps.

Step 1: Some map segments are marked for eviction from memory in order to satisfy a memory constraint enforced by the *pReducerInBufPerc* parameter, which specifies the amount of memory allowed to be occupied by the map segments before the reduce function begins.

$$
tMaxSegmentBufferSize = pReducerInBufPerc \times pTaskMem \tag{6.69}
$$

[2] The Merge phase in the Reduce task is also called "Sort phase" in the literature, even though no sorting occurs.

The amount of memory currently occupied by map segments is:

$$tCurrSegmentBufferSize = tNumSegmentsInMem \times tSegmentUncomprSize \tag{6.70}$$

Hence, the number of map segments to evict from, and retain in, memory are:

If ($tCurrSegmentBufferSize > tMaxSegmentBufferSize$)

$$tNumSegmentsEvicted = \left\lceil \frac{tCurrSegmentBufferSize - tMaxSegmentBufferSize}{tSegmentUncomprSize} \right\rceil$$

else

$$tNumSegmentsEvicted = 0 \tag{6.71}$$

$$tNumSegmentsRemainMem = tNumSegmentsInMem - tNumSegmentsEvicted \tag{6.72}$$

If the number of existing shuffle files on disk is less than *pSortFactor*, then the map segments marked for eviction will be merged into a single shuffle file on disk. Otherwise, the map segments marked for eviction are left to be merged with the shuffle files on disk during Step 2 (i.e., Step 1 does not happen).

If ($tNumShufFilesOnDisk < pSortFactor$)

$tNumShufFilesFromMem = 1$

$tShufFilesFromMemSize = tNumSegmentsEvicted \times tSegmentComprSize$

$tShufFilesFromMemRecs = tNumSegmentsEvicted \times tSegmentRecs$

$tStep1MergingSize = tShufFilesFromMemSize$

$tStep1MergingRecs = tShufFilesFromMemRecs$

else

$tNumShufFilesFromMem = tNumSegmentsEvicted$

$tShufFilesFromMemSize = tSegmentComprSize$

$tShufFilesFromMemRecs = tSegmentRecs$

$tStep1MergingSize = 0$

$$tStep1MergingRecs = 0 \tag{6.73}$$

The total cost of Step 1 (which could be zero) is:

$$\begin{aligned} cStep1Time = & tStep1MergingRecs \times csMergeCPUCost \\ & + \frac{tStep1MergingSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\ & + tStep1MergingSize \times csLocalIOWriteCost \end{aligned} \tag{6.74}$$

Step 2: Any shuffle files that reside on disk will go through a merging phase in multiple merge rounds (similar to the process in Section 6.4.3). This step will happen only if there exists at least one shuffle file on disk. The total number of files to merge during Step 2 is:

$$tFilesToMergeStep2 = tNumShufFilesOnDisk + tNumShufFilesFromMem \tag{6.75}$$

The number of intermediate reads (and writes) are:

$$tIntermMergeReads2 = calcNumSpillsIntermMerge(tFilesToMergeStep2, pSortFactor) \quad (6.76)$$

The main difference from Section 6.4.3 is that the merged files in this case have different sizes. We account for the different sizes by attributing merging costs proportionally. Hence, the total size and number of records involved in the merging process during Step 2 are:

$$\begin{aligned} tStep2MergingSize = & \frac{tIntermMergeReads2}{tFilesToMergeStep2} \times \\ & [\ tNumMergShufFiles \times tMergShufFileSize \\ & +tNumUnmergShufFiles \times tUnmergShufFileSize \\ & +tNumShufFilesFromMem \times tShufFilesFromMemSize] \end{aligned} \quad (6.77)$$

$$\begin{aligned} tStep2MergingRecs = & \frac{tIntermMergeReads2}{tFilesToMergeStep2} \times \\ & [\ tNumMergShufFiles \times tMergShufFileRecs \\ & +tNumUnmergShufFiles \times tUnmergShufFileRecs \\ & +tNumShufFilesFromMem \times tShufFilesFromMemRecs] \end{aligned} \quad (6.78)$$

The total cost of Step 2 (which could also be zero) is:

$$\begin{aligned} cStep2Time = & tStep2MergingSize \times csLocalIOReadCost \\ & +tStep2MergingSize \times cIntermUnomprCPUCost \\ & +tStep2MergingRecs \times csMergeCPUCost \\ & +\frac{tStep2MergingSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\ & +tStep2MergingSize \times csLocalIOWriteCost \end{aligned} \quad (6.79)$$

Step 3: All files on disk and in memory will be merged together. The process is identical to step 2 above. The total number of files to merge during Step 3 is:

$$\begin{aligned} tFilesToMergeStep3 &= tNumSegmentsRemainMem \\ &\quad + calcNumSpillsFinalMerge(tFilesToMergeStep2, pSortFactor) \end{aligned} \quad (6.80)$$

The number of intermediate reads (and writes) are:

$$\begin{aligned} & tIntermMergeReads3 = \\ & \quad calcNumSpillsIntermMerge(tFilesToMergeStep3, pSortFactor) \end{aligned} \quad (6.81)$$

Hence, the total size and number of records involved in the merging process during Step 3 are:

$$tStep3MergingSize = \frac{tIntermMergeReads3}{tFilesToMergeStep3} \times dShuffleSize \quad (6.82)$$

$$tStep3MergingRecs = \frac{tIntermMergeReads3}{tFilesToMergeStep3} \times dShuffleRecs \quad (6.83)$$

The total cost of Step 3 (which could also be zero) is:

$$\begin{aligned} cStep3Time = & tStep3MergingSize \times csLocalIOReadCost \\ & + tStep3MergingSize \times cIntermUnomprCPUCost \\ & + tStep3MergingRecs \times csMergeCPUCost \\ & + \frac{tStep3MergingSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\ & + tStep3MergingSize \times csLocalIOWriteCost \end{aligned} \quad (6.84)$$

The total cost of the Merge phase is:

$$cMergePhaseTime = cStep1Time + cStep2Time + cStep3Time \quad (6.85)$$

6.4.6 Modeling the Reduce and Write Phases in the Reduce Task

Finally, the user-defined reduce function will processed the merged intermediate data to produce the final output that will be written to HDFS. The size and number of records processed by the reduce function is:

$$\begin{aligned} dReduceInBytes = & \frac{tNumShuffleFiles \times tShuffleFileSize}{dsIntermCompressRatio} \\ & + \frac{tNumSegmentsInMem \times tSegmentComprSize}{dsIntermCompressRatio} \end{aligned} \tag{6.86}$$

$$\begin{aligned} dReduceInRecs = & tNumShuffleFiles \times tShuffleFileRecs \\ & + tNumSegmentsInMem \times tSegmentRecs \end{aligned} \tag{6.87}$$

The size and number of records produce by the reduce function is:

$$dReduceOutBytes = dReduceInBytes \times dsReduceSizeSel \tag{6.88}$$

$$dReduceOutRecs = dReduceInRecs \times dsReduceRecsSel \tag{6.89}$$

The input data to the reduce function may reside in both memory and disk, as produced by the Shuffle and Merge phases.

$$\begin{aligned} tInRedFromDiskSize = & tNumMergShufFiles \times tMergShufFileSize \\ & + tNumUnmergShufFiles \times tUnmergShufFileSize \\ & + tNumShufFilesFromMem \times tShufFilesFromMemSize \end{aligned} \tag{6.90}$$

The total cost of the Reduce phase is:

$$\begin{aligned} cReducePhaseTime = & tInRedFromDiskSize \times csLocalIOReadCost \\ & + tInRedFromDiskSize \times cIntermUncompCPUCost \\ & + dReduceInRecs \times csReduceCPUCost \end{aligned} \quad (6.91)$$

The total cost of the Write phase is:

$$\begin{aligned} & cWritePhaseTime = \\ & \quad dReduceOutBytes \times csOutComprCPUCost \\ & \quad + dReduceOutBytes \times dsOutCompressRatio \times csHdfsWriteCost \end{aligned} \quad (6.92)$$

6.5 Simulating the Execution of a MapReduce Workload

The virtual job profile contains detailed dataflow and cost information estimated at the task and phase level for the hypothetical job j'. The What-if Engine uses a *Task Scheduler Simulator*, along with the job profiles and information on the cluster resources, to simulate the scheduling and execution of map and reduce tasks in j' (recall Figure 6.1). The Task Scheduler Simulator is a pluggable component. Our current implementation is a lightweight discrete event simulation of Hadoop's default FIFO scheduler. For instance, a job with 60 tasks to be run on a 16-node cluster can be simulated in 0.3 milliseconds.

The Task Scheduler Simulator is aware of both dataflow and resource dependencies among the MapReduce jobs in a workflow. Therefore, when it is given a new hypothetical job j' to simulate, it will schedule it (a) concurrently with existing jobs that have a resource dependency with j', and (b) after all jobs that have a dataflow dependency with j'. The output from the simulation is a complete description of the (hypothetical) execution of job j' in the cluster.

Existing approaches to simulating Hadoop execution: *Mumak* (Tang, 2009) and *MRPerf* (Wang et al., 2009) are existing Hadoop simulators that perform discrete

event simulation to model MapReduce job execution. Mumak needs a job execution trace from a previous job execution as input. Unlike our What-if Engine, Mumak cannot simulate job execution for a different cluster size, network topology, or even different numbers of map or reduce tasks from what the execution trace contains.

MRPerf is able to simulate job execution at the task level like our What-if Engine. However, MRPerf uses an external network simulator to simulate the data transfers and communication among the cluster nodes; which leads to a per-job simulation time on the order of minutes. Such a high simulation overhead prohibits MRPerf's use by a cost-based optimizer that needs to perform hundreds to thousands of what-if calls per job.

6.6 Estimating Derived Data Properties and Workflow Performance

After the execution of a job is simulated on the cluster, the What-if Engine estimates the properties of the job's derived datasets (see Figure 6.1). For this purpose, we have implemented a *Virtual Distributed File System (DFS)* to keep track of file and block metadata. The estimated dataflow fields in the virtual profile are used, along with analytical models, to estimate data properties like the number of files produced, the file sizes, and whether the files are compressed or not. The simulated task execution is then used to determine the block placement of the data in the Virtual DFS. The estimated properties of the derived datasets will be used during the virtual profile estimation of the later jobs in the workflow that access these datasets as input.

The final output—after all jobs have been simulated—is a description of the complete (hypothetical) workflow execution in the cluster. The desired answer to the what-if question—e.g., predicted workflow running time, amount of local I/O, or a visualization of the task execution timeline—can be computed from the workflow's simulated execution.

Table 6.3: Cluster-wide Hadoop parameter settings for five EC2 node types.

EC2 Node Type	Map Slots per Node	Reduce Slots per Node	Max Memory per slot (MB)
m1.small	2	1	300
m1.large	3	2	1024
m1.xlarge	4	4	1536
c1.medium	2	2	300
c1.xlarge	8	6	400

Table 6.4: MapReduce programs and corresponding datasets for the evaluation of the What-if Engine.

Abbr.	MapReduce Program	Dataset Description
CO	Word Co-occurrence	10GB of documents from Wikipedia
JO	Join	60GB data from the TPC-H Benchmark
LG	LinkGraph	20GB compressed data from Wikipedia
TF	TF-IDF	60GB of documents from Wikipedia
TS	Hadoop's TeraSort	30GB-60GB data from Hadoop's TeraGen
WC	WordCount	30GB-60GB of documents from Wikipedia

6.7 Evaluating the Predictive Power of the What-if Engine

In our experimental evaluation, we used Hadoop clusters running on Amazon EC2 nodes of various sizes and node types. Table 3.2 lists the EC2 node types we used, along with the resources available for each node type. For each node type, we used empirically-determined fixed values for the cluster-wide Hadoop configuration parameters—namely, the number of map and reduce task execution slots per node, as well as the maximum memory available per task slot (shown on Table 6.3).

Table 6.4 lists the MapReduce programs and datasets used in our evaluation. We selected representative MapReduce programs used in different domains: text analytics (WordCount), natural language processing (Word Co-occurrence), information retrieval (TF-IDF: Term Frequency-Inverse Document Frequency[3]), creation of large hyperlink graphs (LinkGraph), and business data processing (Join, TeraSort) (Lin

[3] TF-IDF is a workflow consisting of three MapReduce jobs.

and Dyer, 2010; White, 2010).

The unoptimized MapReduce jobs are executed using popular *rules-of-thumb* settings. Following rules of thumb has become a standard technique for setting job configuration parameter settings (Hadoop Tutorial, 2011; Lipcon, 2009; White, 2010). The rules of thumb are discussed further in Section 7.1. Unless otherwise noted, we used the training samples produced by the Custom benchmark to train the relative models for estimating cost statistics (recall Section 6.3).

The goal of the experimental evaluation is to study the ability of the What-if Engine to provide reliable estimates for the execution time of MapReduce jobs in various scenarios. Our evaluation methodology is as follows:

- We evaluate the What-if Engine's accuracy in estimating the sub-task timings, as well as predicting the overall job completion time.
- We evaluate the predictive power of the What-if Engine for tuning the cluster size for elastic MapReduce workloads.
- We evaluate the accuracy of the What-if Engine in estimating the execution time for a program p to be run on the production cluster (r_{tgt}) based on a profile learned for p on the development cluster (r_{src}).
- We evaluate the accuracy of the relative models learned for predicting cost statistics for the three training benchmarks developed to generate training samples (recall Section 6.3).

Since our evaluation concentrates on the What-if Engine, we focus on the job running times and ignore any data loading times.

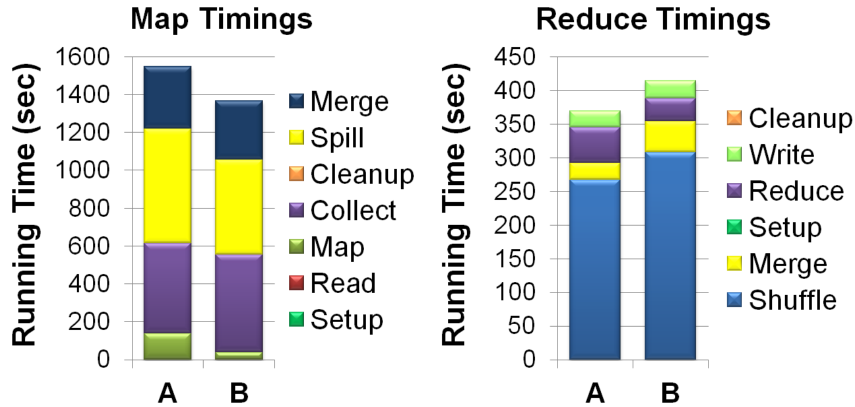


FIGURE 6.3: Map and reduce time breakdown for Word Co-occurrence jobs from (A) an actual run and (B) as predicted by the What-if Engine.

6.7.1 Accuracy of What-if Analysis

The experimental results in this section resulted from running the MapReduce programs listed in Table 6.4 on a Hadoop cluster running on 16 Amazon EC2 nodes of the c1.medium type. Each node runs at most 2 map tasks and 2 reduce tasks concurrently. Thus, the cluster can run at most 30 map tasks in a concurrent map wave, and at most 30 reduce tasks in a concurrent reduce wave.

First, we present the results from running the Word-Cooccurrence program over 10GB of real data obtained from Wikipedia. Figure 6.3 compares the actual task and phase timings with the corresponding predictions from the What-if Engine. Even though the predicted timings are slightly different from the actual ones, the relative percentage of time spent in each phase is captured fairly accurately. To evaluate the accuracy of the What-if Engine in predicting the overall job execution time, we ran Word Co-occurrence under 40 different configuration settings. We then asked the What-if Engine to predict the job execution time for each setting. Figure 6.4(a) shows a scatter plot of the actual and predicted times for these 40 jobs. Observe the proportional correspondence between the actual and predicted times, and the clear identification of settings with the top-k best and worst performance (indicated by

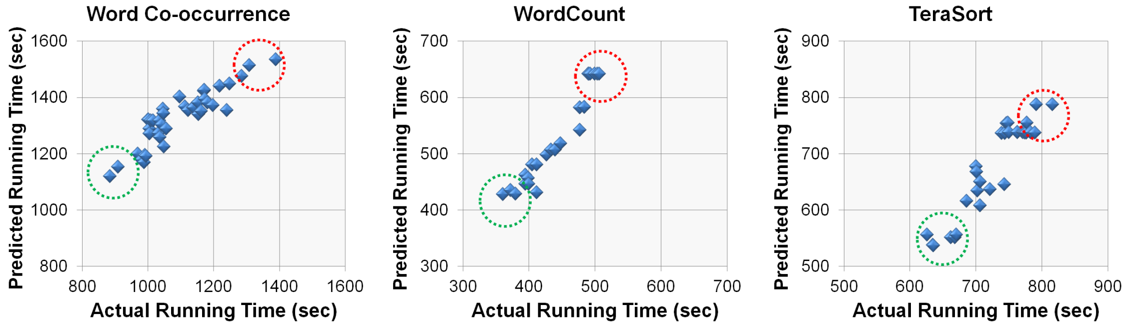


FIGURE 6.4: Actual Vs. predicted running times for (a) Word Co-occurrence, (b) WordCount, and (c) TeraSort jobs running with different configuration parameter settings.

the green and red dotted circles respectively).

We repeated the above experiment with WordCount and TeraSort programs processing 30GB of data each. Figures 6.4(b) and 6.4(c) show two scatter plots of the actual and predicted running times for several WordCount and TeraSort jobs when run using different configuration settings. Once again, we observe that the What-if Engine can clearly identify the settings that will lead to good and bad performance.

As discussed in Section 4.2, the fairly uniform gap between the actual and predicted timings is due to the profiling overhead of BTrace. Since dynamic instrumentation mainly needs additional CPU cycles, the gap is largest when the MapReduce program runs under CPU contention—which was the case when the Word Co-occurrence job was profiled. Unlike the case of Word Co-occurrence in Figure 6.4(a), the predicted values in Figures 6.4(b) and 6.4(c) are closer to the actual values; indicating that the profiling overhead is reflected less in the costs captured in the job profile. As mentioned earlier, we expect to close this gap using commercial Java profilers that have demonstrated vastly lower overheads than BTrace (Louth, 2009).

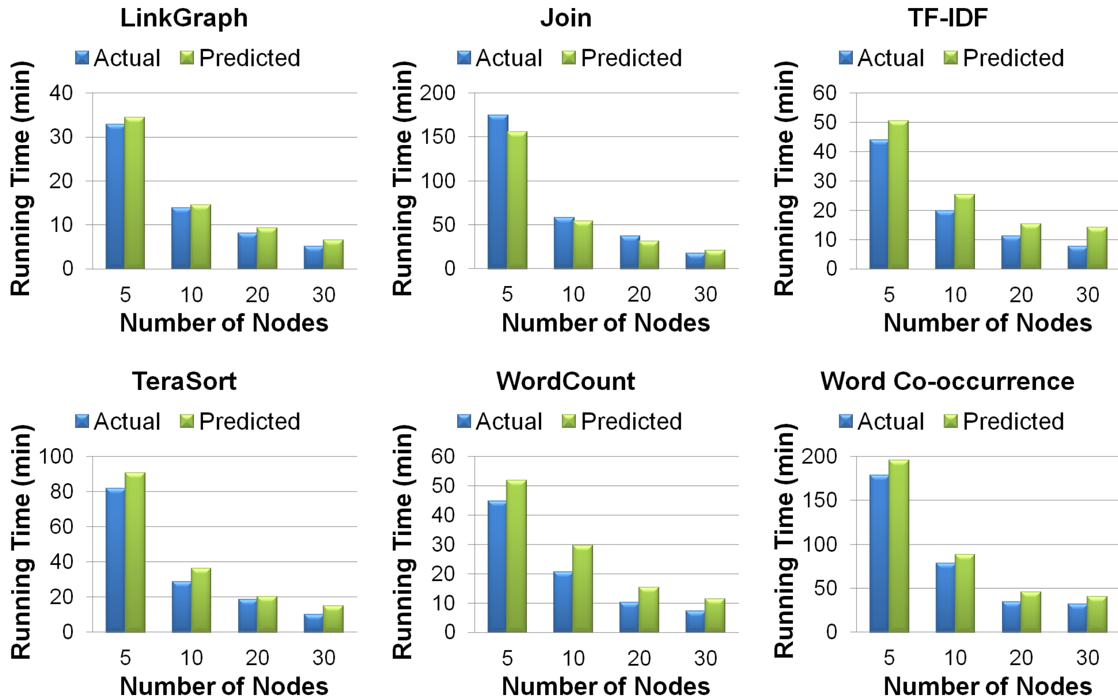


FIGURE 6.5: Actual and predicted running times for MapReduce jobs as the number of nodes in the cluster is varied.

6.7.2 Tuning the Cluster Size

The class of what-if questions we consider in this section is how will the performance of a MapReduce job change if the number of nodes in the existing cluster changes? We evaluate the ability of the What-if Engine to answer such a question automatically.

Figure 6.5 shows the actual and predicted running times for all MapReduce jobs as the number of nodes in the cluster is varied. All Hadoop clusters for this experiment used m1.large EC2 nodes. To make the predictions, we used job profiles that were obtained on a 10-node Hadoop cluster of m1.large EC2 nodes. We observe that the What-if Engine is able to capture the execution trends of all jobs across the clusters with different sizes. That is, the What-if Engine predicts correctly the sublinear speedup achieved for each job as we increase the number of nodes in the cluster.

From the perspective of predicting absolute values, the What-if Engine usually

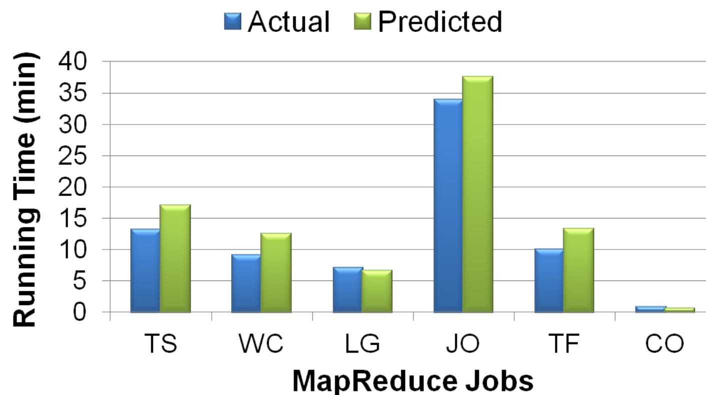


FIGURE 6.6: Actual and predicted running times for MapReduce jobs when run on the production cluster. The predictions used job profiles obtained from the development cluster.

over-predicts job execution time (by 20.1% on average and 58.6% worse case in Figure 6.5). As before, the prediction difference is due to overhead added by BTrace while measuring function timings at nanosecond granularities. While the gap is fairly uniform for different settings of the same MapReduce job, the gap among different jobs varies significantly, making it difficult to correct for it during the prediction process.

6.7.3 Transitioning from Development to Production

Another common use case we consider in our evaluation is the presence of a development cluster, and the need to stage jobs from the development cluster to the production cluster. In our evaluation, we used a 10-node Hadoop cluster with m1.large EC2 nodes as the development cluster, and a 30-node Hadoop cluster with m1.xlarge EC2 nodes as the production one. We profiled all MapReduce programs listed in Table 6.4 on the development cluster. We then executed the MapReduce programs on the production cluster using three times as much data as used in the development cluster (i.e., three times as much data as listed in Table 6.4).

Figure 6.6 shows the actual and predicted running times for each job when run

on the production cluster. The What-if Engine used job profiles obtained from the development cluster for making the predictions. Apart from the overall running time, the What-if Engine can also predict several other aspects of the job execution like the amount of I/O and network traffic, the running time and scheduling of individual tasks, as well as data and computational skew.

Overall, the What-if Engine is capable of accurately capturing the performance trends when varying the configuration parameter settings or the cluster resources.

6.7.4 Evaluating the Training Benchmarks

The ability of the What-if Engine to make accurate predictions across clusters relies on the relative models employed to predict cost statistics. The models we used, like all black-box models, require representative training data in order to make accurate predictions. As discussed in Section 6.3, we have developed three training benchmarks that employ different strategies to collect training samples.

Apriori benchmark: This benchmark includes all jobs listed in Table 6.4, which also form our testing workload. Each job runs over a 600MB random sample of the original input data.

Fixed benchmark: This benchmarks executes the MapReduce jobs WordCount and TeraSort multiple times using different configuration settings. We varied the settings for using intermediate data compression, output compression, and the combine function, since these settings provide tradeoffs between CPU and I/O usage. Each job processed 600MB of randomly generated text using Hadoop's RandomTextGenerator and TeraGen.

Custom benchmark: This benchmark consists of a data generation job template and a data processing job template, as discussed in Section 6.3. The data generation job template is run twice (with output compression turned on and off) and the data processing job template is run four times (corresponding to the four possible

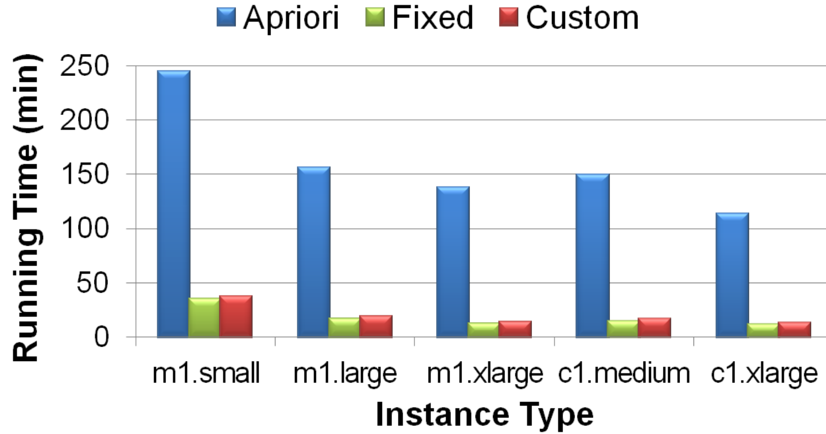


FIGURE 6.7: Total running time for each training benchmark.

combinations of using compression and the combine function).

All benchmarks were run on 10-node Hadoop clusters on EC2 nodes. Each job in each benchmark processed 600MB of data and was run using rules-of-thumb settings. We tested the prediction accuracy of the relative models trained by each benchmark on a test workload consisting of all jobs listed in Table 6.4.

Figure 6.7 shows the running time of each benchmark for collecting all the training data. The Apriori benchmark takes a significantly longer time to complete compared to the other two benchmarks as it executes more MapReduce jobs. Unlike the running time for the Fixed and Custom benchmark, the running time for the Apriori benchmark is unpredictable and directly depends on representative jobs provided by the user. The Custom benchmark, on the other hand, completes fast due to its focused nature of going after a spectrum of cost statistics within the same job.

In order to compare the prediction accuracy of the relative models when trained with the three benchmarks, we created a test workload consisting of all MapReduce jobs from Table 6.4. The test workload was executed on five 10-node Hadoop clusters—one for each node type we considered in our evaluation (see Table 6.3). We then used the job profiles obtained on the m1.large cluster to predict the job

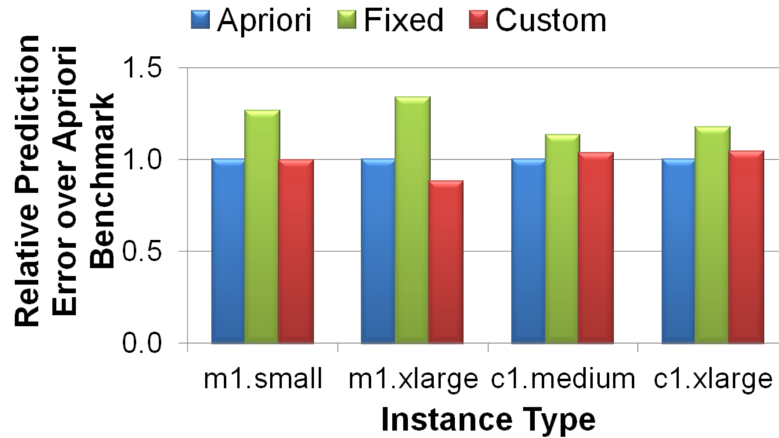


FIGURE 6.8: Relative prediction error for the Fixed and Custom benchmarks over the Apriori benchmark when asked to predict cost statistics for a test workload.

profiles for the other four clusters (i.e., relative predictions). As the Apriori benchmark assumes full knowledge of the test workload, we will use it as the baseline when comparing the prediction accuracy of the three benchmarks.

Figure 6.8 shows the relative prediction error from using the Fixed and Custom benchmarks against using the Apriori benchmark. Even though the processing performed by the jobs in the Custom benchmark is completely independent from and unrelated to the test workload, the prediction errors we observed are relatively low, typically less than 15%. The Fixed benchmark results in the highest prediction errors: running a predefined set of jobs with various settings does not seem to provide adequate coverage of the possible cost statistics encountered during the execution of the test workload.

Even though the Apriori benchmark leads to good predictions when the test workload contains the same or similar jobs with the training workload, it can lead to poor predictions for new jobs. For evaluation purposes, we excluded the TF-IDF job from the training workload of the Apriori benchmark. We then tested the relative models with the TF-IDF job profiles. We observed higher prediction errors compared to predictions for the other jobs: Figure 6.9 shows how the Apriori benchmark is now

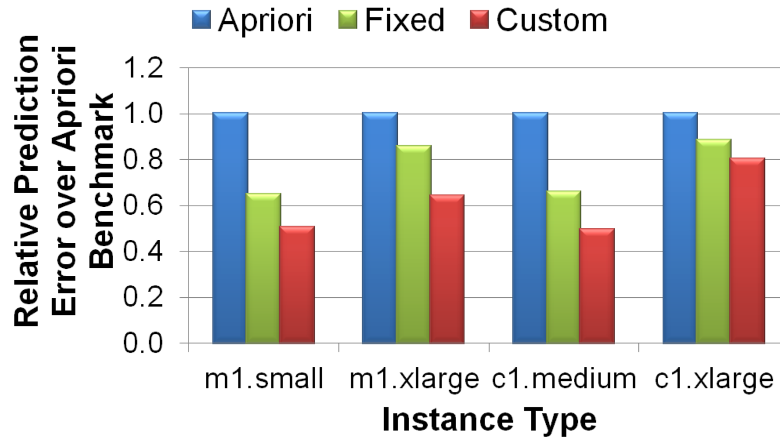


FIGURE 6.9: Relative prediction error for the Fixed and Custom benchmarks over the Apriori benchmark without TF-IDF when asked to predict cost statistics for the TF-IDF job.

outperformed by the Custom benchmark.

Overall, when the workload is known a priori and the high time to collect the training data is not a concern, using the Apriori benchmark is the best option. Otherwise, the Custom benchmark is a reliable and robust option for collecting training samples quickly that lead to good predictions.

7

Cost-based Optimization for MapReduce Workloads

Suppose we want to execute a MapReduce workflow W on a given input dataset and a given MapReduce cluster. We can think of the settings for the configuration parameters for all jobs in W as specifying an execution plan for W. Different choices of the execution plan give rise to potentially different values for the performance metric of interest for the workflow. In Chapter 6, we have seen how the What-if Engine estimates the cost of one execution plan. The main problem we address in this chapter is to automatically and efficiently choose a good execution plan for a MapReduce workflow given the input dataset and the cluster characteristics. For ease of exposition, we focus on optimizing individual MapReduce jobs first, followed by optimization strategies for MapReduce workflows as well as cluster resources.

The optimization problem we are facing in MapReduce systems is similar in nature to the cost-based query optimization problem in Database systems. Query optimizers are responsible for finding a good execution plan p for a given query q, given an input set of tables with some data properties d, and some resources r allocated to run the plan. Database query optimizers use statistics about the data

(e.g., histograms) and known properties of logical operators (e.g., selection, join) to estimate the size of data processed as input and produced as output by each physical operator (e.g., index scan, hash join) in an execution plan. These dataflow estimates and a *cost model* are used to estimate the performance cost y of a plan p. The query optimizer employs a *search strategy* (e.g., dynamic programming (Selinger et al., 1979)) to explore the space of all possible execution plans in search for the plan with the least estimated cost y.

There is more than thirty years of work on database query optimization technology that can be leveraged. However, due to the MADDER principles, there exist some major challenges that we need to overcome in order to employ a cost-based optimization approach to MapReduce programs.

- **Black-box map and reduce functions:** Map and reduce functions are usually written in general-purpose programming languages, like Java and Python, that are not restrictive or declarative like SQL. Thus, the approach of modeling a small and finite space of relational operators does not work for MapReduce programs.

- **Lack of schema and statistics about the input data:** Almost no information about the schema and statistics of input data may be available before the MapReduce job is submitted. Furthermore, keys and values are often extracted dynamically from the input data by the map function, so it may not be possible to collect and store statistics about the data beforehand.

- **Differences in plan spaces:** The execution plan space for SQL queries is very different from the plan space of job configuration parameters for MapReduce programs; so algorithms from SQL query optimizers do not translate directly for optimizing MapReduce programs.

- **Interactions among various plan choices:** The jobs in a workflow exhibit *dataflow dependencies* because of producer-consumer relationships as well as *cluster resource dependencies* because of concurrent scheduling. Optimizing configuration settings for MapReduce workflows is challenging because it has to account for these dependencies and the consequent interactions among configuration settings of different jobs in a workflow.

- **Intertwining optimization and provisioning decisions:** In addition to selecting job-level and workflow-level configuration settings, the user is faced regularly with complex *cluster sizing problems* that involve finding the cluster size and the type of resources to use in the cluster from the large number of choices offered by current cloud platforms (recall Section 2.2.1).

The Profiler, What-if Engine, and the Cost-based Optimizers—the three main components of our solution shown in Figure 2.2—are designed to address these challenges in MapReduce workflow optimization.

This chapter is organized as follows. Section 7.1 discusses related work for MapReduce job optimization, MapReduce workflow optimization, and cluster provisioning. Section 7.2 presents the first cost-based *Job Optimizer* for finding good configuration settings automatically for simple to arbitrarily complex MapReduce programs. We formulate the workflow optimization problem, and describe a family of automatic *Workflow Optimizers* that were developed to address this problem in Section 7.3. Finally, the *Cluster Resource Optimizer*, presented in Section 7.4, is responsible for enumerating and searching the space of possible cluster resources. To the best of our knowledge, these contributions are being made for the first time in the literature.

7.1 Current Approaches to MapReduce Optimization

MapReduce is emerging rapidly as a viable competitor to existing systems for Big Data analytics. While MapReduce currently trails existing systems in peak query performance (Pavlo et al., 2009), a number of ongoing research projects are addressing this issue through optimization opportunities arising at different levels of the MapReduce stack (Abouzeid et al., 2009; Bu et al., 2010; Dittrich et al., 2010; Jiang et al., 2010). Starfish fills a different void by enabling MapReduce users and applications to get good performance automatically without any need on their part to understand and manipulate the many optimization knobs available.

Optimizations for MapReduce jobs: Today, when users are asked to find good configuration settings for MapReduce jobs, they have to rely on their experience, intuition, knowledge of the data being processed, *rules of thumb* from human experts or tuning manuals, or even guesses to complete the task. Following popular rules of thumb has become a standard technique for setting job configuration parameter settings (Hadoop Tutorial, 2011; Lipcon, 2009; White, 2010). For example, *mapred.reduce.tasks* (the Hadoop parameter that specifies the number of reduce tasks in a job) is set to roughly 0.9 times the total number of reduce execution slots in the cluster. The rationale is to ensure that all reduce tasks run in one wave while leaving some slots free for re-executing failed or slow tasks. It is important to note that many rules of thumb still require information from past job executions to work effectively. For example, setting *io.sort.record.percent* requires calculating the average map output record size based on the number of records and size of the map output produced during a job execution. This rule of thumb sets *io.sort.record.percent* to $\frac{16}{16+avg_record_size}$. (The rationale here involves source-code details of Hadoop.)

The response surfaces in Figures 3.5 and 3.6 in Section 3.2 show that the rule-of-thumb settings gave poor performance in both cases. In fact, the rule-of-thumb

settings for WordCount gave one of its worst execution times: *io.sort.record.percent* and *io.sort.mb* were set too high. In the case of TeraSort, the rule of thumb for *io.sort.record.percent* was able to achieve a local minimum, when *mapred.reduce.tasks* was set to 27 (based on the rule of thumb: $0.9 \times 30\ slots = 27$). However, a much larger number of reduce tasks was necessary in order to achieve a better performance for TeraSort.

When information from previous job executions is available, postmortem performance analysis and diagnostics can also help with identifying performance bottlenecks in MapReduce jobs. *Hadoop Vaidya* (Hadoop Vaidya, 2011) and *Hadoop Performance Monitoring UI* (Hadoop Perf UI, 2011) execute a small set of predefined diagnostic rules against the job execution counters to diagnose various performance problems, and offer targeted advice. Unlike our optimizers, the recommendations given by these tools are qualitative instead of quantitative. For example, if the ratio of spilled records to total map output records exceeds a user-defined threshold, then Vaidya will suggest increasing *io.sort.mb*, but without specifying by how much to increase. On the other hand, our cost-based approach automatically suggests concrete configuration settings to use.

A MapReduce program has semantics similar to a *Select-Project-Aggregate* (SPA) in SQL with User-defined functions (UDFs) for the selection and projection (map) as well as the aggregation (reduce). This equivalence is used in recent work to perform semantic optimization of MapReduce programs (Blanas et al., 2010; Cafarella and Ré, 2010; Nykiel et al., 2010; Olston et al., 2008a). *HadoopToSQL* and *Manimal* perform static analysis of MapReduce programs written in Java in order to extract declarative constructs like filters and projections. These constructs are then used for database-style optimizations such as the use of B-Tree indexes, avoiding reads of unneeded data, and column-aware compression (Cafarella and Ré, 2010; Iu and Zwaenepoel, 2010). Manimal does not perform profiling, what-if analysis, or

cost-based optimization; it uses rule-based optimization instead. *MRShare* performs multi-query optimization by running multiple SPA programs in a single MapReduce job (Nykiel et al., 2010). MRShare proposes a (simplified) cost model for this application. SQL joins over MapReduce have been proposed in the literature (Afrati and Ullman, 2009; Blanas et al., 2010), but cost-based optimization is either missing or lacks comprehensive profiling and what-if analysis.

Apart from the typical business application domains, MapReduce is useful in the scientific analytics domain. The *SkewReduce* system (Kwon et al., 2010) focuses on applying some specific optimizations to MapReduce programs from this domain. SkewReduce includes an optimizer to determine how best to partition the map-output data to the reduce tasks. Unlike our cost-based optimizers, SkewReduce relies on user-specified cost functions to estimate job execution times for the various different ways to partition the data.

In summary, previous work related to MapReduce job optimization targets semantic optimizations for MapReduce programs that correspond predominantly to SQL specifications (and were evaluated on such programs). In contrast, Starfish's What-if Engine support simple to arbitrarily complex MapReduce programs expressed in whatever programming language the user or application finds convenient. We focus on the optimization opportunities presented by the large space of MapReduce job configuration parameters.

Optimizations for higher-level MapReduce workloads: For higher levels of the MapReduce stack that have access to declarative semantics, many optimization opportunities inspired by database query optimization and workload tuning have been proposed. Hive and Pig employ rule-based approaches for a variety of optimizations such as filter and projection pushdown, shared scans of input datasets across multiple operators from the same or different analysis tasks (Nykiel et al.,

2010), reducing the number of MapReduce jobs in a workflow (Lee et al., 2011), and handling data skew in sorts and joins. The *epiC* system supports System-R-style join ordering (Wu et al., 2011). Improved data layouts inspired by database storage have also been proposed (e.g., Jindal et al. (2011)). All of this work is complementary to our work on optimizing configuration parameters for MapReduce workflows in an interaction-aware manner.

Automatic workflow optimization has a long history in grid computing. Initially, grid systems focused on scheduling tasks on unused CPU capacity (Thain et al., 2005). Data placement and movement occurred as a side effect of task execution. As dataset sizes grew, researchers began to incorporate *data-driven scheduling* approaches that consider storage demands and data transfer costs (Bent et al., 2009; Shankar and Dewitt, 2007). Awareness of data transfer costs and other scheduling-level optimizations are built into task schedulers proposed for newer systems like Hadoop, *Dryad*, and *SCOPE* (Isard et al., 2009; Zaharia et al., 2010). Our work differs from the above work in a number of ways. First and foremost, the above work is about scheduling a *predetermined* set of tasks *per job*. Our techniques complement the scheduling-level optimizations by automatically picking the best degree of task-level parallelism to process the jobs in a workflow. Furthermore, these decisions are made to optimize the overall workflow performance, and not at the level of individual jobs.

Dryad and SCOPE are data-parallel computing engines whose designs lie between databases and MapReduce (Isard et al., 2007; Zhou et al., 2010). *DryadLINQ* is a language layer that integrates distributed queries into high-level .NET programming languages. These systems contain certain optimizations that are not present in MapReduce implementations, e.g., the ability to perform a large class of distributed aggregations by composing a tree of partial aggregations (Yu et al., 2009). While we work with MapReduce, the optimization techniques we propose are applicable

broadly in data-parallel job workflows. DryadLINQ supports a dynamically-changing execution graph based on run-time information, which is similar to a dynamic optimization approach that we propose in this chapter.

Applying optimizations at the higher or lower levels of the MapReduce stack have their respective advantages and disadvantages. Intuitively, optimizations at a higher level are expected to give bigger gains than optimizations at a lower level. Nevertheless, optimizations done at lower levels of the MapReduce stack apply irrespective of the higher-level interface used (e.g., Hive or Pig) to specify workflows. In addition, the heavy use of UDFs makes it impossible to apply many optimizations at the higher level. Finally, we have observed from operational use of MapReduce clusters that configuration parameters are controlled by administrators. Thus, similar to physical design of Database systems, administrators can tune configuration parameters in order to tune important or poorly-performing jobs as needed.

Cluster provisioning and modeling: Our work shares some goals with a recent work on provisioning Hadoop on cloud platforms (Kambatla et al., 2009). The proposed approach uses the following steps: (i) for a training workload of MapReduce jobs, perform brute-force search over the resource configuration space to find the best configuration; (ii) use the collected data to build a signature database that maps resource utilization signatures from the jobs to the optimal configuration; and (iii) given a new job j, run a scaled-down version of j to get j's resource utilization signature, and probe the signature database to find the best match. Only two configuration parameters were considered, and no solution was proposed for finding the number of nodes in the cluster. Furthermore, a brute-force approach will not scale to the complex configuration space arising in MapReduce systems.

There has been considerable interest recently in using black-box models like regression trees to build workload performance predictors in large-scale Data Centers

(Bodik et al., 2009). These models can be trained automatically from samples of system behavior, and retrained when major changes happen. However, these models are only as good as the predictive behavior of the independent variables they use and how well the training samples cover the prediction space. As the number of independent variables that affect workload performance increases (e.g., data properties, configuration parameter settings, and scheduling policies), the number of training samples needed to learn effective black-box models increases dramatically.

There have been proposals to eliminate modeling altogether, relying instead on actual performance measurements through planned *experiments* (Duan et al., 2009; Zheng et al., 2009). While this approach can give accurate predictions for some specific problems, representative experiments are nontrivial to set up and take time to run. Given the growing number of commercial cloud platforms, recent research has looked into benchmarking them (Li et al., 2010). Such benchmarks complement our work on building relative black-box models that can predict the performance of a workload W on one provider A based on the performance of W measured on another provider B.

7.2 Cost-based Optimization of MapReduce Jobs

MapReduce job optimization is defined as:

> Given a MapReduce program p to be run on input data d and cluster resources r, find the setting of configuration parameters $c_{opt} = \underset{c \in S}{\operatorname{argmin}} F(p, d, r, c)$ for the cost model F represented by the What-if Engine over the full space S of configuration parameter settings.

The *Cost-based Optimizer (CBO)* addresses this problem by making what-if calls with settings c of the configuration parameters selected through an enumeration and search over S. Recall that the cost model F represented by the What-if Engine is

implemented as a mix of simulation and model-based estimation. As observed from Figures 3.5 and 3.6 in Section 3.2, F is high-dimensional, nonlinear, nonconvex, and multimodal. For providing both efficiency and effectiveness, the CBO must minimize the number of what-if calls while finding near-optimal configuration settings.

The What-if Engine needs as input a job profile for the MapReduce program p. In the common case, this profile is already available when p has to be optimized. The program p may have been profiled previously on input data d_0 and cluster resources r_0 which have the same properties as the current d and r respectively. Profiles generated previously can also be used when the dataflow proportionality assumption can be made. Such scenarios are common in companies like Facebook, LinkedIn, and Yahoo! where a number of MapReduce programs are run periodically on log data collected over a recent window of time (Gates, 2010; Sood, 2010).

Recall from Section 6.2 that the job profile input to the What-if Engine can also come fully or in part from an external module like Hive or Pig that submits the job. This feature is useful when the dataflow proportionality assumption is expected to be violated significantly, e.g., when a repeated job runs on input data with highly dissimilar statistical properties. In addition, we have implemented two methods for the CBO to use for generating a new profile when one is not available to input to the What-if Engine:

1. The CBO can decide to forgo cost-based optimization for the current job execution. However, the current job execution will be profiled to generate a job profile for future use.

2. The Profiler can be used in a *just-in-time* mode to generate a job profile using sampling as described in Section 4.4.

Once a job profile is available, the CBO will use it along with the input data properties and the cluster resources to find the best configuration settings for the new

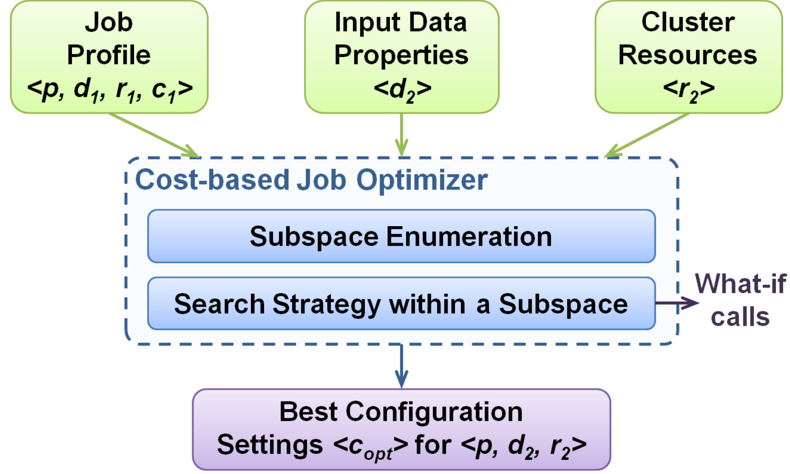


FIGURE 7.1: Overall process for optimizing a MapReduce job.

(hypothetical) job. Figure 7.1 shows the overall process for optimizing a MapReduce job. The CBO uses a two-step process: (i) subspace enumeration, and (ii) search within each enumerated subspace. The two steps are discussed next.

7.2.1 Subspace Enumeration

A straightforward approach the CBO can take is to apply enumeration and search techniques to the full space of parameter settings S. (Note that the parameters in S are those whose performance effects are modeled by the What-if Engine, and are listed in Table 3.1.) However, the high dimensionality of S affects the scalability of this approach. More efficient search techniques can be developed if the individual parameters in c can be grouped into clusters, denoted $c^{(i)}$, such that the globally-optimal setting c_{opt} in S can be composed from the optimal settings $c^{(i)}_{opt}$ for the clusters. That is:

$$c_{opt} = \bigodot_{i=1}^{l} \underset{c^{(i)} \in S^{(i)}}{\operatorname{argmin}} F(p, d, r, c^{(i)}), \text{ with } c = c^{(1)} \cdot c^{(2)} \cdots c^{(l)} \tag{7.1}$$

Here, $S^{(i)}$ denotes the subspace of S consisting of only the parameters in $c^{(i)}$. $\odot$ denotes a composition operation.

Equation 7.1 states that the globally-optimal setting c_{opt} can be found using a divide and conquer approach by (i) breaking the higher-dimensional space S into the lower-dimensional subspaces $S^{(i)}$, (ii) considering an independent optimization problem in each smaller subspace, and (iii) composing the optimal parameter settings found per subspace to give the setting c_{opt}.

MapReduce gives a natural clustering of parameters into two clusters: parameters that predominantly affect map task execution, and parameters that predominantly affect reduce task execution. For example, Hadoop's *io.sort.mb* parameter only affects the Spill phase in map tasks, while *mapred.job.shuffle.merge.percent* only affects the Shuffle phase in reduce tasks. The two subspaces for map tasks and reduce tasks respectively can be optimized independently. As we will show in Section 7.2.3, the lower dimensionality of the subspaces decreases the overall optimization time drastically.

Some parameters have small and finite domains, e.g., Boolean. At the other extreme, the CBO has to narrow down the domain of any parameter whose domain is unbounded. In these cases, the CBO relies on information from the job profile and the cluster resources. For example, the CBO uses the maximum heap memory available for map task execution, along with the program's memory requirements (predicted based on the job profile), to bound the range of *io.sort.mb* values that can contain the optimal setting.

7.2.2 Search Strategy within a Subspace

The second step of the CBO involves searching within each enumerated subspace to find the optimal configuration in the subspace. There is an extensive body of work on finding good settings in complex response surfaces using techniques like simulated

annealing (Romeijn and Smith, 1994) and genetic algorithms (Goldberg, 1989). We have adapted the *Recursive Random Search* technique, a fairly recent technique developed to solve black-box global optimization problems (Ye and Kalyanaraman, 2003). For comparison purposes, we also implemented Optimizers that attend to cover the parameter space S using a *Gridding* approach.

Gridding (Equispaced or Random): Gridding is a simple technique to generate points in a space with n parameters. The domain $dom(c_i)$ of each configuration parameter c_i is discretized into k values. The values may be equispaced or chosen randomly from $dom(c_i)$. Thus, the space of possible settings, $DOM \subseteq \prod_{i=0}^{n} dom(c_i)$, is discretized into a grid of size k^n. The CBO makes a call to the What-if Engine for each of these k^n settings, and selects the setting with the lowest estimated job execution time.

Recursive Random Search (RRS): RRS first samples the subspace randomly to identify promising regions that contain the optimal setting with high probability. It then samples recursively in these regions which either move or shrink gradually to locally-optimal settings based on the samples collected. RRS then restarts random sampling to find a more promising region to repeat the recursive search. We adopted RRS for three important reasons: (a) RRS provides probabilistic guarantees on how close the setting it finds is to the optimal setting; (b) RRS is fairly robust to deviations of estimated costs from actual performance; and (c) RRS scales to a large number of dimensions (Ye and Kalyanaraman, 2003).

In summary, we implemented six different cost-based optimizers for finding near optimal configuration parameter settings. There are two choices for subspace enumeration: *Full or Clustered* that deal respectively with the full space S or smaller subspaces for map and reduce tasks; and three choices for search within a subspace: *Gridding Equispaced*, *Gridding Random*, and *RRS*.

Table 7.1: MapReduce programs and corresponding datasets for the evaluation of the Job Optimizer.

Abbr.	MapReduce Program	Dataset Description
CO	Word Co-occurrence	10GB of documents from Wikipedia
WC	WordCount	30GB of documents from Wikipedia
TS	Hadoop's TeraSort	30GB data from Hadoop's TeraGen
LG	LinkGraph	10GB compressed data from Wikipedia
JO	Join	30GB data from the TPC-H Benchmark

7.2.3 Evaluating Cost-based Job Optimization

The experimental setup used is a Hadoop cluster running with 1 master and 15 slave Amazon EC2 nodes of the c1.medium type. Each slave node runs at most 2 map tasks and 2 reduce tasks concurrently. Thus, the cluster can run at most 30 map tasks in a concurrent map wave, and at most 30 reduce tasks in a concurrent reduce wave. Table 7.1 lists the MapReduce programs and datasets used in our evaluation. We selected representative MapReduce programs used in different domains: text analytics (WordCount), natural language processing (Word Co-occurrence), creation of large hyperlink graphs (LinkGraph), and business data processing (Join, TeraSort) (Lin and Dyer, 2010; White, 2010).

We compare the Cost-based Optimizers (CBOs) against the Rules-of-Thumb (RoT) approach that suggests configuration settings based on the rules of thumb used by Hadoop experts to tune MapReduce jobs (described in Section 7.1). Applying the rules of thumb requires information from past job execution as input. CBOs need job profiles as input which were generated by the Profiler by running each program using the rules-of-thumb settings. Our default CBO is Clustered RRS. Our evaluation methodology is:

1. We evaluate our cost-based approach against Rules-of-Thumb to both validate the need for a CBO and to provide insights into the nontrivial nature of cost-

Table 7.2: MapReduce job configuration settings in Hadoop suggested by Rules-of-Thumb (RoT) and the Cost-based Optimizer (CBO) for the Word Co-occurrence program.

Conf. Parameter (described in Table 3.1)	RoT Settings	CBO Settings
io.sort.factor	10	97
io.sort.mb	200	155
io.sort.record.percent	0.08	0.06
io.sort.spill.percent	0.80	0.41
mapred.compress.map.output	TRUE	FALSE
mapred.inmem.merge.threshold	1000	528
mapred.job.reduce.input.buffer.percent	0.00	0.37
mapred.job.shuffle.input.buffer.percent	0.70	0.48
mapred.job.shuffle.merge.percent	0.66	0.68
mapred.output.compress	FALSE	FALSE
mapred.reduce.tasks	27	60
min.num.spills.for.combine	3	3
Use of the combine function	TRUE	FALSE

based optimization of MapReduce programs.

2. We compare the six different CBOs proposed in terms of effectiveness and efficiency.

3. We evaluate our cost-based approach against Rules-of-Thumb in more trying scenarios where predictions have to be given for a program p running on a large dataset d_2 on the production cluster r_2 based on a profile learned for p from a smaller dataset d_1 on a small development cluster r_1.

4. We evaluate the accuracy versus efficiency tradeoff from the approximate profile generation techniques in the Profiler.

1. Rule-based Vs. cost-based optimization: We ran the Word Co-occurrence MapReduce program using the configuration parameter settings shown in Table 7.2 as suggested by the Rules-of-Thumb (RoT) and the (default) CBO. Jobs J_{RoT} and

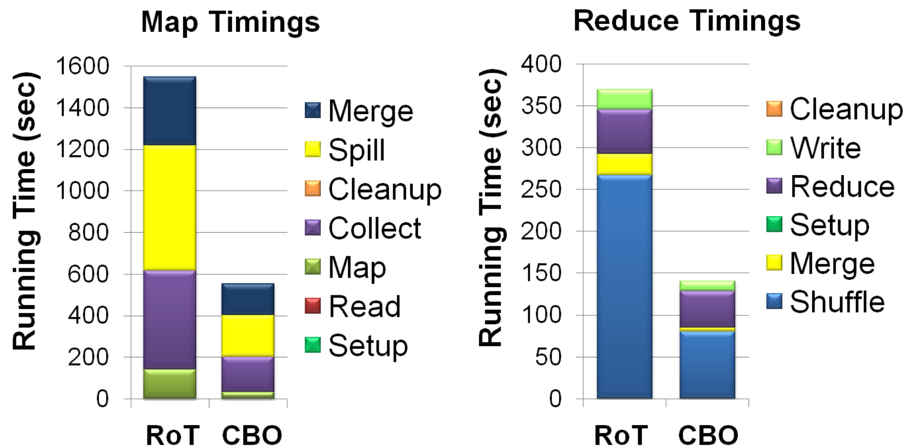


FIGURE 7.2: Map and reduce time breakdown for two Word Co-occurrence jobs run with configuration settings suggested by Rules-of-Thumb (RoT) and the Cost-based Optimizer (CBO).

J_{CBO} denote respectively the execution of Word Co-occurrence using the Rules-of-Thumb and CBO settings. Note that the same Word Co-occurrence program is processing the same input dataset in either case. While J_{RoT} runs in 1286 seconds, J_{CBO} runs in 636 seconds (around 2x faster).

Figure 7.2 shows the task time breakdown from the job profiles collected by running Word Co-occurrence with the Rules-of-Thumb and CBO-suggested configuration settings. Our first observation from Figure 7.2 is that the map tasks in job J_{CBO} completed on average much faster compared to the map tasks in J_{RoT}. The higher settings for *io.sort.mb* and *io.sort.spill.percent* in J_{RoT} (see Table 7.2) resulted in a small number of large spills. The data from each spill was processed by the Combiner and the Compressor in J_{RoT}, leading to high data reduction. However, the Combiner and the Compressor together caused high CPU contention, negatively affecting all the compute operations in J_{RoT}'s map tasks (executing the user-provided map function, serializing, and sorting the map output).

The Optimizer decided to lower the settings for *io.sort.mb* and *io.sort.spill.percent* compared to the settings in J_{RoT}, leading to more, but individually smaller, map-

side spills. Since sorting occurs on the individual spills, smaller spills improve the overall sorting time. However, more spills may require additional merge rounds to produce one single map output file that will be transfered to the reducers. Since the Optimizer is aware of the tradeoffs between sorting and merging, it chose to increase the setting for *io.sort.factor* to ensure that all the spills will be merged in a single merge round, avoiding unnecessary I/O caused from intermediate merge rounds.

In addition, CBO chose to disable both the use of the Combiner and compression (see Table 7.2) in order to alleviate the CPU-contention problem. Consequently, the CBO settings caused an increase in the amount of intermediate data spilled to disk and shuffled to the reducers. CBO also chose to increase the number of reduce tasks in J_{CBO} to 60 due to the increase in shuffled data, causing the reducers to execute in two waves. However, the additional local I/O and network transfer costs in J_{CBO} were dwarfed by the huge reduction in CPU costs; effectively, giving a more balanced usage of CPU, I/O, and network resources in the map tasks of J_{CBO}. Unlike CBO, Rules-of-Thumb are not able to capture such complex interactions among the configuration parameters and the cluster resources, leading to significantly suboptimal performance.

2. Efficiency and Effectiveness of CBOs: We now evaluate the efficiency and effectiveness of our six CBOs and Rules-of-Thumb in finding good configuration settings for all the MapReduce programs in Table 7.1. Figure 7.3 shows running times for MapReduce programs run using the job configuration parameter settings from the respective optimizers. Rules-of-Thumb settings provide an average 4.6x and maximum 8.7x improvement over Hadoop's Default settings (shown in Table 3.1) across all programs. Settings suggested by our default Clustered RRS CBO provide an average 8.4x and maximum 13.9x improvement over Default settings, and an average 1.9x and maximum 2.2x improvement over Rules-of-Thumb settings.

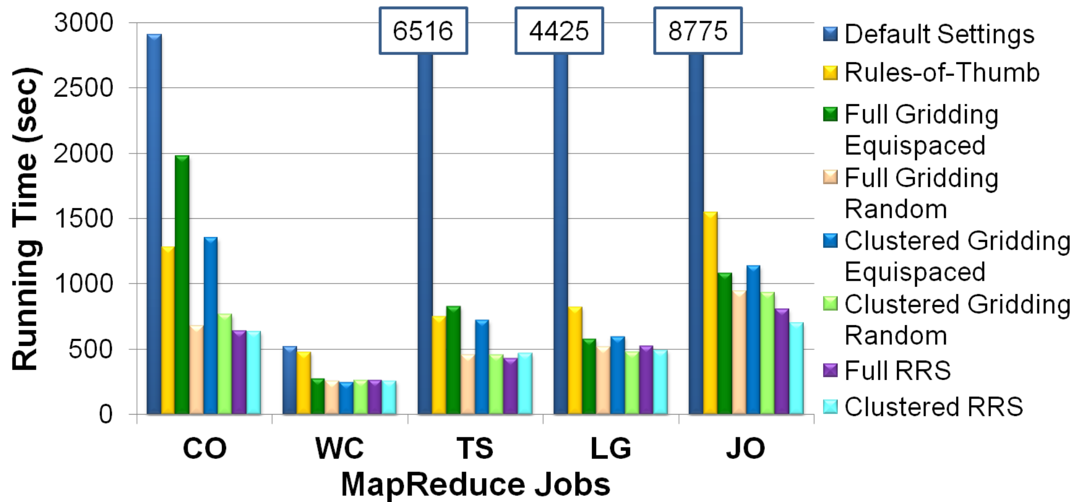


FIGURE 7.3: Running times for MapReduce jobs running with Hadoop's Default, Rules-of-Thumb, and CBO-suggested settings.

Figure 7.3 shows that the RRS Optimizers—and Clustered RRS in particular—consistently lead to the best performance for all the MapReduce programs. All the Gridding Optimizers enumerate up to k=3 values from each parameter's domain. The Gridding Equispaced (Full or Clustered) Optimizers perform poorly sometimes because using the minimum, mean, and maximum values (the three values that correspond to k=3) from each parameter's domain can lead to poor coverage of the configuration space. The Gridding Random Optimizers perform better.

Figures 7.4 and 7.5 respectively show the optimization time and the total number of what-if calls made by each CBO. (Note the log scale on the y-axis.) The Gridding Optimizers make an exponential number of what-if calls, which causes their optimization times to range in the order of a few minutes. For Word Co-occurrence, the Full Gridding Optimizers explore settings for n=14 parameters, and make 314,928 calls to the What-if Engine. Clustering parameters into two lower-dimensional subspaces decreases the number of what-if calls drastically, reducing the overall optimization times down to a few seconds. For Word Co-occurrence, the Clustered Gridding Optimizers made only 6,480 what-if calls.

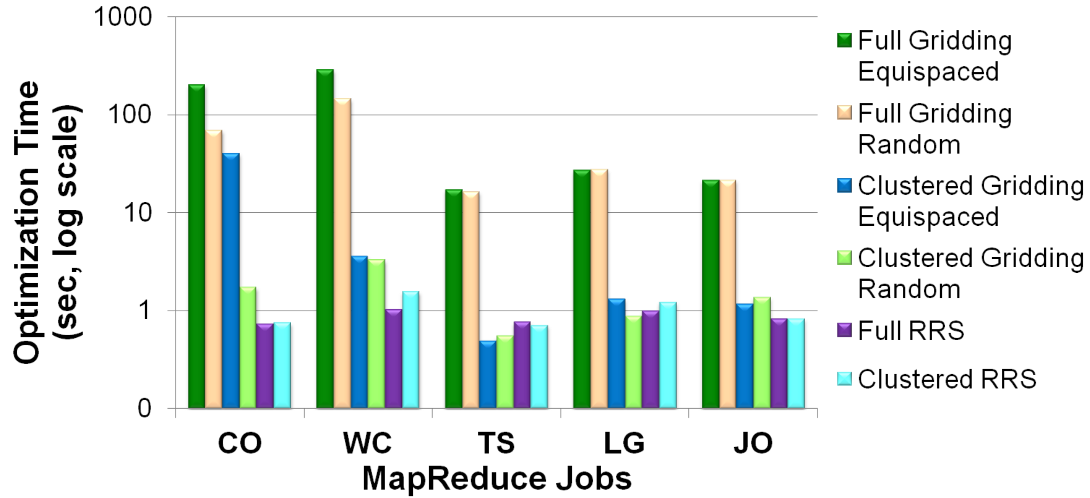


FIGURE 7.4: Optimization time for the six Cost-based Optimizers for various MapReduce jobs.

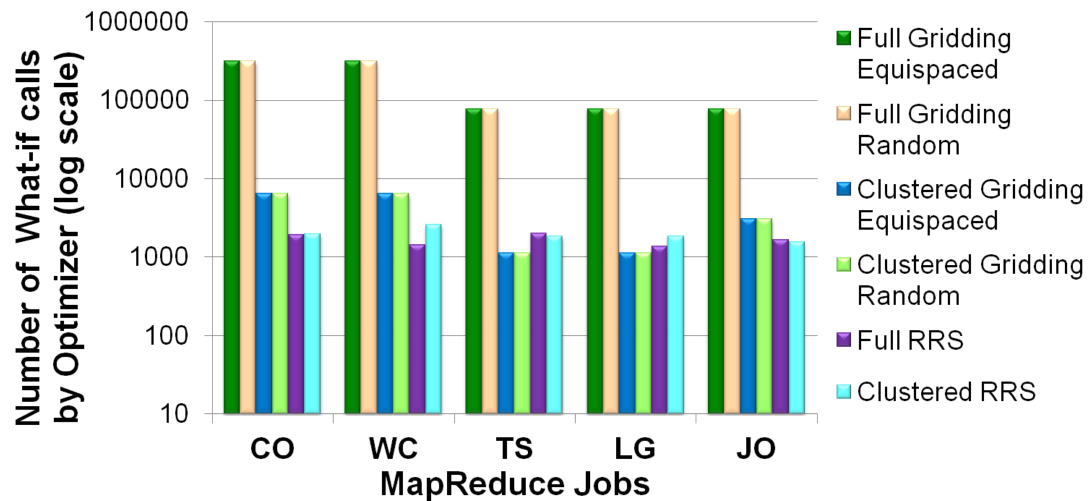


FIGURE 7.5: Number of what-if calls made (unique configuration settings considered) by the six Cost-based Optimizers for various MapReduce jobs.

The RRS Optimizers explore the least number of configuration settings due to the targeted sampling of the search space. Their optimization time is typically less than 2 seconds. Our default Clustered RRS CBO found the best configuration setting for Word Co-occurrence in 0.75 seconds after exploring less than 2,000 settings.

3. Cost-based optimization in other common scenarios: Many organizations

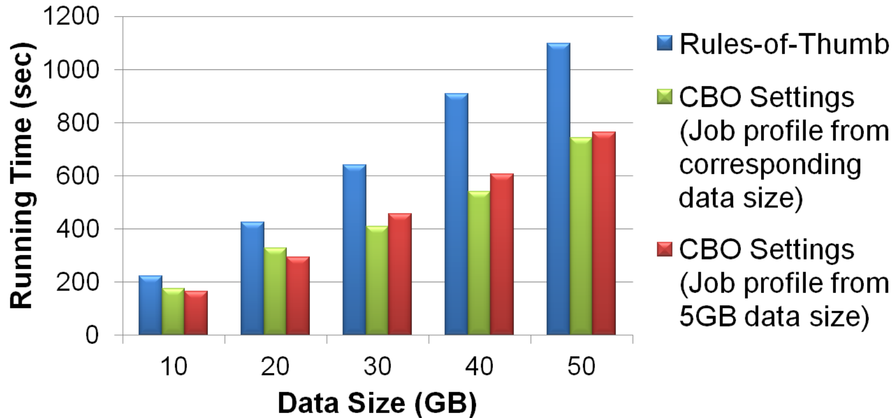


FIGURE 7.6: The job execution times for TeraSort when run with (a) Rules-of-Thumb settings, (b) CBO-suggested settings using a job profile obtained from running the job on the corresponding data size, and (c) CBO-suggested settings using a job profile obtained from running the job on 5GB of data.

run the same MapReduce programs over datasets with similar data distribution but different sizes (Sood, 2010). For example, the same report generation program may be used to generate daily, weekly, and monthly reports. Or, the daily log data collected and processed may be larger for a weekday than the data for the weekend. For the experiments reported here, we profiled the TeraSort MapReduce program executing on a small dataset of size 5GB. Then, we used the generated job profile $prof(J_{5GB})$ as input to the Clustered RRS Optimizer to find good configuration settings for TeraSort jobs running on larger datasets.

Figure 7.6 shows the running times of TeraSort jobs when run with the CBO settings using the job profile $prof(J_{5GB})$. For comparison purposes, we also profiled each TeraSort job when run over the larger actual datasets, and then asked the CBO for the best configuration settings. We observe from Figure 7.6 that, in all cases, the performance improvement achieved over the Rules-of-Thumb settings is almost the same; irrespective of whether the CBO used the job profile from the small dataset or the job profile from the actual dataset. Thus, when the dataflow proportionality assumption holds—as it does for TeraSort—obtaining a job profile from running a

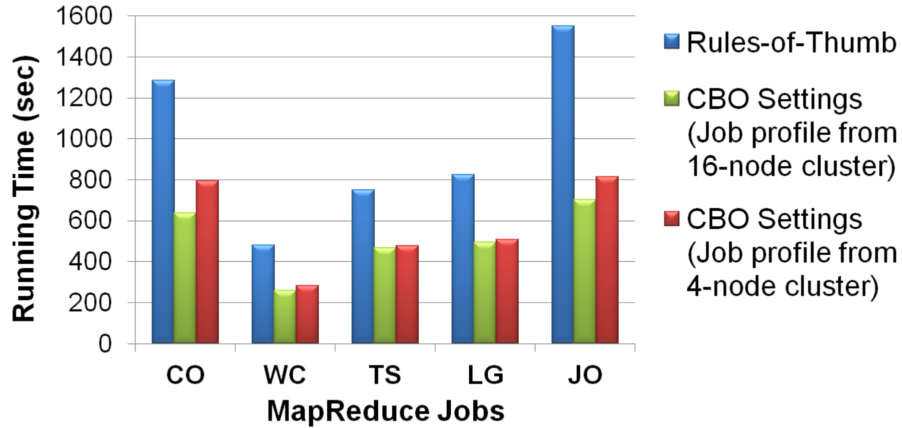


Figure 7.7: The job execution times for MapReduce programs when run with (a) Rules-of-Thumb settings, (b) CBO-suggested settings using a job profile obtained from running the job on the production cluster, and (c) CBO-suggested settings using a job profile obtained from running the job on the development cluster.

program over a small dataset is sufficient for the CBO to find good configuration settings for the program when it is run over larger datasets.

The second common use-case we consider in our evaluation is the use of a development cluster for generating job profiles. In many companies, developers use a small development cluster for testing and debugging MapReduce programs over small (representative) datasets before running the programs, possibly multiple times, on the production cluster. For the experiments reported here, our development cluster was a Hadoop cluster running on 4 Amazon EC2 nodes of the c1.medium type. We profiled all MapReduce programs listed in Table 7.1 on the development cluster. For profiling purposes, we used 10% of the original dataset sizes from Table 7.1 that were used on our 16-node (production) cluster of c1.medium nodes.

Figure 7.7 shows the running times for each MapReduce job j when run with the CBO settings that are based on the job profile obtained from running j on the development cluster. For comparison purposes, we also profiled the MapReduce jobs when run on the production cluster, and then asked the CBO for the best configuration settings. We observe from Figure 7.7 that, in most cases, the performance

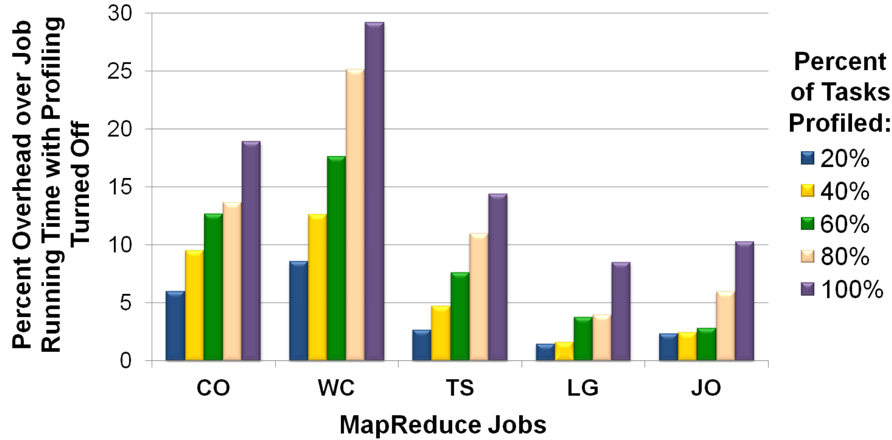


FIGURE 7.8: Percentage overhead of profiling on the execution time of MapReduce jobs as the percentage of profiled tasks in a job is varied.

improvement achieved over the Rules-of-Thumb settings is almost the same; irrespective of whether the CBO used the job profile from the development cluster or the production cluster.

Therefore, when the dataflow proportionality holds, obtaining a job profile by running the program over a small dataset in a development cluster is sufficient for the CBO to find good configuration settings for when the program is run over larger datasets in the production cluster. We would like to point out that this property is very useful in elastic MapReduce clusters, especially in cloud computing settings: when nodes are added or dropped, the job profiles need not be regenerated.

4. Approximate job profiles through sampling: Profiling causes some slowdown in the running time of a MapReduce job j. To minimize this overhead, the Profiler can selectively profile a random fraction of the tasks in j. For this experiment, we profiled all MapReduce jobs listed in Table 7.1 while enabling profiling for only a random sample of the tasks in each job. As we vary the percentage of profiled tasks in each job, Figure 7.8 shows the profiling overhead by comparing against the same job running with profiling turned off. For all MapReduce jobs, as the percent-

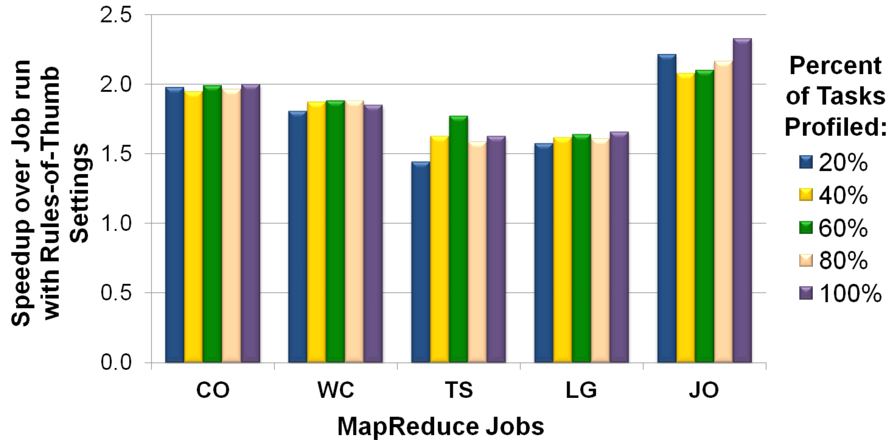


FIGURE 7.9: Speedup over the job run with Rules-of-Thumb settings as the percentage of profiled tasks used to generate the job profile is varied.

age of profiled tasks increases, the overhead added to the job's running time also increases (as expected). It is interesting to note that the profiling overhead varies significantly across different jobs. The magnitude of the profiling overhead depends on whether the job is CPU-bound, uses a Combiner, uses compression, as well as the job configuration settings.

Figure 7.9 shows the speedup achieved by the CBO-suggested settings over the Rules-of-Thumb settings as the percentage of profiled tasks used to generate the job profile is varied. In most cases, the settings suggested by CBO led to nearly the same job performance improvements; showing that the CBO's effectiveness in finding good configuration settings does not require that all tasks be profiled. Therefore, by profiling only a small fraction of the tasks, we can keep the overhead low while achieving high degrees of accuracy in the collected information.

7.3 Cost-based Optimization of MapReduce Workflows

For a given MapReduce workflow W and cluster resources r, a Workflow Optimizer must enumerate and search efficiently through the high-dimensional space of configuration parameter settings (making appropriate what-if calls) in order to find the

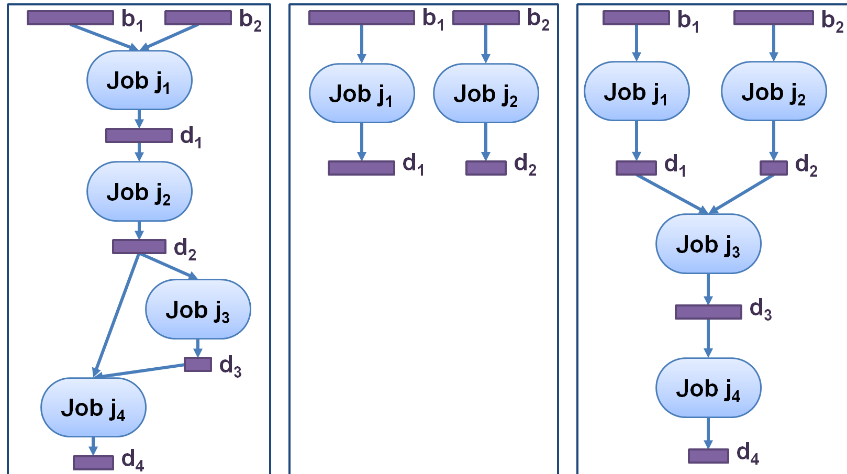


FIGURE 7.10: MapReduce workflows for (a) Query H4 from the Hive Performance Benchmark; (b) Queries P1 and P7 from the PigMix Benchmark run as one workflow; (c) Our custom example.

best ones. Even though the role of a Workflow Optimizer might seem similar to the corresponding role of a Job Optimizer, there exist numerous challenges in optimizing MapReduce workflows with multiple jobs, dependencies, and potential interactions. The primary challenge is that the space of configuration parameter settings for a workflow W comprises the huge cartesian product of the individual configuration spaces of all jobs in W. We have developed optimizers that traverse this space efficiently based on a characterization of the dataflow-based and resource-based interactions that can arise in W. We will begin by introducing two experimental results that illustrate the core insights behind the contributions we make in this Section.

7.3.1 Dataflow and Resource Dependencies in Workflows

The jobs in a workflow exhibit *dataflow dependencies* because of producer-consumer relationships as well as *cluster resource dependencies* because of concurrent scheduling. The dataflow and cluster resource dependencies can result in significant interactions between the configuration parameter settings for the jobs in a MapReduce workflow. Figure 7.10 shows some example MapReduce workflows that exhibit both

types of dependencies. We discuss the dependencies and the consequent interactions next.

Dataflow dependencies in workflows and consequent (dataflow-based) interactions: We first consider a MapReduce workflow with four jobs for Query H4 from Facebook's *Hive Performance Benchmark*, modeled after the work by Pavlo et al. to compare large-scale analytics systems (Pavlo et al., 2009). The workflow processes 110GB of data on a 21-node Hadoop cluster (1 master and 20 slaves) on Amazon EC2 nodes. (Section 7.3.3 gives the details of the cluster.)

Figure 7.11 shows the resulting performance of the workflow when optimized by three different approaches:

- *Rules of Thumb:* The "Rules-of-Thumb" settings are based on manual tuning of the jobs by following popular rules that expert Hadoop administrators use to set configuration parameters for MapReduce jobs (e.g., Lipcon (2009)).

- *Job-level Optimizer:* This automated and cost-based optimizer is a natural extension to the Job Optimizer presented in Section 7.2 above. The Job-level Optimizer, which goes in topological sort order through the jobs in the workflow, invokes the Job Optimizer to optimize each job independent of all other jobs.

- *Workflow Optimizer:* This automated and cost-based optimizer is one of the new optimizers we describe in Section 7.3.2.

Figure 7.11 shows that the optimized workflow given by the Workflow Optimizer is around 3x faster than the manually-optimized workflow and 2.5x faster than the workflow optimized by considering jobs independently.

The root cause of this performance gap comes from the *dataflow dependencies* present in the workflow. Note from Figure 7.10(a) that the output dataset of job j_1

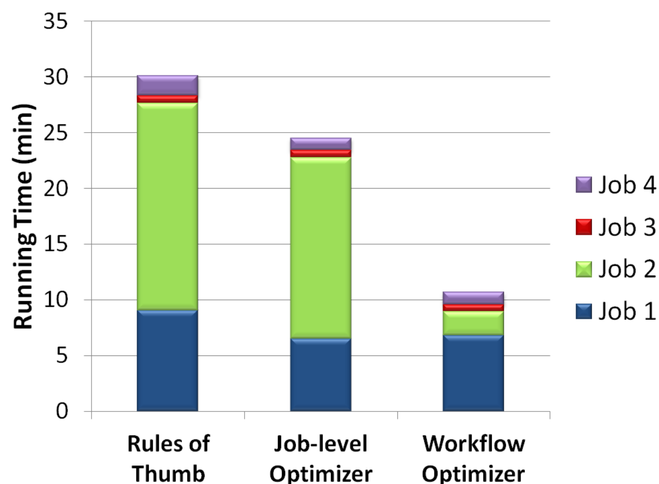


FIGURE 7.11: Execution times for jobs in the workflow from Figure 7.10(a) when run with settings suggested by (a) popular Rules of Thumb; (b) a Job-level Workflow Optimizer; (c) an Interaction-aware Workflow Optimizer.

in the workflow forms the input dataset of job j_2. As a result, there is a potential for *interaction* between the configuration parameter settings of j_1 and the configuration parameter settings of j_2, caused by the dataflow dependency between these jobs. An interaction between the configuration parameter settings of j_1 and j_2 means that the performance given by a configuration setting c_2 for j_2 will depend on the configuration setting c_1 used for j_1. Thus, the jobs cannot be optimized independently without running the risk of finding suboptimal configurations; which is what happened in Figure 7.11.

Specifically, the 4.6GB output dataset produced by j_1 compresses extremely well. Thus, both the Job-level Optimizer and Rules-of-Thumb chose to enable data compression for j_1's output; which is clearly optimal for j_1 in isolation, and results in 436MB of output written to HDFS. When j_2 is executed later, it will process its 436MB input dataset using two map tasks due to the 256MB block size. As a result, each map task in j_2 will have to process 2.3GB of uncompressed data. Thus, j_2 will make poor use of the cluster resources due to its low degree of parallelism as well as

excessive local I/O due to spills and merges in the map tasks.

On the other hand, the Workflow Optimizer takes the potential of interactions into account when dataflow dependencies exist. It automatically realizes (based on estimated costs) that a slightly suboptimal configuration for j_1 can result in a major performance boost for j_2 and give excellent performance overall by making the best use of cluster resources. The Workflow Optimizer disables output data compression for j_1, and uses 35 map tasks in j_2 to process the 4.6GB of resulting input data. Each map task in j_2 processes only around 132MB and produces 35MB; giving the 2.5-3x overall speedup.

Another possible execution plan for jobs j_1 and j_2 would be to enable output data compression for j_1 (as suggested by the Job-level Optimizer), but force a larger number of map tasks for job j_2 (as suggested by the Workflow Optimizer). This plan is possible only when two conditions are met: (i) the compression format used for j_1's output is "splittable", i.e., parts of a single compressed file can be read independently, and (ii) the *InputFormat* [1] used to manage how the input data is split and read by the map tasks allows the user to specify custom split points. Even though splittable compression formats can be used (e.g., LZO, Bzip2), only some InputFormats have parameters for specifying how to split the data. In addition, these InputFormat parameters are typically treated as hints and are not always respected. Hence, these parameters are not included in the optimization space of any of our optimizers, and consequently, such an execution plan is not possible

Resource dependencies in workflows and consequent (resource-based) interactions: Next, we consider the two-job workflow consisting of queries P1 and P7 from Yahoo!'s PigMix benchmark (Dai, 2011) run over 400GB of data. These two jobs can run concurrently in the cluster since there is no dataflow dependency path

[1] InputFormat describes the input specification for a MapReduce job running in Hadoop and is customizable by the user.

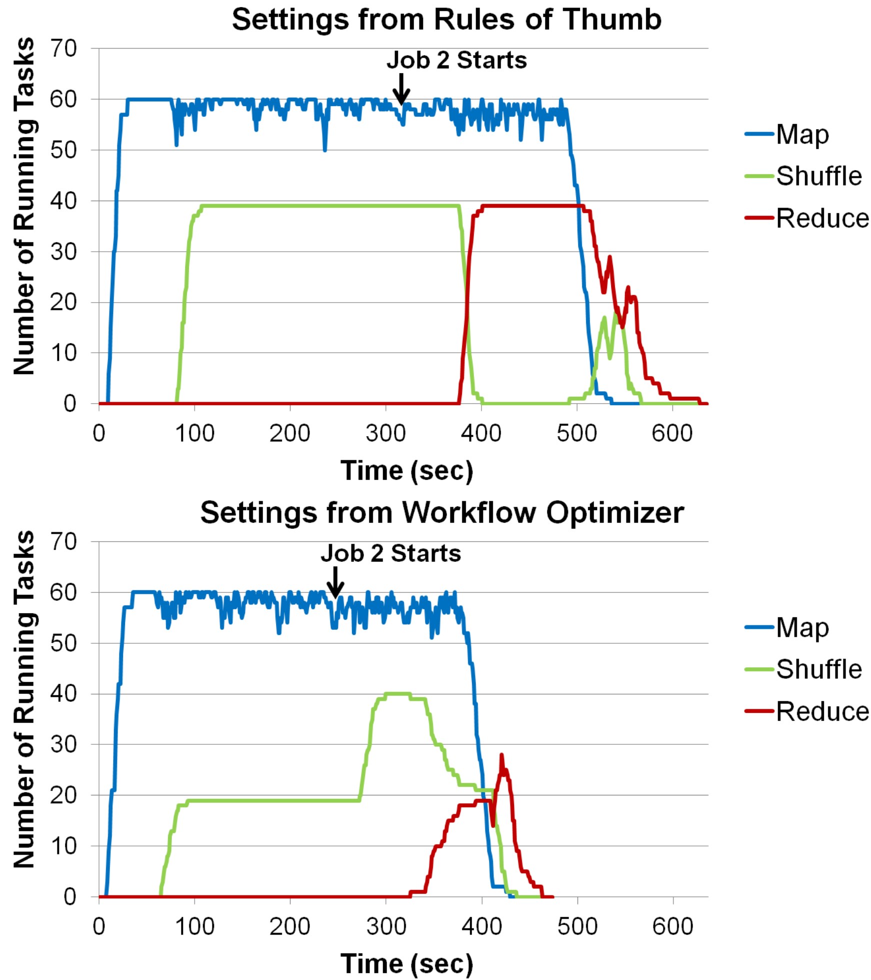


FIGURE 7.12: Execution timeline for jobs in the workflow from Figure 7.10(b) when run with settings suggested by (a) popular Rules of Thumb; (b) an Interaction-aware Workflow Optimizer.

from one to the other. Figures 7.12(a) and (b) show respectively the performance of this workflow when optimized manually using Rules of Thumb and automatically with the Workflow Optimizer. The figures are timelines showing the number of tasks running concurrently from both jobs. Note that reduce tasks start by performing a *Shuffle* of the map output data before entering the *Reduce* phase[2].

The crucial difference between the configuration settings generated by Rules of Thumb and the Workflow Optimizer is in the number of reduce tasks. A widely-used

[2] Recall the phases of MapReduce job execution discussed in Section 3.1

rule of thumb sets the number of reduce tasks to 36 (which is 90% of the cluster's maximum reduce capacity of 40 reduce task slots) for both jobs, resulting in the following execution seen in Figure 7.12(a):

- Job j_1's map tasks start, followed soon after by its reduce tasks. The cluster's maximum map capacity is 60 map task slots, so job j_1's 260 map tasks will run roughly in 5 waves.
- Job j_2's map tasks start once all of job j_1's map tasks complete and free the map task slots. However, job j_2's reduce tasks cannot start since most reduce task slots are hoarded by job j_1.[3]

A *cluster resource dependency* exists here between jobs that can run concurrently. This dependency causes an interaction between the configuration parameter settings of the two jobs. Our interaction-aware Workflow Optimizer chooses to set j_1 and j_2 to have 19 and 21 reduce tasks respectively. Both jobs will run slower individually than if they had 36 reduce tasks since each reduce task will now process more data. However, the lower degrees of parallelism enable both jobs to together utilize the cluster resources better, and complete the workflow 1.34x faster. In particular, soon after job j_2 begins execution, j_2's reduce tasks will get scheduled on the 21 available reduce slots (since j_1 only runs 19 reduce tasks and the cluster has a total of 40 reduce slots), and will start shuffling data from j_2's completed map tasks while j_1's reduce tasks are still executing (see Figure 7.12).

Table 7.3 illustrates a space of optimization choices missed by Rules of Thumb and the Job-level Optimizer because they do not consider interactions caused by resource dependencies. The table shows the number of reduce tasks selected by the

[3] Similar hoarding problems have been observed by others (Zaharia et al., 2010). However, schedulers like the Fair-Share Scheduler will not solve this particular problem because both jobs come from the same user/workflow and have identical priorities.

Table 7.3: Number of reduce tasks chosen and speedup over Rules-of-Thumb settings by the Workflow Optimizer for the two jobs in the workflow from Figure 7.10(b) as we vary the total input size.

Data Size	#Tasks (Job j_1)	#Tasks (Job j_2)	#Tasks (Total)	Speedup (Job j_1)	Speedup (Job j_2)	Speedup (Total)
200GB	26	8	34	1.25	1.30	1.30
300GB	24	16	40	1.24	1.27	1.25
400GB	19	21	40	1.31	1.35	1.34
500GB	38	20	58	1.22	1.25	1.24

Workflow Optimizer based on estimated costs, and resulting speedup, as we vary the input data size from 200GB to 500GB. For smaller data sizes, a lower degree of parallelism that permits both jobs to execute in a single concurrent reduce wave is better. As data sizes increase, it becomes better to let jobs utilize all reduce slots. The Workflow Optimizer improves performance by more than 1.2x in all cases.

The above experimental results illustrate the importance of optimizing configuration parameters for MapReduce workflows in an interaction-aware manner. Efficient solutions for this problem will help improve the performance of the entire MapReduce stack automatically, irrespective of the higher-level interface used to specify workflows.

7.3.2 MapReduce Workflow Optimizers

MapReduce workflow optimization is defined as:

> Given a MapReduce workflow W that will run the jobs $\{j_i\}$ in the job graph G_W on input datasets $\{b_i\}$ and cluster resources r, find the configuration parameter settings c_i for each job j_i in W that minimizes the overall execution time of W.

For this purpose, the Workflow Optimizer is given three inputs:

1. The workflow profile generated for W by the Profiler; discussed in Section 4.1.

2. The properties of the base input datasets on which W will be run; discussed in Section 3.1.

3. The cluster setup and resource allocation that will be used to run W; discussed in Section 3.1.

The Workflow Optimizer's role is to enumerate and search efficiently through the high-dimensional space of configuration parameter settings S_W, making appropriate calls to the What-if Engine, in order to find good configuration settings for each job in W.

Workflow Optimization Space: Recall the set of configuration parameter settings for individual MapReduce jobs discussed in Section 3.1. This set constitutes the *Job-level Optimization Space* S_j for a single MapReduce job j. The *Workflow Optimization Space* S_W for the entire workflow W is the cartesian product of the job-level optimization spaces for each job in W. That is, $S_W = \prod_{j \in W} S_j$.

The above cartesian product is the result of the multiple dependencies that can exist among the jobs in a MapReduce workflow. Consider our running example workflow from Figure 7.10(c). Jobs j_1 and j_2 have a resource dependency and can be scheduled to run concurrently in the cluster. Therefore, the choice for the number of reduce tasks in j_1 can directly affect the performance of j_2, and consequently, the choice for the number of reduce tasks in j_2. Since job j_3 exhibits a dataflow dependency with both j_1 and j_2, the number of reduce tasks in the two jobs—which determines the data properties for the derived datasets d_1 and d_2—can affect the choice for the map-side parameters of j_3. Hence, these choices must be made together.

In theory, a choice for any configuration parameter in a particular job can influence the choice of any other configuration parameter in any other job in the

same workflow. However, in practice—and primarily due to the specific programming model of the MapReduce framework—arbitrary interactions among parameters across multiple jobs are rare (if present at all). We exploit this property in our workflow optimization process below.

Workflow Optimization Process: A straightforward approach for the optimization process is to apply enumeration and search techniques to the full optimization space S_W over the entire directed acyclic graph (DAG) G_W. However, the high dimensionality of S_W renders this approach impractical. More efficient search techniques can be developed using a divide-and-conquer approach: G_W is divided into (possibly overlapping) subgraphs, denoted $G_W^{(i)}$, with lower-dimensional subspaces $S_W^{(i)}$, such that the globally-optimal choices in S_W can be found by composing the optimal choices found for each $S_W^{(i)}$.

Each $G_W^{(i)}$ along with the corresponding $S_W^{(i)}$ defines an *Optimization Unit*. The core idea behind an optimization unit is to bring together a set of related decisions that depend on each other, but are independent of the decisions made at other optimization units. In other words, the goal is to break down the large space S_W into independent subspaces, such that $S_W = \bigcup S_W^{(i)}$. Consider again the example workflow from Figure 7.10(c). The decision for the number of reduce tasks for the first job does not affect the optimizations that are applicable to the last job in the workflow; thereby, these optimizations can be made independently.

Within each optimization unit, a Workflow Optimizer is responsible for enumerating and evaluating the different configuration settings applicable to the jobs within the unit. By making appropriate calls to the What-if Engine, the Workflow Optimizer can evaluate the performance of jobs for different settings in order to make the optimal decisions.

Overall, a Workflow Optimizer follows a two-step process:

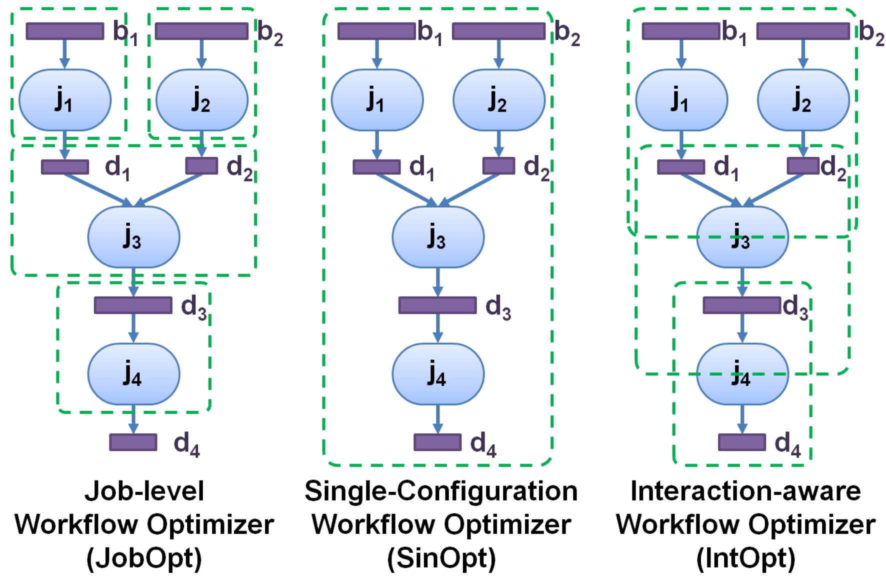


FIGURE 7.13: The optimization units (denoted with dotted boxes) for our example MapReduce workflow for the three Workflow Optimizers.

1. Given the MapReduce workflow DAG G_W, the Optimizer will identify the optimization units—each consisting of one or more MapReduce jobs—and build a DAG of optimization units G_U based on G_W.

2. The Optimizer will traverse the graph G_U in topological sort order to optimize each optimization unit in turn, and then combine the unit-optimal settings in order to build the globally-optimal execution strategy for W.

We have designed and developed three Workflow Optimizers that use different definitions of optimization units (shown in Figure 7.13) to (possibly) find the optimal settings from the optimization space. The three Optimizers demonstrate the spectrum of optimization techniques that are available for workflow optimization as well as different ways of dividing the workflow optimization space.

Job-level Workflow Optimizer

Given a MapReduce workflow W, a straightforward approach to optimize W is to optimize each job in W independently of the other jobs. This approach is employed by the *Job-level Workflow Optimizer* (denoted *JobOpt*).

Optimization unit: Each optimization unit $U^{(i)}$ consists of a single MapReduce job, while the optimization space $S_W^{(i)}$ consists of only the job-level configuration parameters that control the execution of the single MapReduce job.

Search strategy within an optimization unit: Given the MapReduce job j in $U^{(i)}$ to be run on input data d and cluster c, the Optimizer must find the setting of configuration parameters from $S_W^{(i)}$ that minimizes the running time of j. For this purpose, the Optimizer is using the same enumeration and search strategy used by the Job Optimizer presented in Section 7.2. The Optimizer first divides the job-level optimization space S_j into two subspaces: the *Map-side Optimization Space* S_m containing parameters that predominantly affect map task execution, and the *Reduce-side Optimization Space* S_r containing parameters that predominantly affect reduce task execution. Hence, $S_j = S_m \cup S_r$. The two subspaces for map tasks and reduce tasks respectively can be optimized independently by using the Recursive Random Search (RRS) method twice.

Overall optimization process: Since each optimization unit consists of only one job, the DAG of optimization units G_U has the same structure as the input DAG of jobs G_W. *JobOpt* optimizes each unit $U^{(i)}$ in G_U in topological order. We will describe the traversal process using the running example in Figure 7.13(a). First, the Optimizer will find the best configuration settings for job j_1 that processes the workflow's base dataset b_1. Job j_1 produces the first derived dataset d_1, which is later consumed by job j_3. Given the configuration settings selected for j_1, the What-

if Engine can estimate the properties of d_1 (e.g., number of files in d_1, size per file, and use of compression). Next, the Optimizer will optimize job j_2 and predict the data properties for its derived dataset d_2; and the process repeats. At the end, the Optimizer will have the optimal configuration settings for each job in the workflow.

The main drawback of *JobOpt* is that it does not consider how the configuration settings for a job affect the performance of concurrent or successor jobs. This limitation is addressed by both the Single-configuration and the Interaction-aware Workflow Optimizers below.

Single-configuration Workflow Optimizer

Many higher-level systems (e.g., Pig and Hive) expose structured interfaces for users to express workflows with, and then automatically generate the corresponding MapReduce jobs to execute on the cluster. This process naturally hides the MapReduce jobs from the users, inevitably preventing them from specifying different parameter settings for different jobs in the same workflow[4]. Motivated from this fact, the *Single-configuration Workflow Optimizer* (denoted *SinOpt*) tries to find a single set of configuration settings that must be the same for all jobs in the workflow; and subject to this constraint, the one that can lead to the best overall workflow performance.

Optimization unit: *SinOpt* defines only one optimization unit that consists of the entire graph G_W. The workflow optimization space S_W is reduced down to a single job-level optimization space S_j since all jobs will use the same configuration settings during execution.

Search strategy within an optimization unit: Similar to *JobOpt*, *SinOpt* will

[4] The number of reduce tasks is an exception. The user can indirectly specify a different number of reduce tasks for different MapReduce jobs by specifying a different degree of parallelism for higher-level declarative constructs.

make appropriate what-if calls with settings c of the configuration parameters selected through RRS over the optimization space S_j. Each what-if call will go through the process of (i) estimating the virtual profile for each job in topological order, (ii) simulating the job execution, and (iii) estimating the derived dataset properties, as described in Chapter 6.

Overall optimization process: Since *SinOpt* consists of a single optimization unit, the overall optimization process simply involves optimizing that unit.

Unlike *JobOpt*, *SinOpt* does take into consideration both resource and dataflow dependencies among the jobs in the workflow. However, the optimization space it considers is extremely limited and—as we will see in the evaluation Section 7.3.3—does not always lead to the best settings.

Interaction-aware Workflow Optimizer

The *Interaction-aware Workflow Optimizer* (denoted *IntOpt*) addresses the limitations of the other two optimizers based on the following intuition: when two jobs j_i and j_k are separated by one or more jobs in the workflow graph (i.e., the dependency path between j_i and j_k contains at least one other job), then the effect of j_i on the execution of j_k diminishes rapidly in practical settings. Hence, decisions made for j_i can be made independently from decisions made for j_k. For example, the choice for compressing the output of job j_1 in our example workflow from Figure 7.10(c), will not affect the choice for using a combine function or some other setting in j_4.

On the other hand, compressing j_1's derived dataset d_1 will affect the performance of job j_3, since j_3 will have to uncompress d_1 to process it. A compressed input is likely to affect the number of map tasks, as well as the settings of memory-related parameters in j_3. The Shuffle phase in j_3, however, acts as a natural barrier and synchronization step in the MapReduce framework, where the reduce tasks have to wait for all the map tasks to complete execution before processing the shuffled data.

Hence, the execution of job j_1, as well as the map phase of j_3, will most likely not affect the reduce phase of j_3.

In summary, *IntOpt* builds the optimization units based on the conjecture that jobs running concurrently in the cluster and the map phase of their successor jobs should be optimized together. Similar to the *JobOpt*, the *IntOpt* follows a divide-and-conquer approach for optimizing a single MapReduce job to explicitly separate it into its map tasks and reduce tasks.

Optimization unit: An optimization unit $U^{(i)}$ consists of (i) the map tasks of a set of resource-dependent (i.e., concurrent) jobs, (ii) their reduce tasks, and (iii) the map tasks of the corresponding successor jobs. Figure 7.13(c) shows a pictorial representation of the interaction-aware optimization units. The first unit consists of the map and reduce tasks of jobs j_1 and j_2, as well as the map tasks of job j_3.

The optimization space $S_W^{(i)}$ is divided into two subspaces: the first subspace represents the cross product of the map-side optimization spaces of the resource-dependent jobs; and the second subspace represents the cross product of the reduce-side optimization spaces of the resource-dependent jobs with the map-side optimization spaces of the successor jobs.

Search strategy within an optimization unit: *IntOpt* uses RRS twice to enumerate and search over the two subspaces in $S_W^{(i)}$. For each enumerated point in the first subspace of $S_W^{(i)}$, the What-if Engine will (i) estimate the virtual profiles for the map tasks in the resource-dependent jobs, (ii) simulate their execution, and (iii) estimate their combined running time. By using RRS the first time, *IntOpt* determines the map-side parameter settings that give near-optimal performance for the map tasks.

For each enumerated point in the second subspace of $S_W^{(i)}$, the What-if Engine will (i) estimate the virtual profiles for the reduce tasks in the resource-dependent jobs,

(ii) estimate the data properties for the derived datasets, (iii) estimate the virtual profiles for the map tasks in the successor jobs, and (iv) estimate the total running time. The second RRS will determine the reduce-side parameter settings for the resource-dependent jobs, completing the search in the current optimization unit.

Overall optimization process: *IntOpt* traverses the DAG of optimization units G_U in topological order. Each unit is responsible for optimizing the resource-dependent jobs it contains, as well as estimating the data properties of the derived data sets. For example, the first optimization unit in Figure 7.13(c) will optimize jobs j_1 and j_2, and estimate the properties for derived datasets d_1 and d_2. The following unit will optimize job j_3 and estimate the properties for d_3; and so on.

Overall, *IntOpt* is able to handle both resource and dataflow dependencies while dividing the high-dimensional workflow optimization space into smaller, more manageable optimization subspaces.

Static and Dynamic Optimization

We have seen how the three Workflow Optimizers traverse the workflow graph G_W, create optimization units, and optimize each job in G_W. The next question in hand is *when* to optimize the workflow. The optimization process can either happen completely before any job gets submitted for execution in the cluster, or be interleaved with job executions. The former approach is called *Static Optimization*, while the latter is called *Dynamic Optimization*.

Static Optimization: With static optimization, all optimization units are optimized at once before any jobs are submitted for execution in the cluster. In this case, only the data properties for the base datasets are known, while the data properties for all derived datasets must be estimated during the job optimization process. Static optimization is useful in two cases:

1. The user wants to get recommendations for parameter settings, without actually submitting the workflow for execution.

2. The user is trying to optimize the workflow using hypothetical base datasets and/or cluster resources.

Dynamic Optimization: The core idea behind dynamic optimization is to optimize each job immediately before it gets submitted to the cluster. Since all jobs are organized into optimization units, however, we only optimize each optimization unit U immediately before the first job j in U gets submitted to the cluster. At this point, all derived datasets processed by j have already been generated (otherwise, j would not have been ready for submission) and their properties can be automatically collected from the cluster; no data estimation is needed.

IntOpt is still doing data estimation as part of the optimization process within a single unit. Consider the first optimization unit of *IntOpt* in Figure 7.13(c). Finding the best reduce-side parameters for jobs j_1 and j_2 requires estimating the data properties for d_1 and d_2. However, those data estimates will not be used by the second optimization unit containing j_3; thereby, any possible estimation mistakes will not affect the optimization process of j_3 and the following MapReduce jobs.

In summary, we have designed and developed three different Workflow Optimizers: Job-level, Single-configuration, and Interaction-aware. Since *SinOpt* is responsible for finding a single set of configurations for all jobs in the workflow, it does not support Dynamic optimization. *JobOpt* and *IntOpt* on the other hand, support both Static and Dynamic Optimization.

7.3.3 Evaluating Cost-based Workflow Optimization

In our experimental evaluation, we used two different Hadoop clusters. The first cluster was running on 21 Amazon EC2 nodes of the m1.large type. Each node has

Table 7.4: MapReduce workflows and corresponding dataset sizes on two clusters for the evaluation of the Workflow Optimizer.

Benchmark	Dataset Size	
	Amazon Cluster	Yahoo! Cluster
TPC-H Benchmark	100 GB	200 GB
PigMix Benchmark	500 GB	1000 GB
Hive Performance Benchmark	100 GB	200 GB
Custom Benchmark	500 GB	1000 GB

7.5 GB memory, 2 virtual cores, 850 GB local storage, and is set to run at most 3 map tasks and 2 reduce tasks concurrently. Thus, the cluster can run at most 60 map tasks in a concurrent map wave, and at most 40 reduce tasks in a concurrent reduce wave. The second cluster is a cluster used for testing purposes at Yahoo! Research and was running on 50 machines arranged in 3 racks. Each machine has 4 GB memory, 2 cores, 2x300 GB disks, and is set to run at most 2 map tasks and 2 reduce tasks concurrently. We will refer to the two clusters respectively as the *Amazon* and the *Yahoo!* clusters.

For our evaluation, we selected representative MapReduce workflows from four different benchmarks, listed on Table 7.4. These particular benchmarks were chosen to have broad industry-wide relevance:

1. *TPC Benchmark-H (TPC-H)* is a decision support benchmark composed of a suite of 22 business oriented ad-hoc queries (TPC, 2009).

2. *PigMix* consists of 17 queries that span the spectrum of representative queries executed on Yahoo!'s production clusters (Dai, 2011).

3. *Hive Performance Benchmark* is modeled after the queries and data used by Pavlo et al. in comparing large-scale analytics systems. It consists of 4 queries performing log analysis (Pavlo et al., 2009).

4. *Custom Benchmark* contains 3 queries that heavily utilize user-defined func-

tions (UDFs). UDFs are commonly found in practice but none of the above benchmarks include any.

Table 7.4 also lists the dataset sizes used for each benchmark for our two clusters. All queries were expressed in Pig Latin and submitted using Pig. The unoptimized MapReduce workflows are executed using Rules-of-Thumb settings found in Lipcon (2009). Unless otherwise noted, the Dynamic Interaction-aware Workflow Optimizer is the default optimizer used in our evaluation. Our experimental methodology is as follows:

1. We evaluate the effectiveness and efficiency of the Workflow Optimizer in finding good configuration settings to use for each job in a MapReduce workflow.

2. We compare the Job-level and Single-configuration Optimizers against the Interaction-aware Optimizer to provide insights into the nontrivial nature of the resource and dataflow dependencies that are present among the jobs in a MapReduce workflow.

3. We compare the Static and Dynamic optimization approaches in terms of both effectiveness and robustness.

1. End-to-end evaluation of the Workflow Optimizer: We evaluate the end-to-end performance of the Dynamic Interaction-aware Optimizer on representative queries selected from the TPC-H, PigMix, and Hive Benchmarks. In order to focus the discussion on the key points of our evaluation, we do not present the results from all workflows from all benchmarks. However, the workflows not presented either exhibit similar results or are trivial, single-job workflows.

Figures 7.14 and 7.15 show the performance improvements achieved by the Workflow Optimizer in the Amazon and Yahoo! clusters, respectively. In all cases, the

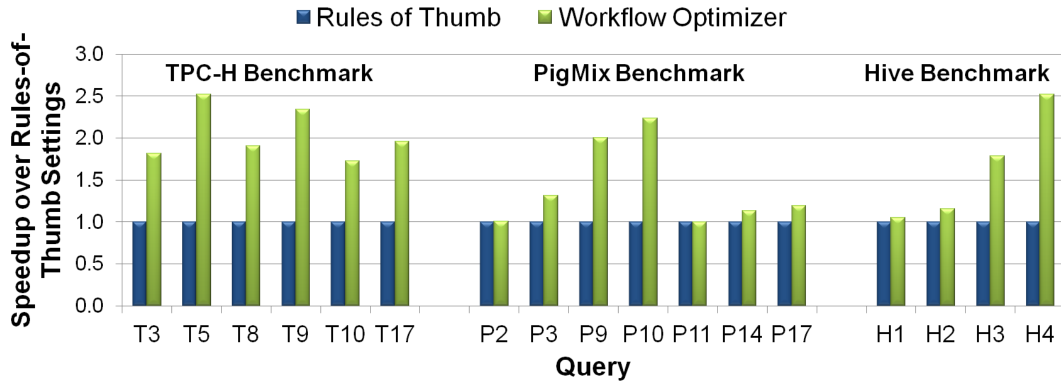


FIGURE 7.14: Speedup achieved over the Rules-of-Thumb settings for workflows running on the Amazon cluster from (a) the TPC-H Benchmark, (b) the PigMix Benchmark, and (c) the Hive Performance Benchmark.

Workflow Optimizer is able to automatically match and surpass the workflow performance obtained when using the Rules-of-Thumb settings, providing over a 2x speedup in multiple cases. The results observed in the two clusters are very similar, albeit the Workflow Optimizer provided slightly lower speedups in the Yahoo! cluster compared to the ones in the Amazon cluster.

It it important to note here that finding the Rules-of-Thumb settings requires a certain level of expertise as well as good knowledge of the internal workings of Hadoop. On the other hand, a nonexpert user can simply use the Workflow Optimizer for automatically obtaining settings that will lead to both better workflow performance as well as better utilization of the cluster resources.

Insights to optimization benefits: There are several different reasons for the optimization benefits we observe in Figures 7.14 and 7.15. In Section 7.3.1, we have already seen two such reasons. The first scenario involved two MapReduce jobs having a dataflow dependency. In that case, compressing the output of the first job, seriously limited the map-phase parallelism of the second job, leading to very poor performance and cluster resource usage. The second scenario involved executing two MapReduce jobs concurrently in the same workflow. Rather than maximizing the

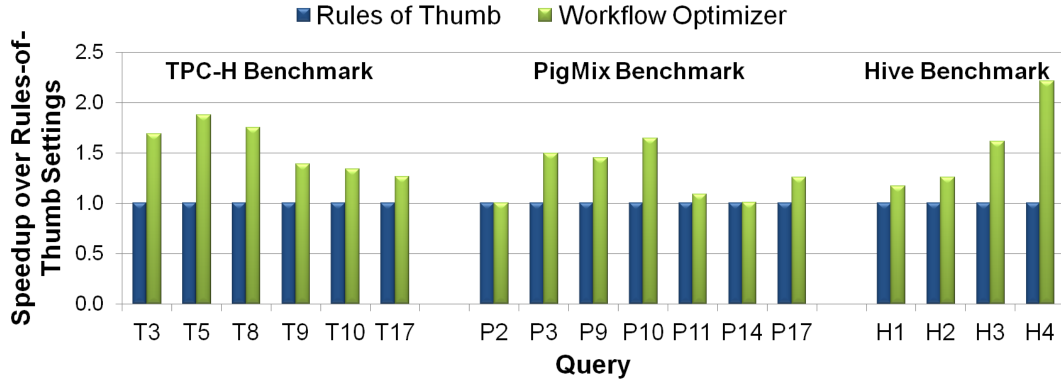


FIGURE 7.15: Speedup achieved over the Rules-of-Thumb settings for workflows running on the Yahoo! cluster from (a) the TPC-H Benchmark, (b) the PigMix Benchmark, and (c) the Hive Performance Benchmark.

reduce phase parallelism of each job independently, it is often better to divide the number of reduce slots among the jobs, in order to achieve a more effective use of the cluster resources.

In order to provide further insights into where these benefits come from, we will drill down into the execution behavior of TPC-H query $T5$. Query $T5$ executes a 6-way join, a group-by aggregation, and sorting, resulting in the generation and execution of 8 MapReduce jobs, j_1 to j_8. Using Rules-of-Thumb settings, $T5$ executes on the Amazon cluster in around 46 minutes. The overall running time of the workflow is dominated by jobs j_3, j_4, and j_5, which run in 15, 15, and 11 minutes respectively. All three jobs execute joins and are, therefore, CPU- and memory-intensive jobs. Rules-of-thumb settings specify the use of map-output compression, which leads to a compression ratio[5] of around 0.3 for the map output of each job. However, the added CPU overhead from the compression does not justify the benefits gained from the reduced I/O—a property that is reflected in the job profiles. By disabling compression and increasing some memory buffers, the Workflow Optimizer is able to reduce the running times of jobs j_3, j_4, and j_5 down to 8, 4, and 3 minutes respectively,

[5] Compression ratio is defined as the ratio of compressed size over uncompressed size.

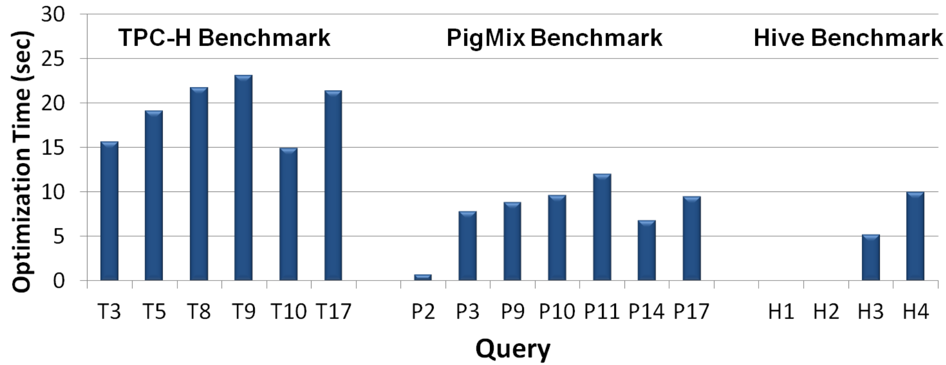


FIGURE 7.16: Optimization overhead for all MapReduce workflows.

leading to a 2.6x speedup in the overall workflow execution time.

In addition, the Workflow Optimizer deemed sufficient to use only 26, 21, and 8 reduce tasks respectively for the three jobs, instead of 36 suggested by the Rules-of-Thumb. Therefore, on top of the performance benefits, the Optimizer's settings also lead to more efficient use of the cluster resources. With a smaller number of reduce tasks, Hadoop avoids the overhead of starting multiple tasks and has more (free) task slots to schedule other jobs on.

Optimization overhead: The second important evaluation metric for an optimizer is the *efficiency* by which it finds the optimal settings. Figure 7.16 shows the time spent performing optimization. Note that the y-axis measures seconds, while all the workflows run in the order of minutes. The average optimization time is 11.1 seconds and the worst time is 23.1 seconds for query $T9$, which executes 8 MapReduce jobs in a non-linear graph. The optimization times for the TPC-H queries are higher compared to the times for the other benchmarks due to the much higher number of MapReduce jobs that have to get optimized. The average optimization times for the other two benchmarks is merely 6.4 seconds. Another interesting observation is the sub-second optimization times for queries $P2$, $H1$, and $H2$. These queries execute map-only jobs, which have a significantly smaller optimization space as the Workflow

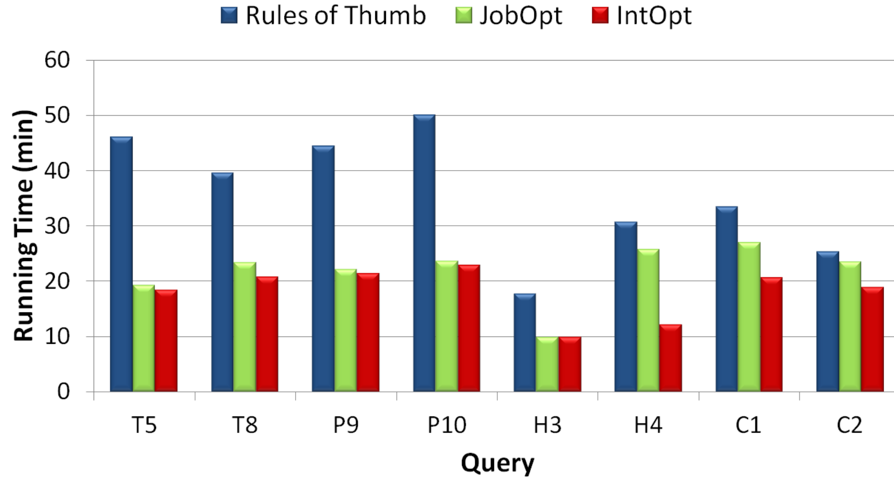


FIGURE 7.17: Running times of queries with settings based on Rules-of-Thumb, the Job-level Workflow Optimizer (JobOpt), and the Interaction-aware Workflow Optimizer (IntOpt).

Optimizer does not need to consider any shuffle- or reduce-related parameters. Overall, the overhead introduced by the optimization process is dwarfed by the possible performance improvements that can be achieved using better configuration settings.

2. Job-level Vs. Interaction-aware Workflow Optimizer: By optimizing each job in isolation, *JobOpt* essentially employs a greedy approach to workflow optimization. Similar to most greedy techniques, *JobOpt* works well in many cases, but can also lead to suboptimal performance in others. In particular, in the absence of significant job interactions, *JobOpt* will perform as well as *IntOpt*. This is evident by the workflow performance observed for many queries seen in Figure 7.17. To avoid repetition, we present results only for the two workflows with the largest number of jobs from each benchmark.

Many of the MapReduce workflows generated for our benchmark queries consist of a linear sequence of MapReduce jobs. Thus, resource interactions are not widely present. However, in the presence of resource or dataflow interactions, *JobOpt* may select settings that are optimal for one job but hurt the performance of another job

FIGURE 7.18: Optimization times for the Job-level Workflow Optimizer (JobOpt) and the Interaction-aware Workflow Optimizer (IntOpt).

in the same workflow. Hive query $H4$ is a prime example and was discussed in detail in Section 7.3.1.

We observe similar patterns with the two queries from our Custom Benchmark (see Figure 7.17). Note that the Custom Benchmark was developed in order to evaluate the optimizers in the presence of jobs containing complex UDFs. Custom query $C1$ executes 2 MapReduce jobs with a dataflow dependency. Both jobs include UDFs in the reduce tasks and the Combiner. Custom query $C1$ executes the same 2 jobs, but with a resource dependency instead. In both cases, a greedy selection for the number of reducers leads to suboptimal use of cluster resources as well as suboptimal workflow performance. *IntOpt*, on the other hand, is able to recognize the dependencies and select the settings that lead to much better performance.

In terms of optimization time, *JobOpt* is on average 4.5 seconds faster than *IntOpt*. Figure 7.18 shows the optimization times for the two optimizers with *JobOpt* having an average of 9.4 seconds and *IntOpt* having an average of 13.9 seconds. Two reasons account for *IntOpt*'s additional overhead. First, the optimization space for each unit of *IntOpt* is larger than the one for each unit of *JobOpt* (see Section 7.3.2). Second,

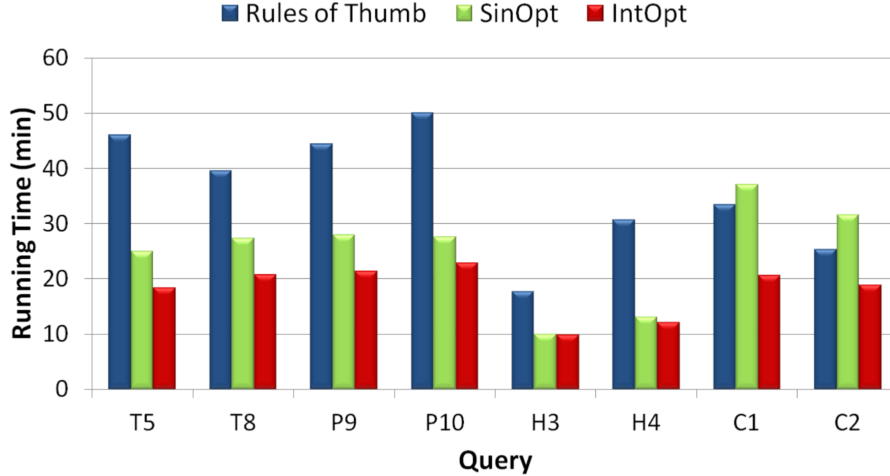


FIGURE 7.19: Running times of queries with settings based on Rules-of-Thumb, the Single-configuration Workflow Optimizer (SinOpt), and the Interaction-aware Workflow Optimizer (IntOpt).

within a single unit, *IntOpt* must estimate the derived data properties with each what-if call, whereas *JobOpt* must do so only once per job.

3. Single-configuration Vs. Interaction-aware Optimizer: Unlike *JobOpt*, *SinOpt* will take job dependencies into account, but it will only consider a single job-level optimization space for the entire workflow. Figure 7.19 shows the workflow performance achieved by *SinOpt* settings compared to the Rules-of-Thumb and the *IntOpt* settings. We observe that, in some cases, selecting the same settings for all jobs can still provide significant performance benefits over the Rules-of-Thumb settings. The main reason behind this observation is that the running time of many workflows is dominated by the running time of 1 or 2 jobs in the workflow. Hence, in these cases, *SinOpt*'s optimization process reduces to finding the best settings for the dominating job(s) in the workflow.

The main scenario for which *SinOpt* can produce suboptimal results is when two or more dominant jobs benefit from conflicting settings. For example, suppose that it is best for one job j_1 in a workflow W to enable the Combiner, whereas it is best

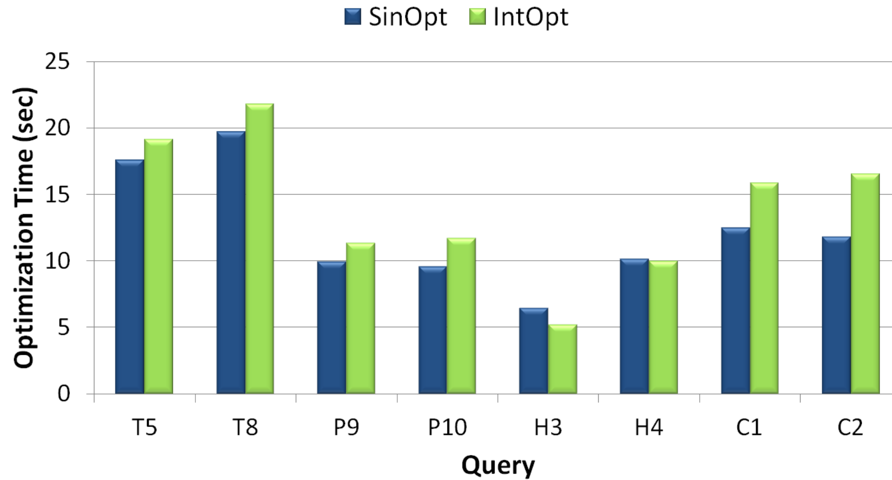


FIGURE 7.20: Optimization times for the Single-configuration Workflow Optimizer (SinOpt) and the Interaction-aware Workflow Optimizer (IntOpt).

for another job j_2 in W to disable it. *SinOpt* is simply not capable of making the best decision for both jobs at the same time. Custom workflow $C1$ exhibits the aforementioned behavior. The Combiner for job j_1 has a record selectivity[6] of 0.78, and thus does not provide sufficient data reduction to justify its CPU overhead. On the other hand, the combiner for j_2 offers substantial data reductions with a record selectivity of 0.31. *SinOpt* selects to disable the Combiner which improves the performance of j_1, but hurts j_2 since j_2 now needs to process more data.

Figure 7.20 shows the optimization times for *SinOpt* and *IntOpt*. On average, the optimization time for *SinOpt* is 12.2 seconds and, thus, only slightly better that the optimization time for *IntOpt* (13.9 seconds). Even though the overall optimization space that *IntOpt* considers is significantly larger than the one for *SinOpt*, recursive random search is still able to efficiently search through the larger space by only adding a very small overhead to the total optimization time.

4. Static Vs. Dynamic optimization: Our final set of experiments targets the

[6] Record selectivity is defined as the ratio of the number of output records over the number of input records.

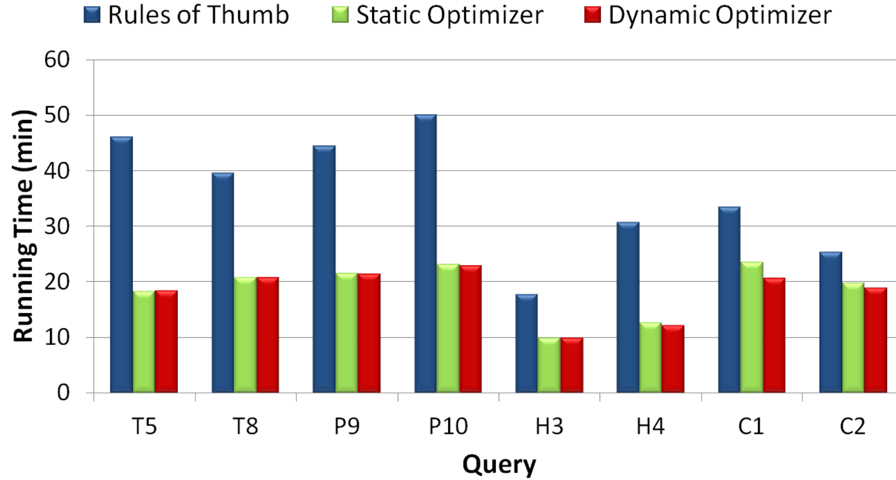


FIGURE 7.21: Running times of queries with settings based on Rules-of-Thumb, and the Static and Dynamic Interaction-aware Workflow Optimizers.

evaluation of the *Static* and *Dynamic* optimization process. As discussed in Section 7.3.2, in Static optimization, the entire workflow is optimized before any job is submitted to the cluster, whereas in Dynamic each optimization unit is optimized immediately before its first job is submitted. Figure 7.21 shows the performance achieved when using the Static and Dynamic Interaction-aware Optimizers. Both optimizers offer remarkably similar performance improvements over the Rules-of-Thumb settings because the derived data estimation process used in Static Optimization is fairly accurate.

In order to study the behavior of the Static Optimizer in the presence of misestimation, we devised Custom workflow $C3$ consisting of 2 MapReduce jobs: The first job j_1 contains a filter UDF for filtering out data, and the second job j_2 processes the data generated by j_1. The filter UDF allows us to externally control the filter ratio[7] of j_1. We executed the workflow multiple times with different filter ratios but each time we told the Optimizers the filter ratio was 0.4. Therefore, the Static Optimizer would estimate the data properties of the derived data incorrectly.

[7] The filter ratio is defined at the ratio of the output data size over the input data size.

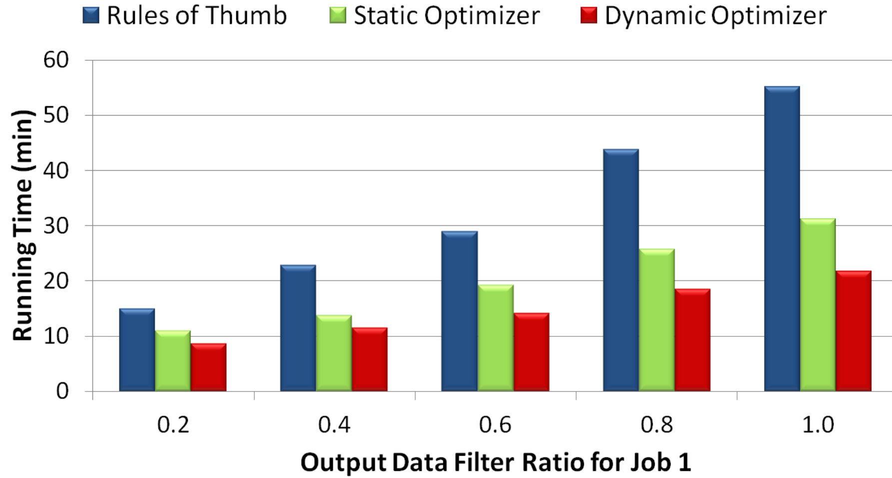


FIGURE 7.22: Running times of queries with settings based on Rules-of-Thumb, and the Static and Dynamic Interaction-aware Workflow Optimizers, as we vary the actual filter ratio of job j_1. The estimated filter ratio is always 0.4.

Figure 7.22 shows the workflow performance achieved by the Static and Dynamic Optimizers against the Rules-of-Thumb as we vary the filter ratio of job j_1. When the filter ratio is set for example to 0.6, the Static Optimizer will overestimate the size of j_1's output data by a filter ratio of 0.2, which roughly corresponds to 15 GB of data. Despite the data mis-estimation, the Static Optimizer is still able to improve upon the Rules-of-Thumb settings in all cases, showcasing the robustness of the selected settings. The Dynamic Optimizer on the other hand, will dynamically observe the properties of the derived dataset and correct any previous mis-estimations, offering greater improvements.

7.4 Cost-based Optimization of Cluster Resources

One of the main purposes of Starfish is to automatically address cluster sizing problems for MapReduce workloads on the cloud. Recall that cluster sizing problems refer to determining the appropriate cluster resources to use—in addition to the MapReduce configuration settings—in order to meet desired requirements on execution time and cost for a given MapReduce workload. The Job and Workflow Optimizers pre-

sented above are responsible for job-level configuration settings, whereas the *Cluster Resource Optimizer* is responsible for the space of possible cluster resources. By using all three components together, Starfish can provide reliable answers to tuning queries.

7.4.1 Cluster Resource Optimizer

The properties used to represent the cluster resources in Starfish include the number of nodes in the cluster, a label per node representing the node type, the cluster's network topology, the number of map and reduce task execution slots per node, and the maximum memory available per task execution slot. From the above properties, only the number of nodes and the node type is included as part of the search space for cluster resources.

As discussed in Section 5.1, our current implementation of the Cluster Resource Optimizer in Starfish does not include the cluster's network topology as part of the search space for cluster resources, since most current cloud providers hide the underlying network topology from clients. In addition, the current tuning query interface does not expose the other cluster-wide configuration parameters either. Empirical evidence has shown that good settings for these parameters are determined primarily by the CPU (number of cores) and memory resources available per node in the cluster.

The Cluster Resource Optimizer is responsible for finding the optimal cluster resources (in addition to job-level configuration settings) to meet desired requirements on execution time and cost for a given MapReduce workflow. In addition to the workflow profile and the input data properties, the input to the optimizer includes a range for the cluster size and a set of labels that represent the resource choices. Given the small discretized space of resource choices (e.g., EC2 node types) offered by cloud platforms, the Cluster Resource Optimizer uses gridding to enumerate all

possible choices in the search space. For each enumerated space point (i.e., a particular cluster size and node type), the Cluster Resource Optimizer uses (i) the Workflow Optimizer to find the optimal configuration settings $\{c_{opt}\}$ for the workflow and (ii) the What-if Engine to estimate the execution time (or cost) of the workflow using $\{c_{opt}\}$. Note that the best configuration settings for the workflow will invariably change if the cluster resources change. Finally, the Cluster Resource Optimizer selects the space point (along with the corresponding configuration settings) with the least estimated time (or cost).

Even though the Cluster Resource Optimizer covers a much smaller search space compared to the other Optimizers, it is still an indispensable component that enables Starfish to support multi-objective cluster provisioning. Starfish uses all Cost-based Optimizers to enumerate and optimize the cluster resources and job configuration parameter settings in tandem.

7.4.2 Evaluating Cost-based Cluster Provisioning

Starfish can reason about the Hadoop job parameter configuration space as well as the cluster resources space. In this section, we evaluate the ability of Starfish to find good cluster and job configuration settings to use for a MapReduce workload under the dual objectives of execution time and cost.

For this evaluation, we used the Amazon EC2 node types c1.medium, m1.large, and m1.xlarge. Table 3.2 lists the resources available for the three node types. For each node type, we used empirically-determined fixed values for the cluster-wide Hadoop configuration parameters—namely, the number of map and reduce task execution slots per node, and the maximum memory available per task slot (shown on Table 6.3). We varied the cluster sizes from 10 to 30 nodes.

The workload we used consists of the MapReduce jobs listed in Table 6.4—namely Word Co-occurrence, Join, LinkGraph, TF-IDF, TeraSort, and WordCount—run one

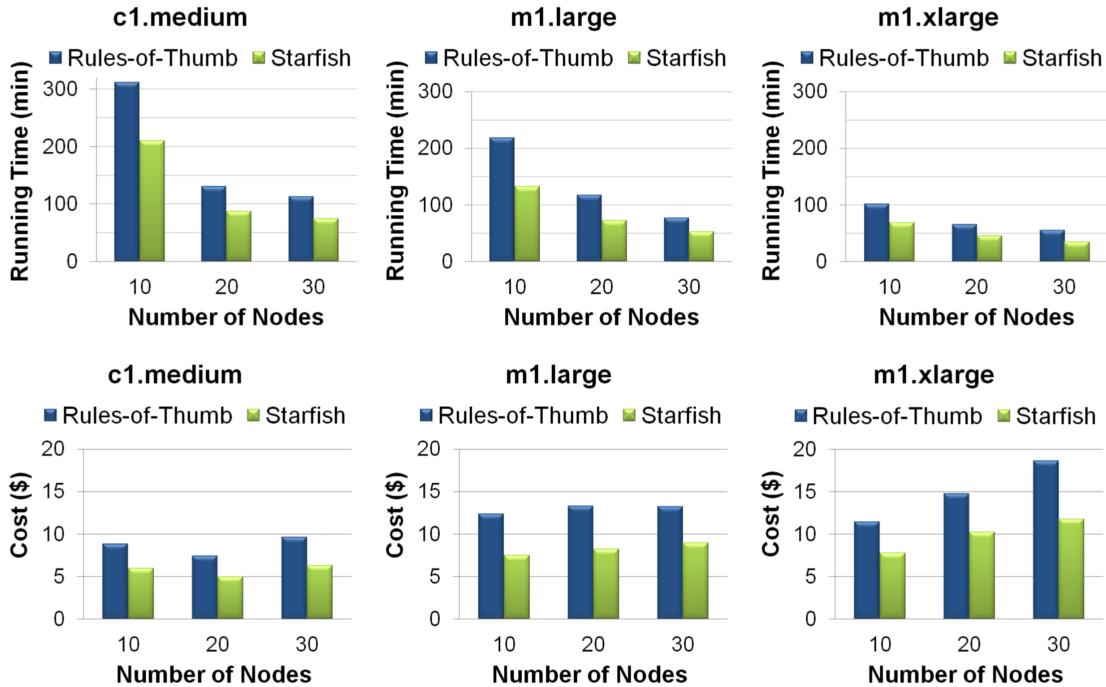


FIGURE 7.23: Running time and monetary cost of the workload when run with (a) Rules-of-Thumb settings and (b) Starfish-suggested settings, while varying the number of nodes and node types in the clusters.

after the other. The job profiles were obtained by running the workload on a 10-node cluster of m1.large EC2 nodes. For monetary cost predictions, Starfish uses a pricing model containing the hourly node costs listed in Table 3.2.

Figure 7.23 shows the running time of the workload when run with the configuration settings suggested by Starfish, across clusters each with a different type of node and number of nodes. Starfish was able to provide up to 1.7x speedup for the workload, which translates into 42% cost savings. Since Starfish is able to reason about the combined cluster resources and configuration space accurately, Starfish is also able to answer general provisioning queries of the form: "What is the best combination of cluster resources and configuration settings to use for running my workload in a way that minimizes execution time (or monetary cost), subject to a maximum tolerable monetary cost (or execution time)?" In our experiment, Starfish was able

to identify correctly that using a 30-node cluster with m1.xlarge nodes would yield the best workload execution time, whereas using a 20-node cluster with c1.medium nodes would minimize the monetary cost.

It is interesting to note the complex interactions between execution times and monetary costs as we vary the number of nodes and node type used in the clusters. As expected, increasing the number of nodes and using more powerful machines lead to better running times. However, the performance improvements are not necessarily linear. Let us consider the 10-node cluster with m1.xlarge nodes. If we use 3x more nodes, then we achieve only 2x performance improvement for the workload, but for only a 1.5x increase in cost. On the other hand, the same 3x increase in the number of nodes for a cluster with m1.large nodes leads to an almost 3x performance improvement with only a 1.2x increase in cost.

Overall, Starfish is able to capture these complex interactions with a good degree of accuracy; so it can help users select the best cluster resources to fit their needs and preferences.

8

An Experiment-driven Approach to Tuning Analytical Queries

Profile-predict-optimize forms the core approach used in this dissertation to address a variety of tunings problems caused by the MADDER principles (recall Section 2.2). The Starfish system discussed in Chapters 4 through 7 applies this approach to automatically tune a MapReduce workload W after observing a single execution of W running on a MapReduce cluster. This application, however, is just one cycle of a more general, self-tuning approach that learns over time and re-optimizes as needed.

We can employ the profile-predict-optimize approach repeatedly; that is, we can collect some targeted profile information, perform optimization, and repeat as needed to perform fine-grained workload tuning in the context of both Database and Dataflow systems. This approach is particularly useful for the tuning scenario where a user is willing to invest some resources upfront for tuning an important workload. We have implemented this approach within the realm of Database systems to improve suboptimal execution plans picked by the query optimizer for queries that are run repeatedly (e.g., by a business intelligence or report generation application).

Traditional query optimizers for Database Systems have predominantly followed a *plan-first execute-next* approach (Selinger et al., 1979): the optimizer uses a search algorithm to enumerate plans, estimates plan costs based on a performance model and statistics, and picks the plan with least estimated cost to execute a given SQL query. While this approach has been widely successful, it causes a lot of grief when the optimizer mistakenly picks a poor plan for a repeatedly run query. Unknown or stale statistics, complex expressions, and changing conditions can cause the optimizer to make mistakes. For example, the optimizer may pick a poor join order, overlook an important index, use a nested-loop join when a hash join would have done better, or cause an expensive, but avoidable, sort to happen (Deshpande et al., 2007). A database administrator (DBA) may have to step in to lead the optimizer towards a good plan (Belknap et al., 2009).

The process of improving the performance of a "problem query" is referred to in the database industry as *SQL tuning*. SQL tuning is also needed while tuning multi-tier services to meet *service-level objectives (SLOs)* on response time or workload completion time (e.g., all reports should be generated by 6:00 AM).

Need for a SQL-tuning-aware query optimizer: SQL tuning is a human-intensive and time-consuming process today, and expensive in terms of the total cost of database ownership. The pain of SQL tuning can be lessened considerably if query optimizers support a feature using which a user or higher-level tuning tool can tell the optimizer: "I am not satisfied with the performance of the plan p being used for the query Q that runs repeatedly. Can you generate a (δ%) better plan?" This Chapter will discuss the design, implementation, and evaluation of *Xplus*, which—to the best of our knowledge—is the first query optimizer to provide this feature.

The key to effective tuning of Q is to make the best use of the information available from running a plan (or subplan) for Q; and to repeat this process until a satisfactory

plan is found. Information available for each operator O of a plan p after p runs includes: (i)*Estimated Cardinality (EC)*, that is, the number of tuples produced by O as estimated by the optimizer during plan selection, (ii)*Actual Cardinality (AC)*, that is, the actual number of tuples produced, and (iii)*Estimated-Actual Cardinality (EAC)*, that is, the number of tuples produced by O as estimated by the optimizer if the optimizer knew the actual cardinality (AC) values of O's children.

Existing approaches: The *Learning Optimizer (Leo)* (Stillger et al., 2001) incorporates AC values obtained from previous plan runs to correct EC values during the optimization of queries submitted by users and applications (Chen and Roussopoulos, 1994). The *pay-as-you-go* approach takes this idea further using proactive plan modification and monitoring techniques to measure approximate cardinalities for subexpressions to supplement the AC values collected from operators in a plan (Chaudhuri et al., 2008). The overall approach of query execution feedback has some limitations in practice:

- *Risk of unpredictable behavior:* Making changes to plans of user-facing queries runs the risk of performance regression because incorporating a few AC values alongside EC values can sometimes lead to the selection of plans with worse performance than before (Markl et al., 2007).

- *Imbalanced use of exploitation and exploration:* Effective tuning of a problem query needs to balance two conflicting objectives. Pure exploitation recommends running the plan with the lowest estimated cost based on the current cardinality estimates. Leo and Pay-As-You-Go take this route which ignores the uncertainty in estimates when picking the plan for the query. In contrast, pure exploration recommends running the plan whose execution will produce information that reduces the uncertainty in current estimates the most, while ignoring plan costs. Oracle's *Automatic Tuning Optimizer (ATO)* (Belknap

et al., 2009) takes an exploration-oriented route where base tables and joins are first sampled to reduce the uncertainty in their cardinality estimates.

- *No support for the SLO setting:* No current optimizer supports the SLO setting where systematic exploration can be critical.

Xplus addresses these limitations. A user or tuning tool can mark a query Q for which the performance of the plan p being picked is not satisfactory; and Xplus will try to find a new plan that gives the desired performance. If Xplus fails to find such a plan, it will still produce Q's optimal plan for the current database configuration and optimizer cost model; with all plans costed using accurate cardinalities. Xplus works with the existing database configuration. While configuration changes (e.g., new indexes, changes to server parameters, or provisioning more resources) can also improve Q's performance, such changes are disruptive in many ways including changes to the performance of other queries. If Xplus fails, then we have the strong guarantee that disruptive changes are unavoidable to get the desired performance. Section 8.4 gives a detailed comparison of Xplus with other SQL tuning approaches.

In addition to the query Q to tune and the current plan p, the user can specify a *stopping condition* for when to stop tuning Q, and *resource constraints* to limit the overhead placed by Xplus on the regular database workload. Xplus accepts a parameter MPL_T (multiprogramming level of tuning) that represents the maximum number of plans Xplus can run concurrently. The current stopping conditions are to run Xplus until (i) a new plan is found that is a given $\delta\%$ better than p; or (ii) a given time interval; or (iii) Xplus can cost all plans for Q using accurate cardinalities (a novel contribution); or (iv) the user exits the tuning session when she is satisfied with the performance of the best plan found so far.

Xplus outputs the best plan p' found so far in the tuning session, and the improvement of p' over p. Database systems provide multiple ways to enable p' to be

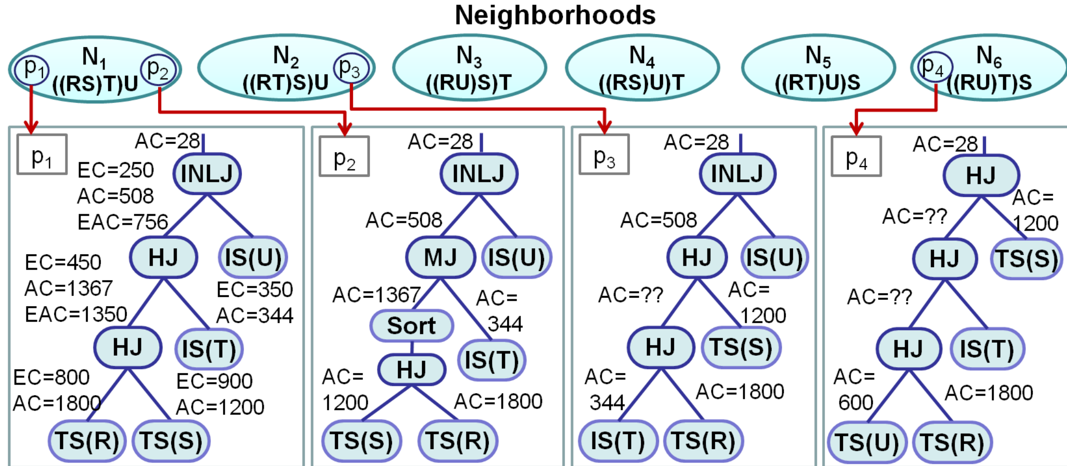


FIGURE 8.1: Neighborhoods and physical plans for our example star-join query.

used for future instances of Q: (i) associating a plan with a query template (*stored outlines* in Oracle (Belknap et al., 2009), *abstract plans* in Sybase ASE (Andrei and Valduriez, 2001), and *Explain_Plan* in PostgreSQL (Herodotou and Babu, 2009)), (ii) *optimizer hints* in Microsoft SQL Server (Bruno et al., 2009) and IBM DB2 (IBM Corp., 2011a), and (iii) *optimization profiles* (IBM Corp., 2009) in DB2.

SQL tuning challenges: We introduce a running example to illustrate the challenges faced during SQL tuning. Suppose Xplus has to tune a star-join query Q that joins four tables R (fact table), S, T, and U, with filter conditions on R and S. Figure 8.1 shows four valid execution plans p_1-p_4 for Q. Let p_1 be the least-cost plan picked by the query optimizer. Figure 8.1 shows the EC, AC, and EAC values for each operator in plan p_1 that are available after a run of p_1.

- *Cardinality estimation errors:* A large difference between an AC and corresponding EC value indicates a cardinality estimation error. For example, AC=1800 differs significantly from EC=800 for the filtered table scan over R, caused possibly by unknown correlations among columns of R.

- *Error propagation:* Blind comparison of AC and EC values can lead to wrong

conclusions because estimation errors propagate up the plan. This issue can be addressed by comparing AC values with corresponding EAC values. For example, consider the hash join between R and S in plan p_1. While the gap between AC=1367 and EC=450 is large for this join, the EAC=1350 is close to AC. (This EAC was computed based on the actual cardinalities, 1800 and 1200, of the join's children.) This information suggests that the estimation error in the join's output cardinality is almost exclusively due to wrong input cardinality estimates, rather than a wrong join selectivity estimate.

- *Livelock in SQL tuning:* Suppose p_2 is the new least-cost plan when the AC values from p_1 are incorporated with existing statistics. Approaches like Leo will use p_2 to run the next instance of Q submitted, which is problematic. Although p_2 is seemingly very different from p_1, p_2 will not bring any *new* AC values. Thus, no further progress will be made towards finding better plans for Q; we say that the tuning of Q is *livelocked*[1].

- *Use of EAC:* Comparing EAC and AC values for joins indicates which joins are more (or less) selective than the optimizer's estimate. For example, R's join with T, whose EAC=756 $>$ AC=508, turned out to be much more selective than estimated. Such information can guide whether a plan like p_3, which has a different join order from p_1, could improve performance. A run of p_3 will also bring the unknown AC value for $R \bowtie T$.

- *Exploitation Vs. exploration:* Would plan p_4 be preferable to p_3 as the plan to run next because a run of p_4 will bring two unknown AC values compared to just one from p_3? At the same time, p_4 is riskier to run because its cost estimate relies on two unknowns. This issue is the manifestation in SQL tuning

[1] Livelocks are well-studied phenomena where a process appears to be executing, but makes no real progress towards its goal.

of the classic "exploit or explore" dilemma in machine learning (Gittins and Jones, 1974).

- *Efficiency features:* Running a subplan instead of the full plan will often bring all the *new* AC values that the plan brings, e.g., the $R \bowtie T$ subplan will bring all new AC values for p_3.

As this example shows, a number of novel opportunities and challenges await a SQL-tuning-aware optimizer like Xplus. The nontrivial challenges Xplus is facing are in (i) choosing a small set of plans to run from the huge plan space, and (ii) guiding the tuning process as new information is collected. To addresses these challenges, Xplus uses a novel abstraction called *plan neighborhoods* to capture useful relationships among plans that simplify information tracking and improve efficiency in SQL tuning. Xplus balances exploitation and exploration efficiently using a combination of tuning *experts* with different goals, and a *policy* to arbitrate among the experts. Efficiency is further improved through features like subplan selection and parallel execution. Finally, Xplus is designed to leverage recent solutions that let the Database system run query plans noninvasively in sandboxed (Belknap et al., 2009) and standby (Belknap et al., 2009; Duan et al., 2009) settings for tuning.

8.1 New Representation of the Physical Plan Space

A query optimizer uses a *performance model* to estimate the cost of plans for a query Q, which consists of a *cardinality model* and a *cost model* (Haas et al., 2009). The cardinality model is used to estimate the cardinality of relational algebra expressions defining the data inputs and output of each operator in a plan. Given these cardinality estimates, the cost model estimates the execution cost of each operator. While modern cost models have been validated to be quite accurate when cardinalities are known, the cardinality models are laced with simplifying assumptions that

can introduce order-of-magnitude errors (Haas et al., 2009).

Definition 1. Cardinality Set: *To estimate the execution cost of an operator O in a plan p, the optimizer uses its cardinality model to estimate cardinalities for a set CS(O) of relational algebra expressions. CS(O) is called O's cardinality set. The cardinality set CS(p) of plan p is the set union of the cardinality sets of all operators in p.*

A function *Generate_CS(O,p)* is written per physical operator type to return the cardinality set *CS(O)* of an instance O of that type in a plan p. The expressions in *CS(O)* are a subset of the relational algebra expressions that define each of O's inputs and outputs in p; and whose cardinalities are needed to find O's execution cost in p. For example, for a hash join operator H over two subplans representing the relational algebra expressions L and R in a plan p, *Generate_CS(H,p)* will return the cardinality set $\{L, R, L \bowtie R\}$. According to PostgreSQL's cost model for a hash join operator, the cardinalities of L, R, and $L \bowtie R$[2] are needed to cost H. Finally, *CS(p)* is generated by invoking the *Generate_CS(O,p)* functions of all operators in plan p using a bottom-up plan traversal.

Definition 2. Plan Neighborhood: *The space of physical plans for a query Q can be represented as a graph G_Q. Each vertex in G_Q denotes a physical plan for Q. An edge exists between the vertices for plans p_1 and p_2 if CS(p_1) $\subseteq$ CS(p_2) or CS(p_2) $\subseteq$ CS(p_1). The connected components of G_Q define the plan neighborhoods of Q.*

CS(N), the cardinality set of plan neighborhood N, is the set union of the cardinality sets of all plans in N, i.e., the cardinalities needed to calculate execution costs for all plans in N. It follows that a plan p belongs to a neighborhood N *iff CS(p)* $\subseteq$ *CS(N)*.

[2] $|L \bowtie R|$ is used for estimating the CPU cost of the hash join.

Neighborhood Table

ID	Cardinality Set	Covered	Least-Cost Plan
N_1	R, S, T, U, σ_p(R), σ_p(S), σ_p(R)⋈σ_p(S), σ_p(R)⋈σ_p(S)⋈T, σ_p(R)⋈σ_p(S)⋈T⋈U	Yes	p_1
N_2	R, S, T, U, σ_p(R), σ_p(S), σ_p(R)⋈T, σ_p(R)⋈σ_p(S)⋈T, σ_p(R)⋈σ_p(S)⋈T⋈U	No	p_3
⋮	⋮	⋮	⋮
N_6	R, S, T, U, σ_p(R), σ_p(S), σ_p(R)⋈U, σ_p(R)⋈U⋈T, σ_p(R)⋈σ_p(S)⋈T⋈U	No	P_4

Cardinality Table

Expression	EC	AC
σ_p(R)	800	1800
σ_p(S)	900	1200
σ_p(R)⋈σ_p(S)	450	1367
σ_p(R)⋈T	650	??
σ_p(R)⋈σ_p(S)⋈T	250	508
σ_p(R)⋈U⋈T	320	??
⋮	⋮	⋮
σ_p(R)⋈σ_p(S)⋈T⋈U	12	28

FIGURE 8.2: Neighborhood and Cardinality Tables for our example star-join query.

Xplus uses two data structures to store all information about neighborhoods: *Neighborhood Table* and *Cardinality Table*. Figure 8.2 illustrates these two data structures for our running example query. The Neighborhood Table stores information about all neighborhoods—including cardinality set and current least-cost plan—with each row corresponding to one neighborhood. Each row in the Cardinality Table stores the EC and (if available) AC values of a relational algebra expression needed for costing. The initialization, use, and maintenance of these data structures during a tuning session are discussed in Section 8.2.

Figure 8.2 shows the cardinality set of neighborhood N_1 for our running example. Plans p_1 and p_2 from Figure 8.1 belong to N_1, with $CS(p_1) \subset CS(N_1)$ and $CS(p_2) \subset CS(N_1)$. We get $CS(p_1) = CS(p_2)$ even though p_2 uses a merge join, has an extra sort, and a different join order between R and S compared to p_1. Consider a plan p_{N_1} (not shown in the figure) that is similar to p_1 except for a hash join (HJ) with table U instead an INLJ. Then, $CS(p_1) \subset CS(p_{N_1})$ because plan p_{N_1} also needs the cardinality of U for costing; and $CS(p_{N_1}) = CS(N_1)$. Plans p_3 and p_4 belong to different neighborhoods than N_1 since $CS(p_3)$ and $CS(p_4)$ are disjoint from $CS(N_1)$.

The progress of Xplus while tuning a query Q can be described in terms of the *coverage* of neighborhoods in Q's plan space.

Definition 3. Covering a Neighborhood: *A neighborhood N is covered when AC values are available for all expressions in CS(N).*

When a tuning session starts, only the AC values from the plan picked originally for Q by the optimizer may be available. More AC values are brought in as Xplus runs (sub)plans during the tuning session, leading to more and more neighborhoods getting covered.

Property 1. *Once a neighborhood N is covered, Xplus can guarantee that all plans in N are costed with accurate cardinalities.* □

Property 2. *Once all neighborhoods are covered for a query Q, Xplus can output Q's optimal plan for the given database configuration and optimizer cost model.* □

The efficiency with which Xplus provides these strong guarantees is captured by Properties 3-5.

Property 3. *Xplus runs* at most *one (possibly modified) plan in a neighborhood N to obtain AC values for CS(N).* □

Property 3 allows Xplus to reduce the number of plans run to tune a query by maximizing the use of information collected from each plan run.

Property 4. *Xplus can cover all neighborhoods for a query Q by running plans for only a fraction of Q's neighborhoods. Almost all these runs are of subplans of Q (and not full plans).* □

Consider our running example query and plan p_1 selected originally by the optimizer (Figure 8.1). Xplus can fully tune this query to provide the strong guarantee in

Property 2 by running only two subplans: one in neighborhood N_3 and one in N_5. As a real-life example, PostgreSQL has around 250,000 valid plans for the complex TPC-H Query Q_9; which gave 36 neighborhoods. Xplus only ran 8 subplans to fully tune this query and give a 2.3x speedup compared to the original PostgreSQL plan.

Property 5. *Xplus can determine efficiently the minimum set of neighborhoods that contain plans whose cost estimates will change based on AC values collected from running a plan for a query.* □

8.2 New Search Strategy over the Physical Plan Space

We will now discuss how Xplus enumerates neighborhoods and plans, chooses the next neighborhood to cover at any point, and selects the (sub)plan to run to cover the chosen neighborhood.

8.2.1 Enumerating Neighborhoods and Plans

Definition 2 lends itself naturally to an approach to enumerate all neighborhoods for a query Q: (i) enumerate all plans for Q and their cardinality sets; (ii) generate the vertices and edges in the corresponding graph G_Q as per Definition 2; and (iii) use Breadth First Search to identify the connected components of G_Q. This approach quickly becomes very expensive for complex queries, so we developed an efficient alternative based on plan *transformations*.

Definition 4. Transformation: *A transformation τ when applied to a plan p for a query Q, gives a different plan p' for Q. τ is an intra (neighborhood) transformation if p and p' are in the same neighborhood; else τ is an inter (neighborhood) transformation.*

Given a sound and complete set of transformations for a given Database system, Xplus can use (i) inter transformations to enumerate all neighborhoods efficiently

for a query starting from an initial plan, and (ii) intra transformations to enumerate all plans in a neighborhood efficiently, starting from a plan stored for each neighborhood. We have developed a set of transformations applicable to the physical plans of PostgreSQL. Transformations are implemented as functions that are applied to a plan data structure representing a physical plan p for a query Q (similar to work on extensible query optimizers, like Graefe and DeWitt (1987)). Applying these transformations multiple times enables Xplus to enumerate all of Q's plans and neighborhoods.

Like many query optimizers (including PostgreSQL), Xplus uses non-cost-based query rewrite rules to identify the *select-project-join-aggregation (SPJA)* blocks in the query. Each SPJA block is optimized separately to produce a plan per block; and these plans are connected together to form the execution plan for the full query. Intra and inter transformations in Xplus are applied to plans for SPJA blocks. Recall that Xplus takes the input query Q and its current (unsatisfactory) plan p as input. Xplus works with the SPJA blocks in p.

Intra Transformations: Intra transformations are applied to a single operator in a plan to generate a different plan in the same neighborhood. These transformations belong to one of two classes:

1. *Method Transformations*, which change one operator implementation (method) to another implementation of the same underlying logical operation. Instances of this class include: (i) Transforming a scan method to a different one (e.g., transforming a full table scan on a table to an index scan); (ii) Transforming a join method to a different one (e.g., transforming a hash join to a merge join); (iii) Transforming a grouping and aggregation method to a different one (e.g., transforming a sort-based aggregation operator to a hash-based one).

2. *Commutativity Transformations*, which swap the order of the outer and inner

subplans of a commutative operator. For example, transforming $L \bowtie R$ to $R \bowtie L$, for subplans L and R, since joins are commutative.

It is important to note that some transformations may not be possible on a given plan. For example, a table scan on table R cannot be transformed into an index scan unless an index exists on R. Also, additional changes may be required along with the application of a transformation in order to preserve plan correctness. For instance, a merge join requires both of its subplans to produce sorted output. If they do not produce sorted output, then a transformation from a hash join to a merge join must add sort operators above the subplans. Note that the addition of these sort operators will not change the cardinality set of the plan (recall plans p_1 and p_2 in Figure 8.1).

Inter Transformations: When applied to a plan p, an inter transformation produces a plan p' in a different neighborhood. That is, $CS(p) \neq CS(p')$. Inter transformations predominantly apply across multiple operators in a plan. (It is theoretically possible that changing the implementation method of an operator in a plan produces a plan in a different neighborhood.) The main inter transformation swaps two tables that do not belong to the same join in a join tree. As with intra transformations, any changes required to preserve the correctness of the new plan are done along with the inter transformation. The two other inter transformations are: (i) Pulling a filter operation F over an operator (by default, F is done at the earliest operator where F's inputs are available in the plan); (ii) Pushing a grouping and aggregation operator below another operator (by default, the grouping and aggregation operator is done after all filters and joins).

Our running example star-join query has six neighborhoods, each with its own distinct join tree (see Figure 8.1). Plan p_2 can be produced from p_1 by applying two intra transformations: a method transformation that changes the middle join from a

hash join to a merge join, and a commutativity transformation that changes $R \bowtie S$ to $S \bowtie R$. Plan p_3 can be produced from p_1 by applying one inter transformation (swap T and S) and one intra transformation (change $R \bowtie T$ to $T \bowtie R$).

8.2.2 Picking the Neighborhoods to Cover

During the tuning process, Xplus must decide which neighborhood to cover next, that is, from which neighborhood to run a plan to collect additional information and convert more cardinality estimates from uncertain to accurate. *Pure exploitation* and *pure exploration* form two extreme decision strategies. In pure exploitation, the plan with the lowest cost based on current cardinality estimates is always preferred, while the uncertainty in these estimates is ignored. In contrast, in pure exploration, the focus is on reducing the uncertainty in cardinality estimates, so a plan that resolves the current uncertainty the most will be preferred over other plans. Exploitation and exploration are naturally at odds with each other, but elements of both are required in holistic SQL tuning.

Xplus balances these conflicting objectives using multiple goal-driven *experts*—given the current global state, an expert has its own assumptions and strategy to recommend the next neighborhood to cover—and a *selection policy* to arbitrate among the experts.

We describe the design of four experts whose combined behavior has worked very well in common query tuning scenarios. These experts operate with different degrees of exploitation and exploration, summarized in Table 8.1.

Pure Exploiter Expert: The Pure Exploiter simply recommends the neighborhood N_{opt} containing the plan with the lowest estimated cost based on the currently available information. N_{opt} is recommended if it has not been covered yet. Otherwise, the Pure Exploiter has no neighborhood to recommend and gets livelocked (i.e., it gets stuck in a local optimum). It is also possible for the Pure Exploiter

Table 8.1: Properties of the current experts in Xplus.

Expert	Exploitation Component	Exploration Component	Can run into a Livelock?
Pure Exploiter	Highest	None	Yes
Join Shuffler	High	Low	Yes
Base Changer	Low	High	No
Pure Explorer	None	Highest	No

to recommend a bad neighborhood by mistake, just like a regular optimizer could select a bad plan, because the plan cost estimation is based on the both AC and EC values. Recall that EC values are computed using statistics supplemented, as is common, with assumptions of uniformity and independence as well as the use of magic numbers. Uncertainty in these estimates is not taken into consideration while computing plan costs.

Join Shuffler Expert: The recommendation strategy of the Join Shuffler is a mix of exploitation and exploration. This expert leans more towards exploitation based on the observation that the more selective joins should appear earlier in the plan's join tree. The Join Shuffler works with the current least-cost plan p_{opt} among all covered neighborhoods. It uses AC and EAC values in p_{opt} to identify joins for which the join selectivity was overestimated, and uses inter transformations to push these joins as low in the plan as possible (with any transformations required to maintain correctness as discussed in Section 8.2.1). If these transformations result in a plan p_{new} in an uncovered neighborhood N_{new}, then N_{new} is recommended; otherwise the Join Shuffler is livelocked.

The degree of overestimation d in the selectivity of a join operator O is computed as the relative difference between O's EAC and AC values. That is, $d = \frac{EAC-AC}{AC}$. Large d for O is an indication that O's join selectivity is much smaller than what the optimizer estimated based on the current available information. Hence, pushing

such joins down the join tree can potentially reduce the data flow up the tree, and decrease the overall cost of the plan.

Base Changer Expert: The Base Changer's recommendation strategy also mixes exploitation and exploration, but this expert leans more towards exploration. The motivation for this expert comes from the observation that the choice of which two tables to join first (called the *base* join) in a join tree often has a high impact on the overall performance of the tree. For example, the base of the plan p_1 in Figure 8.1 is $R \bowtie S$. As anecdotal evidence, optimizer *hints* in most Database systems have direct support to specify the first join or the first table to use in the join tree, e.g., the *leading* hint in Oracle (Belknap et al., 2009).

The Base Changer considers each two-way join in the query as a possible base, and creates the rest of each join tree based on the join order in the current least-cost plan p_{opt}. This strategy causes the Base Changer to recommend neighborhoods with plenty of uncertain cardinality estimates. The mild exploitation component of this expert comes from using p_{opt} to build parts of the join trees. Unless all neighborhoods are covered, the Base Changer will always have a neighborhood to recommend; it will never run into a livelock (unlike the previous two experts).

Pure Explorer Expert: If the overall degree of certainty in cardinality estimates is low, then a lot of information may need to be collected in order to find an execution plan with the desired performance. Compared to the other experts, the Pure Explorer is designed to gather more statistics quickly. For each uncovered neighborhood N_u in the space, the Pure Explorer computes the number of expressions in the cardinality set of N_u whose AC values are not available. The Pure Explorer will then recommend the uncovered neighborhood with the highest number of such expressions. Ties are broken arbitrarily. Similar to the Base Changer, the Pure Explorer will never livelock.

For example, suppose that neighborhood N_1 from our running example in Figure

8.1 has been covered. The Cardinality Table contains AC values for each expression in the cardinality set of N_1 (seen in Figure 8.2). Suppose the Pure Explorer has two options for the neighborhood to recommend: N_2 and N_6. In this case, the Pure Explorer will recommend covering N_6 since it will bring in AC values for two uncertain expressions, namely, $\sigma_p(R) \bowtie U$ and $\sigma_p(R) \bowtie T \bowtie U$; whereas covering N_2 will only bring in the AC value for $\sigma_p(R) \bowtie T$.

Xplus supports three different policies to determine which expert's recommendation should be followed at any point of time.

Round-Robin Policy: This policy consults the experts in a round-robin fashion. Apart from its simplicity, this policy has the advantage of ensuring fairness across all experts.

Priority-based Policy: This policy assigns a predefined priority to each expert. Each time a new recommendation is needed, the experts are consulted in decreasing order of priority. If an expert has no neighborhood to recommend, then the expert with the next highest priority is consulted. By default, priorities are assigned to experts in decreasing order of the degree of exploitation they do. Thus, Pure Exploiter has the highest priority, followed in order by Join Shuffler, Base Changer, and Pure Explorer. Overall, this strategy realizes a common (greedy) approach that humans use when faced with an exploitation versus exploration problem: explore only when further exploitation is not possible currently.

Reward-based Policy: This policy consults experts based on an online assignment of rewards that reflects how well the recommendations from each expert have performed in the past. Each time a new recommendation is needed, the expert with the current highest reward is consulted. If this expert has no neighborhood to recommend, then the expert with the next highest reward is consulted; and so on. Rewards are assigned as follows: If the overall least-cost plan changes based on an expert E's

recommendation, then E's reward is incremented by 1; otherwise, it is reduced by 1.

8.2.3 *Picking the Plan to Run in a Neighborhood*

Once a neighborhood N is selected for coverage, the next step is to pick the least-cost (possibly-modified) plan p_{run} such that a run of p_{run} will bring AC values for all expressions in *MCS(N)*. *MCS(N)*, called the *missing cardinality set* of N, is the subset of expressions in *CS(N)* for which accurate cardinalities are currently unavailable. Xplus uses the following algorithm:

(a) Use intra transformations to enumerate all plans in N.

(b) For each plan $p \in N$, generate plan p' that has any modifications needed to collect all AC values for *MCS(N)*. Find the cost of p'.

(c) Pick p_{run} as the plan p' with least cost over all plans from (b).

Xplus supports two general plan-modification techniques for the nontrivial Step (b): subplan identification and additional scans.

Subplan identification: This technique finds the smallest connected subplan of p, starting from the operators at the leaves of of p, whose execution will bring all AC values for *MCS(N)*.

Additional scans: While most needed AC values correspond to the output cardinality of some operator in p, there are exceptions that need to be handled: (i) an indexed nested-loop join (INLJ) will not collect the inner table's cardinality, and (ii) table scans or index-based access in the presence of filters containing ANDs/ORs of individual predicates may not collect AC values for specific predicate combinations needed for costing (Chaudhuri et al., 2008). Fortunately, both exceptions arise at leaf operators of p. Thus, Xplus addresses them by adding additional table or index scans to p in a cost-based manner.

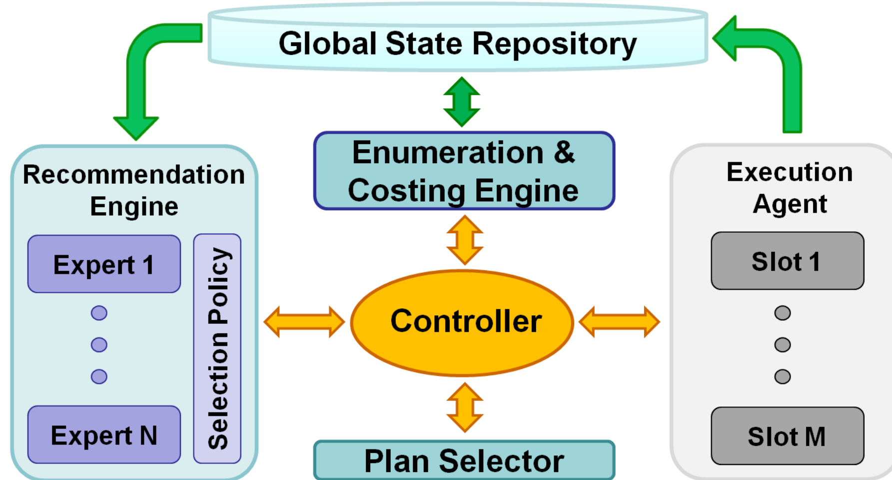


Figure 8.3: System architecture of Xplus.

The above plan-modification techniques were sufficient for PostgreSQL plans. (No changes were made to the PostgreSQL execution engine's source code.) Query plans in systems like IBM DB2 pose other exceptions like early exit from pipelines. Adding (blocking) materialization operators to plans for statistics collection is a plan-modification technique that can handle such exceptions (Markl et al., 2004).

8.3 Implementation of Xplus

8.3.1 Architecture

Xplus consists of six major components as shown in Figure 8.3:

- *Global State Repository*, which stores monitoring information collected by running plans, as well as conventional database statistics from the system catalog.
- *Enumeration and Costing Engine*, which enumerates neighborhoods and plans, and estimates plan costs based on the information in the Repository.
- *Recommendation Engine*, which uses multiple Experts and a Selection Policy to recommend neighborhoods to cover.

- *Plan Selector*, which selects the least-cost (sub)plan to collect the missing AC values in each recommended neighborhood.

- *Execution Agent*, which schedules selected (sub)plans for execution based on the specified resource constraints. Monitoring information from each execution is added to the Repository.

- *Controller*, which shepherds each input query through its lifecycle by invoking the above components appropriately.

Xplus is implemented currently as a Java application that interacts with the Database system through a well-defined interface provided by the system. SQL Database systems contain external or internal interfaces to: (a) get cardinality and cost estimates for physical plans, and (b) run a specified plan and collect AC values for operators during the run. We implemented a new server command in PostgreSQL, called *Explain_Plan*, to expose its interface for costing and running plans to external applications (Herodotou and Babu, 2009). Since the plan neighborhood abstraction, central to how Xplus works, is not present in current optimizers, we developed plan transformations as described in Section 8.2. Overall, only minor changes were needed to PostgreSQL internals to support SQL tuning with Xplus. No changes were made to PostgreSQL's execution engine.

8.3.2 Extensibility Features

Xplus provides three dimensions for extensibility: adding new experts, new selection policies, and new controllers. SQL tuning problems that are hard to fix in a commercial Database system usually get referred to the optimizer developers. Based on the reports seen over time, the developers may notice a defect in the optimizer that causes it to pick poor plans in certain conditions. Rather than modifying the optimizer and thoroughly testing the new version, an easy temporary fix can be to

release a new Xplus expert that spots the mistake pattern and recommends plans to correct it. The expert is dropped once the optimizer is corrected and tested (which is very time consuming). This scenario illustrates one of the many positive impacts that Xplus can have on optimizer development and usage.

Adding a new expert, selection policy, or controller involves implementing specific interfaces defined by Xplus. We used this feature to implement five different controllers, described next. Recall that a controller is responsible for taking a given query through its entire lifecyle (tuning or conventional processing) in the system.

Plan-first-execute-next controller: This non-tuning controller enables Xplus to simulate the conventional query lifecyle: get the least estimated cost plan in the plan space, and run it to generate the query results.

Serial (experts) controller: This controller repeatedly invokes the Xplus components in sequence until the stopping condition is met. The Recommendation Engine picks the next neighborhood N to cover in consultation with the Experts and the Selection Policy. N is given to the Plan Selector for selecting the least-cost plan to run to collect all missing AC values for *CS(N)*. The returned (sub)plan is run by the Execution Agent subject to the resource constraints specified. The new monitoring data is entered into the Repository.

Parallel (experts) controller: This controller runs the Recommendation Engine, Enumeration and Costing Engine, Plan Selector, and Execution Agent concurrently to enable inter-component parallelism in Xplus. The Parallel controller provides multiple benefits:

- If MPL_T is set greater than 1, then these many (sub)plans will be run concurrently.
- If new AC values from a plan execution are available when a costing cycle

completes, another cycle is started to find the new least-cost plan; which helps when the plan space is large.

- Running the Recommendation Engine and Plan Selector in parallel with other components can hide plan recommendation latency in the presence of complex experts or plan modifications.

Leo controller: This controller implements tuning as done by the Leo optimizer (Stillger et al., 2001). In Xplus, the Leo controller effectively means using the serial controller with the Pure Exploiter as the only expert. Whenever the current plan finishes, the Pure Exploiter is consulted for the neighborhood to cover next. The Leo controller cannot make further progress when the Pure Exploiter gets livelocked.

ATO controller: This controller implements how Oracle's ATO (Belknap et al., 2009) performs SQL tuning by collecting cardinality values for per-table filter predicates and two-way joins. After these cardinality values are collected, the new least-cost plan is recommended. Oracle's ATO estimates cardinality values using random sampling. Since PostgreSQL's execution engine has no external support for sampling, the ATO controller collects accurate AC values using the least-cost scan operator for filter predicates, and the least-cost join operator for two-way joins.

We could not implement controllers for other related work like Eddies (Avnur and Hellerstein, 2000), RIO (Babu et al., 2005), Pay-As-You-Go (Chaudhuri et al., 2008), and POP (Markl et al., 2004) because they all require nontrivial changes to the plan execution engine in the database. Such changes to the execution engine can help Xplus collect AC values faster and more efficiently.

8.3.3 Efficiency Features

We have implemented a large range of the features in Xplus that reduce the time to find a better plan as well as make Xplus scale to large queries.

1. *Use of parallelism:* In addition to incorporating inter-component parallelism through the Parallel Controller, Xplus implements intra-component parallelism in multiple places. The Execution Agent can schedule multiple plans to run in parallel based on the MPL_T setting. The Costing Engine, which leverages the plan space partitioning imposed by neighborhoods, can cost plans in parallel. The Recommendation Engine can invoke different experts in parallel.

2. *Executing subplans instead of full plans:* The Plan Selector implements this optimization as described in Section 8.2.3, giving major efficiency improvements as we will see in Section 8.5.

3. *Prioritize neighborhoods:* It is possible that some neighborhoods consist almost exclusively of highly suboptimal plans. Xplus proactively identifies and avoids such neighborhoods, even when not all AC values are available for them. As an example, consider plan p_4 belonging to neighborhood N_6 in Figure 8.1. Let AC values be available for $\sigma_p(R)$, U, and $\sigma_p(R) \bowtie U$, but not for the rest of the plan. Also suppose that the three available AC values are very high. Then, irrespective of what values the unknown cardinalities take, Xplus may discover that the cost of doing $\sigma_p(R) \bowtie U$ makes all plans in neighborhood N_6 worse than the overall best plan among the neighborhoods covered so far. If so, Xplus can avoid N_6.

4. *Optimized preprocessing:* Recall how the Plan Selector may need to add additional scans to plans to collect needed AC values. Xplus proactively identifies such exceptions during neighborhood enumeration, and does cost-based table or index scans per table to collect needed AC values in a preprocessing step. Table-by-table preprocessing is efficient because it makes better use of the buffer cache.

5. *Use of materialization:* Plans from different neighborhoods may share common subexpressions (e.g., $R \bowtie S$ for neighborhoods N_1 and N_4 in Figure 8.1). Finding commonalities and creating materialized views help avoid re-computation during SQL tuning.

6. *Use of sampling:* Database systems have made giant strides in internal and external support for sampling (Olken and Rotem, 1995). Xplus could use sampling instead of executing (sub)plans on the full data.

7. *Execution engine modifications:* Xplus (especially the Plan Selector) can benefit from a number of plan-modification techniques proposed in the database research literature to increase statistics and cost monitoring capabilities during plan execution (Babu et al., 2005; Chaudhuri et al., 2008; Markl et al., 2004).

The first four features listed above are fully integrated into Xplus, with the first two been the most useful. The last three features are left for future work.

8.4 Comparing Xplus to Other SQL-tuning Approaches

In this section, we discuss current approaches to SQL tuning and provide a detailed comparison with Xplus. Table 8.2 provides a high-level summary of the similarities and differences between Xplus and other approaches with respect to various important properties.

Using feedback from query execution: Leo (Stillger et al., 2001) corrects cardinality estimation errors made by the query optimizer by comparing EC values with AC values obtained when plans run (Chen and Roussopoulos, 1994). This approach can find a better plan for a poorly-performing query Q over multiple executions of Q or of queries with similar subexpressions. The Pay-As-You-Go approach took this idea further using proactive plan modification and monitoring techniques to measure

Table 8.2: Comparison of Xplus, Leo, Pay-As-You-Go, and ATO.

Property	Xplus	Leo	Pay-As-You-Go	ATO
Balanced use of exploitation and exploration	Yes	No	No	No
Support for SLO tuning tasks	Yes	No	No	No
Risk of unpredictable changes to user-facing query behavior	No	Yes	Yes	No
Requires changes to the query execution engine	No	No	Yes	No
Provides optimality guarantees for given database configuration and optimizer cost model	Yes (Prop. 2, Sec. 8.1)	No	No	No
Use of parallelism	Yes	No	No	Possible
Use of collected statistics to improve plans for other queries	Possible	Yes	Yes	Possible
Potential to address errors in the optimizer cost model	Yes	No	No	Yes
Possibility for running into a livelock in the SQL tuning process (Section 8.2.2)	Depends on choice of experts	Yes	Yes	No
Use in fully automated tuning	Possible	Yes	Yes	Yes

approximate cardinality values for subexpressions, in addition to the subexpressions contained in a running plan (Chaudhuri et al., 2008). While query execution feedback is related closely to how Xplus performs SQL tuning, there are some key differences between Xplus and the existing query execution feedback approaches:

- SQL tuning is inherently an unpredictable and risky process in that a plan better than the optimizer's original pick p may be found only after some plans worse than p are tried. Given how difficult query optimization is, there is always an element of trial-and-error in SQL tuning. Furthermore, experiences with Leo show that incorporating some AC values alongside EC values can cause optimizers to pick plans whose performance is worse than before (Markl et al., 2007). Thus, attempting SQL tuning directly on queries being run by users runs the risk of unpredictable behavior and performance regression. DBAs and

users usually prefer predictable, possibly lower, performance over unpredictable behavior (Babcock and Chaudhuri, 2005). For this reason, unlike Leo and Pay-As-You-Go, Xplus deliberately keeps the SQL tuning path separate from the normal path of submitted queries.

- The concept of balancing the exploitation and exploration objectives explicitly in SQL tuning is unique to Xplus. Leo and Pay-As-You-Go are exploitation-heavy approaches, ignoring the uncertainty in estimates when picking the plan for the query.

- A serious concern with using an exploitation-heavy approach is the possibility of a livelock (see Section 8.2.2) because the subexpressions produced by a plan dictate which set of actual cardinality values are available from running that plan.

- Unlike Leo and Xplus, implementing the Pay-As-You-Go approach (and related ones like RIO (Babu et al., 2005) and POP (Markl et al., 2004)) in a Database system requires nontrivial changes to the plan execution engine.

- Execution of plans brings in valuable information like EC, AC, and EAC cardinality values. Other types of information that Xplus can track based on plan execution include estimated and actual costs (including *estimated-actual costs* similar to EAC), I/O patterns, and resource bottlenecks. Significant differences between actual costs and the corresponding estimated-actual costs may indicate errors in the optimizer cost model.

Oracle's Automatic Tuning Optimizer (ATO): Xplus shares common goals with Oracle's ATO (Belknap et al., 2009), but differs in the algorithms and system architecture. While Leo and Pay-As-You-Go focus on exploitation, ATO is on the

exploration side. When invoked for a query Q, ATO does additional processing to reduce the uncertainty in cardinality estimates for Q. First, ATO collects random samples from the base tables to validate the cardinality estimates of Q's filter predicates. Given more time, ATO performs random sampling to validate all two-way join cardinality estimates (possibly, up to all n-way joins). ATO uses a sandboxed environment for the additional processing to limit the overhead of the tuning process on the rest of the database workload. If the new estimates cause the least-cost plan to change, then ATO compares the performance of the new plan against the old one by running both plans in a competitive fashion. Unlike Xplus, ATO has features to recommend new statistics to collect, indexes to create, and rewrites to the query text.

Adaptive query processing: Xplus is one point in the design spectrum that includes a long line of work on adaptive query processing (Deshpande et al., 2007). The Rdb system introduced *competition-based* query plan selection, namely, running multiple plans for the same query concurrently, and retaining the best one (Antoshenkov and Ziauddin, 1996). Database systems for emerging application domains (e.g., MongoDB) are using this concept to address the lack of statistics in these settings.

Eddies (Avnur and Hellerstein, 2000) identified the relationship of adaptive query processing to *multi-armed bandit (MAB)* problems from machine learning (Gittins and Jones, 1974). The study of MAB problems has led to theory and algorithms to balance the exploitation-exploration tradeoff under certain assumptions, including algorithms to control multiple competing experts (Gittins and Jones, 1974). Xplus uses a similar approach by designing experts for SQL tuning who recommend new plans to try based on the information available so far. While the experts in Xplus are static, small in number, and recommend query plans, each expert in Eddies and

Rdb is a candidate plan for the query; which makes the architecture and algorithms of Xplus very different from that of Eddies and Rdb. Experts have also been used in query optimizers to exercise rewrite rules and heuristics during the optimization of a query (Kemper et al., 1993).

Multiple optimization levels: Most current optimizers provide multiple levels of optimization. For example, when a query is optimized in IBM DB2's most powerful 9th level, all available statistics, query rewrite rules, access methods, and join orders are considered (IBM Corp., 2011b). The design of *X*plus (which stands for 10+) came from considering what hypothetical classes 10 and higher of DB2's optimizer could usefully do. Our answer is that these higher levels will execute selected (sub)plans proactively, and iterate based on the observed information; a challenging task left to DBAs today.

8.5 Experimental Evaluation

The purpose of the evaluation of Xplus is threefold. First, we evaluate the effectiveness and efficiency of Xplus in tuning poorly-performing queries. Second, we compare Xplus with previous work. Finally, we evaluate different expert-selection policies, combinations of experts, and the impact of the efficiency features outlined in Section 8.3.3. All experiments were run on an Ubuntu Linux 9.04 machine, with an Intel Core Duo 3.16GHz CPU, 8GB of RAM, and an 80GB 7200 RPM SATA-300 hard drive. The database server used was PostgreSQL 8.3.4. We used the TPC-H Benchmark (TPC, 2009) with a scale factor of 10. We used the DB2 index advisor (db2advis) to recommend indexes for the TPC-H workload since we have observed that its index recommendations work well for PostgreSQL. All table and column statistics are up to date except when creating problem queries due to stale statistics. Unless otherwise noted, all results were obtained using the Parallel Experts

Table 8.3: Tuning scenarios created with TPC-H queries.

	TPC-H Queries								
Tuning Scenario Class	2	5	7	9	10	11	16	20	21
Query-level issues	✓		✓	✓	✓			✓	
Data-level issues	✓	✓	✓			✓			✓
Statistics-level issues		✓			✓		✓		✓
Physical-design issues			✓	✓			✓	✓	

Controller with MPL_T=2 and the Priority policy with all experts.

We will present the evaluation of Xplus in terms of *tuning scenarios*, where a query performs poorly due to some root cause. Four classes of query tuning scenarios are common in practice:

1. *Query-level issues*: A query may contain a complex predicate (e.g., with a UDF) for which cardinality estimation is hard.

2. *Data-level issues*: Real-life datasets contain skew and correlations that are hard to capture using common database statistics.

3. *Statistics-level issues*: Statistics may be stale or missing, e.g., when a lot of new data is loaded into a warehouse per day.

4. *Physical-design issues*: The optimizer may not pick a useful index, or it may pick an index that causes a lot of random I/O.

We created multiple instances per tuning scenario class. Query-level issues were created by making minor changes to the TPC-H query templates, mainly in the form of adding filter predicates. Data-level issues were created by injecting Zipfian skew into some columns. We decreased the amount of statistics collected by PostgreSQL for some columns to create issues at the statistics and physical design levels. Table 8.3 summarizes the issues that were caused for each TPC-H query. Often problems are due to some combination of multiple root causes, which is reflected in Table 8.3.

Table 8.4: Overall tuning results of Xplus for TPC-H queries.

Query	Run Time of PostgreSQL Plan (sec)	Run Time of Xplus Plan (sec)	Speedup Factor	Number of Subplans Xplus Ran	Time to Find Xplus Plan	
					Absolute (sec)	Normalized (Col6/Col2)
2	8.67	0.59	14.8	5	40.42	4.66
5	1037.80	399.01	2.6	8	149.76	0.14
7	257.55	21.38	12.0	6	131.58	0.51
9	1722.27	754.82	2.3	8	870.78	0.51
10	2248.52	695.70	3.2	4	149.15	0.07
11	20.00	3.55	5.6	2	29.11	1.46
16	15.90	0.77	20.7	2	27.04	1.70
20	3.36	2.32	1.4	4	7.13	2.13
21	509.51	72.17	7.1	4	45.83	0.09

8.5.1 Overall Performance of Xplus

The first set of experiments targets the ability and efficiency of Xplus in finding better execution plans for poorly-performing queries. Table 8.4 provides the results for nine different tuning scenarios. (All plan running times shown are averaged over six runs.) In all nine cases, Xplus found a better execution plan, offering an average speedup of 7.7 times faster compared to the original plan (selected by the PostgreSQL query optimizer) to be tuned. In three cases, Xplus found a new plan that is over an order of magnitude faster.

The last two columns of Table 8.4 show the time Xplus takes to find the better plan. The absolute times (second-last column) are small, which shows the high degree of efficiency that our implementation achieves. In particular, the last column shows *normalized tuning time*, which is the ratio of the time taken by Xplus to find the better plan to the running time of the original plan to be tuned. The low values in this column clearly demonstrate how Xplus gives its benefits in the time it takes to run the original plan a very small number of times (often < 2).

Figure 8.4 shows the execution timeline of Xplus while tuning TPC-H Query 7. The y-axis shows the execution time of the best plan found so far in the covered

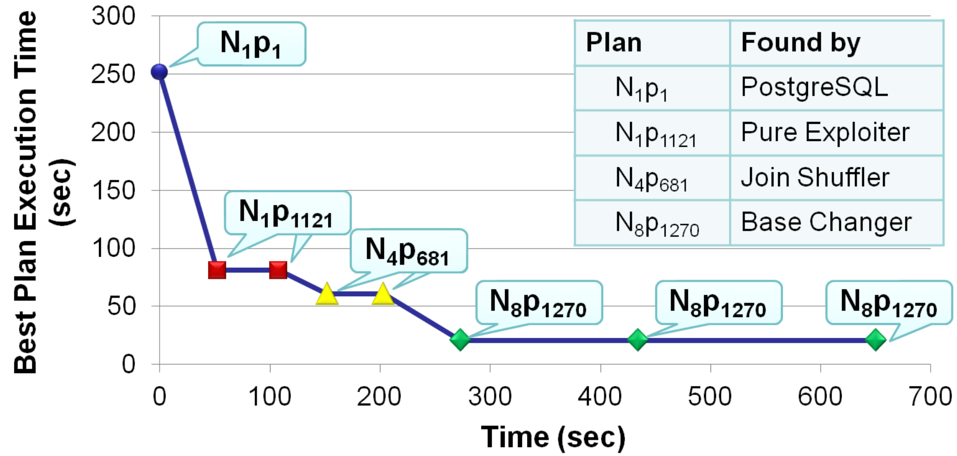


FIGURE 8.4: Progress of the execution time of the best plan in the covered space as Xplus tunes TPC-H Query 7.

space. The plan found by the PostgreSQL optimizer is N_1p_1 (plan p_1 in neighborhood N_1), which runs in 257.55 seconds. Let us see the role of the experts in Xplus in this tuning task. For ease of plotting the timeline, the Serial Controller with the Priority policy and all four experts was used. First, a neighborhood recommended by the Pure Exploiter led to the discovery of plan N_1p_{1121}, which gave a speedup factor of 3.1. The Pure Exploiter livelocked at this point. The Join Shuffler then recommended a neighborhood that led to plan N_4p_{681}; increasing the speedup to 4.1. It took a recommendation from the exploration-heavy Base Changer for Xplus to find plan N_8p_{1270} with a speedup of 12. All neighborhoods were covered by the execution of 7 subplans (not full plans). Recall the strong guarantee that Xplus provides once all neighborhoods are covered (Property 2). This tuning task is an excellent example of how exploitation and exploration are both needed to reach the optimal plan.

8.5.2 Comparison with Other SQL-tuning Approaches

We now compare Xplus with two other SQL tuning approaches: Leo and Oracle's ATO using the respective controllers discussed in Section 8.3.2. For the same nine tuning scenarios from Table 8.4, Figure 8.5 shows the speedup factor of the plans

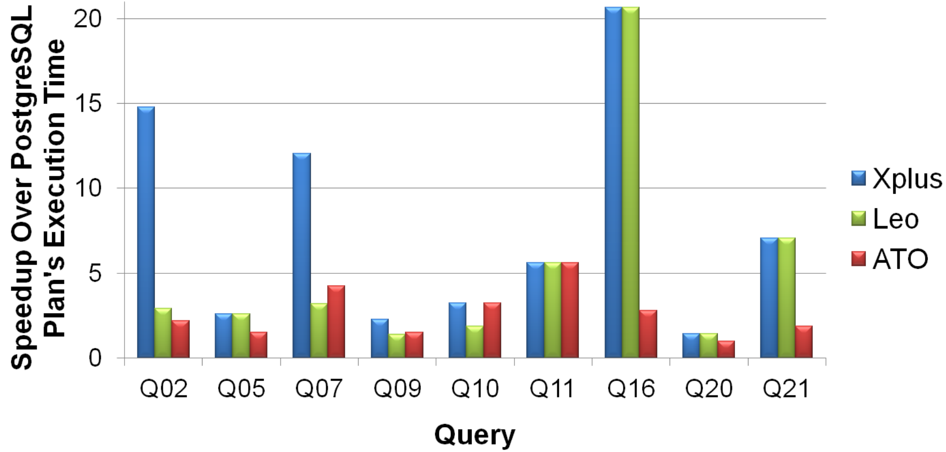


FIGURE 8.5: Speedup from Xplus, Leo, and ATO controllers over the execution time of the original PostgreSQL plan.

produced by the three approaches compared to the original plan to be tuned. Xplus found a better plan than Leo in 4 cases, offering up to an order of magnitude additional speedup. Xplus found a better plan than ATO in 7 cases, with similar improvements. The performance advantages of Xplus are more prominent for more complex queries.

SQL tuning is also needed while tuning multi-tier services to meet service-level objectives (SLOs) on response time or workload completion time. Motivated by this scenario, Table 8.5 shows the performance of Xplus, Leo, and ATO controllers for the following tuning task per query Q: *find a plan that is 5x faster than the current plan picked by the PostgreSQL optimizer for* Q. For each approach, we show its normalized tuning time and *result*. For the Leo and ATO controllers, the result is one of: (i) *Success*, if a plan with $\geqslant$5 speedup is found; or (ii) *Failure*(α), if the controller could only find a plan with α <5 speedup. In contrast, when Xplus fails to find a plan with $\geqslant$5 speedup, it provides the guarantee *Guarantee*(α): for the given database configuration and optimizer cost model, the optimal plan for Q only gives α speedup. With this knowledge, the user or tuning tool can plan for the disruptive

Table 8.5: Tuning results of Xplus, Leo controller, and ATO controller when asked to find a 5x better plan. Time is normalized over the execution time of the original PostgreSQL plan.

Query	Xplus		Leo Controller		ATO Controller	
	Time	Result	Time	Result	Time	Result
2	4.66	Success	5.09	Failure(2.9)	4.99	Failure(2.2)
5	2.56	Guarantee(2.6)	0.57	Failure(2.6)	1.92	Failure(1.5)
7	0.51	Success	0.26	Failure(3.2)	1.03	Failure(4.2)
9	2.91	Guarantee(2.3)	0.91	Failure(1.4)	2.13	Failure(1.5)
10	0.07	Guarantee(3.2)	0.03	Failure(1.9)	0.23	Failure(3.2)
11	1.46	Success	0.14	Success	0.54	Success
16	1.70	Success	0.10	Success	1.35	Failure(2.8)
20	2.23	Guarantee(1.4)	2.12	Failure(1.4)	3.07	Failure(1.0)
21	0.09	Success	0.01	Success	0.59	Failure(1.9)

changes needed to the physical design, server parameters, or resource provisioning to get the desired performance for the query.

Table 8.5 demonstrates the advantages of Xplus. Xplus finds a 5x faster plan in five cases in Table 8.5, and provides a strong guarantee in the rest. The Leo and ATO controllers succeed in only three cases and one case respectively. The Leo controller fails to complete a task because it runs into a livelock, whereas the ATO controller fails because the cardinality estimates gathered from sampling tables and two-way joins are not enough to produce a plan with the desired performance.

8.5.3 Internal Comparisons for Xplus

Figures 8.6(a) and 8.6(b) illustrate an important trend that emerged in our evaluation. These figures consider five strategies for plan recommendation: Priority, Round Robin, and Rewards, each with all four experts; and Priority with (a) the Pure Exploiter and Pure Explorer (called *Exploiter-Explorer*), and (b) the Pure Explorer only (called *Explorer-Only*). These strategies are compared based on *convergence time* (how soon they found the best plan), as well as the *completion time* (how long they took to cover all neighborhoods).

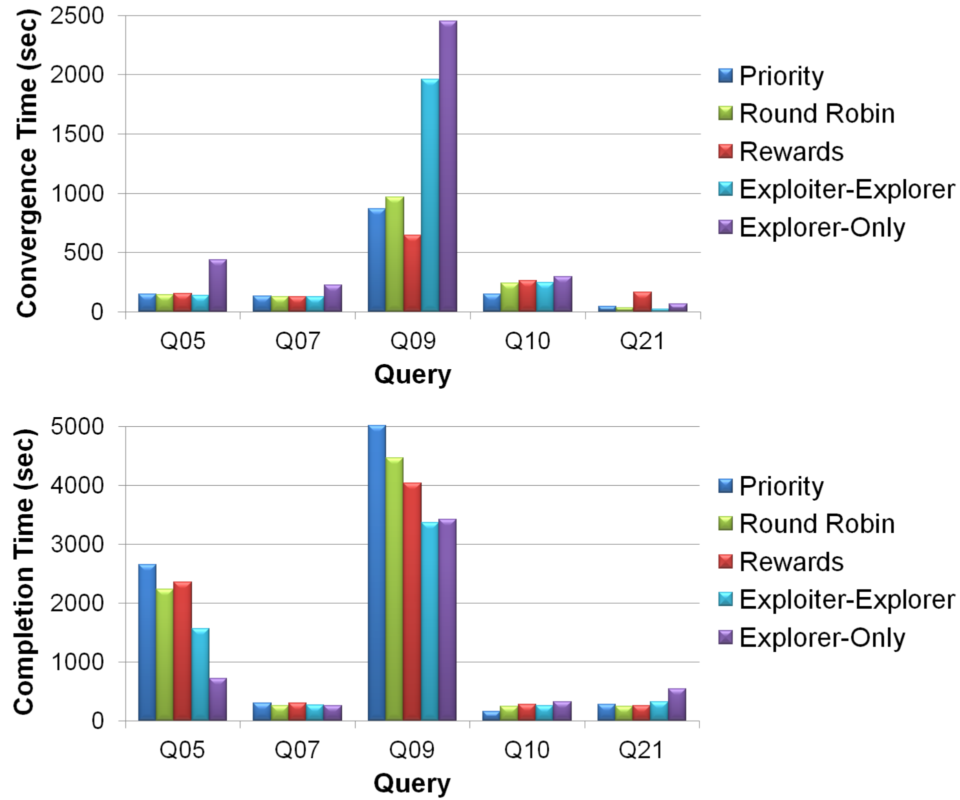


FIGURE 8.6: (a) Convergence times and (b) completion times for the expert-selection policies.

Note from Figure 8.6 that exploration-heavy policies (like Explorer-Only and Exploiter-Explorer) take longer to converge, but lead to lower completion times. Exploitation targets missing statistics related to the current least-cost plan, which leads to better convergence. However, the time to gather all statistics is longer as exploitation makes small steps towards this goal. Exploration brings in more information in each step, often decreasing the total number of executed (sub)plans and the overall completion time.

Based on these observations, we offer the following guideline to choose the policy and experts for a SQL tuning task. If the user or DBA wishes to find the best plan possible (e.g., to decide whether disruptive tuning can be avoided), then she should select an exploration-heavy strategy. On the other hand, if she is interested

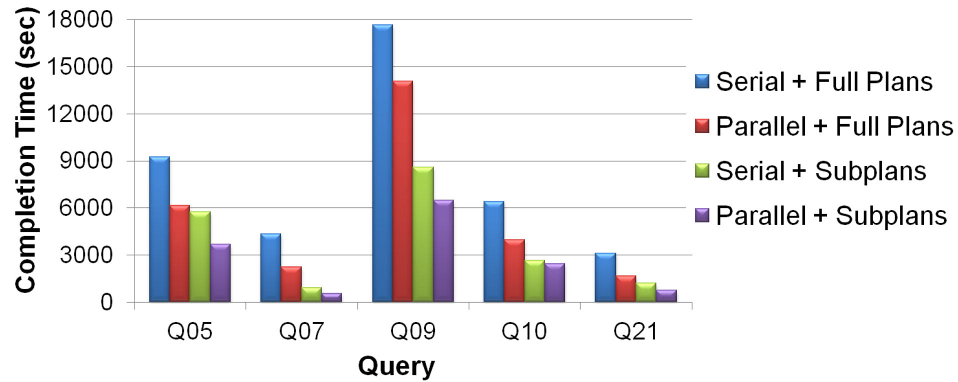


FIGURE 8.7: Impact of the efficiency features in Xplus.

in quick improvements to the current plan, then a strategy that favors exploitation over exploration is more suitable.

Figure 8.7 shows the impact of the two important efficiency features of Xplus: use of parallelism and running subplans instead of full plans whenever possible. Use of subplans is particularly beneficial for complex and long-running queries. For example, Xplus ran 8 subplans to cover all neighborhoods for TPC-H Query 5. Most of these subplans contained only around half of the tables in a full plan for the query, causing Xplus to complete four times faster.

9

Increasing Partition-awareness in Cost-based Query Optimization

"M" in MADDER principles represents the need to support data analysis over a variety of data sources. At the same time, data partitioning—and table partitioning in particular—is a powerful mechanism for improving query performance and system manageability in both Database systems (IBM Corp., 2007; Morales, 2007; Talmage, 2009) and Dataflow systems (Thusoo et al., 2009; Chaiken et al., 2008). Uses of partitioning range from more efficient loading and removal of data on a partition-by-partition basis to finer control over the choice of physical design, statistics creation, and storage provisioning based on the workload. Table 9.1 lists various uses of table partitioning. Deciding how to partition tables, however, is now an involved process where multiple objectives—e.g., getting fast data loading along with good query performance—and constraints—e.g., on the maximum size or number of partitions per table—may need to be met.

Unfortunately, cost-based query optimization technology has not kept pace with the growing usage and user control of table partitioning. Previously, query optimizers

Table 9.1: Uses of table partitioning in Database systems.

Uses of Table Partitioning in Database Systems
Efficient pruning of unneeded data during query processing
Parallel access to data during query processing (e.g., parallel scans and partitioned parallelism)
Reducing data contention during query processing and administrative tasks. Faster data loading, archival, and backup
Efficient statistics maintenance in response to insert, delete, and update rates. Better cardinality estimation for subplans that access few partitions
Prioritized data storage on faster/slower disks based on access patterns
Fine-grained control over physical design for database tuning
Efficient and online table and index defragmentation at the partition level

had to consider only the restricted partitioning schemes specified by the database administrator (DBA) on base tables. Today, the query optimizer faces a diverse mix of partitioning schemes that expand on traditional schemes such as hash and equi-range partitioning.

The growing usage of table partitioning has been accompanied by efforts to give applications and user queries the ability to specify partitioning conditions for tables that they derive from base data. MapReduce frameworks enable users to provide partitioning functions that dictate how the data output by the map tasks is partitioned across the reduce tasks. This feature is used extensively by the Hive and Pig frameworks (Thusoo et al., 2009; Olston et al., 2008b). Hive's SQL-like query language offers direct support to specify table partitioning. Finally, we are also starting to see SQL extensions that provide first-class support for partitioning (Friedman et al., 2009). Given such features, DBAs may not be able to control or restrict how tables accessed in a query are partitioned.

In this chapter, we present new cost-based techniques to generate efficient plans for queries involving multiway joins over partitioned tables. We will focus on horizontal partitioning in centralized row-store Database systems sold by major database

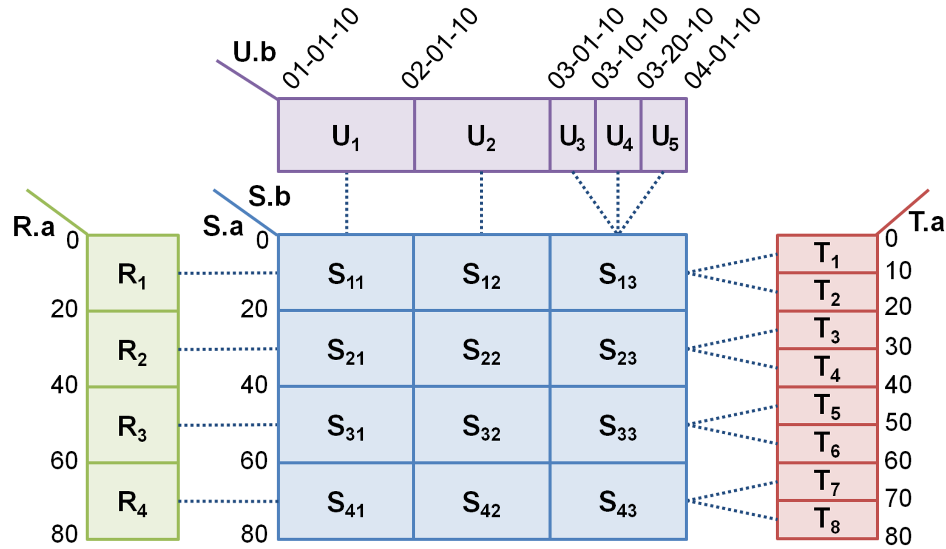


FIGURE 9.1: Partitioning of tables R, S, T, and U. Dotted lines show partitions with potentially joining records.

vendors as well as popular open-source systems like MySQL and PostgreSQL. The need for partitioning in these systems has been shown previously (Agrawal et al., 2004; Zeller and Kemper, 2002). Our work is also useful in parallel Database systems like Aster nCluster (AsterData, 2012; Friedman et al., 2009), HadoopDB (Abouzeid et al., 2009), and Teradata (Teradata, 2012) since these systems try to partition data such that most queries in the workload need intra-node processing only. The contributions of this chapter are complementary to the overall tuning approach proposed in this dissertation, and necessary for both Database and Dataflow systems to support the MADDER principles.

9.1 Optimization Opportunities for Partitioned Tables

We will begin with an illustration of the diverse mix of partitioning schemes that expand on traditional schemes such as hash and equi-range partitioning. Figure 9.1 shows partitioning schemes for tables $R(a)$, $S(a, b)$, $T(a)$, and $U(b)$, where attribute a is an integer and b is a date. Table S exhibits *hierarchical* (or *multi-dimensional*)

partitioning; S is equi-partitioned on ranges of a into four partitions S_1-S_4, each of which is further partitioned on ranges of b. Such scheme can deal with multiple granularities or hierarchies in the data (Baldwin et al., 2003).

Tables R, S, and T are all partitioned on a—typical for multiple related data sources or even star/snowflake schemas—but with different ranges due to data properties and storage considerations. For example, if the number of records with the same value of a is large in T (e.g., user clicks), then smaller ranges will give more manageable partitions.

Table U is partitioned using nonequi ranges on b for data loading and archival efficiency as well as workload performance. Daily partitions for daily loads are an attractive option since it is faster to load an entire partition at a time. However, maintenance overheads and database limitations on the maximum number of partitions can prevent the creation of daily partitions. Hence, 10-day ranges are used for recent partitions of U. Older data is accessed less frequently, so older 10-day partitions are merged into monthly ones to improve query performance and archival efficiency.

The flexible nature and rising complexity of partitioning schemes pose new challenges and opportunities during the optimization of queries over partitioned tables. Consider an example query Q over the partitioned tables R, S, and T in Figure 9.1.

```
Q:  Select  *
    From    R, S, T
    Where   R.a = S.a and S.a = T.a and S.b ⩾ 02-15-10 and T.a < 25;
```

Use of filter conditions for partition pruning: An optimization that many current optimizers apply to Q is to *prune* partitions T_4-T_8 and S_{11}, S_{21}, S_{31}, S_{41} from consideration because it is clear from the partitioning conditions that records in these partitions will not satisfy the filter conditions. Partition pruning can speed

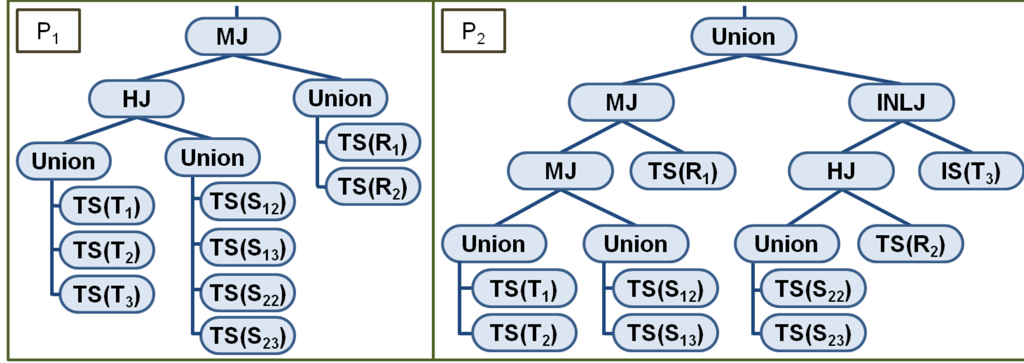


FIGURE 9.2: P_1 is a plan generated by current optimizers for the running example query Q. P_2 is a plan generated by our partition-aware optimizer. IS and TS are respectively index and table scan operators. HJ, MJ, and INLJ are respectively hash, merge, and index-nested-loop join operators. Union is a bag union operator.

up query performance drastically by eliminating unnecessary table and index scans as well as reducing memory needs, disk spills, and contention-related overheads.

Use of join conditions for partition pruning: Based on a transitive closure of the filter and join conditions, partition pruning can also eliminate partitions S_{32}, S_{33}, S_{42}, S_{43}, R_3, R_4, and U_1.

Most current optimizers will stop here as far as exploiting partitions during the optimization of Q is concerned; and generate a plan like P_1 shown in Figure 9.2. In a plan like P_1, the leaf operators logically append together (i.e., do a bag union of) the unpruned partitions for each table. Each unpruned partition is accessed using regular table or index scans. The appended partitions are joined using operators like hash, merge, and (index) nested-loop joins.

Partition-aware join path selection: Depending on the data properties, physical design, and storage characteristics of the Database system, a plan like P_2 shown in Figure 9.2 can significantly outperform plan P_1. P_2 exploits a number of properties arising from partitioning in the given setting:

- Records in partition R_1 can join only with $S_{12} \cup S_{13}$ and $T_1 \cup T_2$. Similarly,

records in partition R_2 can join only with $S_{22} \cup S_{23}$ and T_3. Thus, the full $R \bowtie S \bowtie T$ join can be broken up into smaller and more efficient *partition-wise joins*.

- The best join order for $R_1 \bowtie (S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ can be different from that for $R_2 \bowtie (S_{22} \cup S_{23}) \bowtie T_3$. One likely reason is change in the data properties of tables S and T over time, causing variations in statistics across partitions [1].
- The best choice of join operators for $R_1 \bowtie (S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ may differ from that for $R_2 \bowtie (S_{22} \cup S_{23}) \bowtie T_3$, e.g., due to storage or physical design differences across partitions (e.g., index created on one partition but not on another).

The above examples illustrate the optimization possibilities for SQL queries over partitioned tables, which enlarge the plan space drastically. To our knowledge, no current optimizer (commercial or research prototype) takes this space into account to find efficient plans with low optimization overhead. We address this limitation by developing a novel *partition-aware SQL query optimizer*:

- **Dealing with plan space explosion:** A nontrivial challenge we have to address in a partition-aware optimizer is to keep the additional computational and memory overheads of the optimization process in check while enabling good plans to be found.
- **Incorporation into state-of-the-art optimizers:** The new techniques we propose are designed for easy incorporation into bottom-up query optimizers (like the seminal System R optimizer (Selinger et al., 1979)) that are in wide use today. With this design, we leverage decades of past investment as well as

[1] Most enterprises keep 6-24 months of historical data online.

potential future enhancements to these optimizers (e.g., new rewrite rules, new join operators, and improvements in statistics and cardinality estimation).

- **Partitions as physical or logical properties?** The conventional wisdom in the database literature as well as implementation in commercial bottom-up query optimizers treat partitions as physical properties (Rao et al., 2002). We show that treating partitions only as physical properties falls well short of making the best use of partitioned tables. Our optimizer considers partitions efficiently at both the logical and physical levels to get the best of two worlds: (a) generating plans like P_2 in Figure 9.2, and (b) preserving *interesting partitions* (Rao et al., 2002) that may benefit operators (e.g., group-by) higher-up in the plan.

- **Supporting practical partitioning conditions:** In addition to conventional DBA-specified partitioning conditions on base tables, our optimizer supports a wide range of user-specified partitioning conditions including multi-dimensional partitions, multi-level hierarchical partitions, and irregular ranges. The challenge here is to deal with complex join graphs arising at the partition level (like Figure 9.1) from the combination of the filter, join, and table-level partitioning conditions for a SQL query.

- **Improving cardinality estimates:** A nonobvious effect arises from the fact that most Database systems keep statistics (e.g., number of distinct values) at the level of individual partitions. Cardinality estimation for appended partitions necessitates combination of per-partition statistics. We have found that estimation errors from such combination are worse for a plan like P_1 that unions multiple partitions together compared to P_2.

9.2 Related Work on Table Partitioning

Various table partitioning schemes as well as techniques to find a good partitioning scheme automatically have been proposed as part of database physical design tuning (e.g., Agrawal et al. (2004); Rao et al. (2002)). In contrast, our goal is to fully exploit possible query optimization opportunities given the existing horizontal partitioning scheme in the database.

Partitioning in centralized Database systems: Commercial Database Management System (DBMS) vendors (e.g., IBM, Microsoft, Oracle, and Sybase) provide support for different types of partitioning, including hash, range, and list partitioning, as well as support for hierarchical (multi-dimensional) partitioning. However, they implement different partition-aware optimization techniques. Most commercial optimizers have excellent support for per-table partition pruning. In addition to optimization-time pruning, systems like IBM DB2 support pruning of partitions at plan execution time, e.g., to account for join predicates in index-nested-loop joins (IBM Corp., 2007). Some optimizers generate plans containing n *one-to-one partition-wise joins* for any pair of tables R and S that are partitioned into the same number n of partitions with one-to-one correspondence between the partitions (Morales, 2007; Talmage, 2009). For joins where only table R is partitioned, Oracle supports dynamic partitioning of S based on R's partitioning; effectively creating a one-to-one join between the partitions.

UNION ALL views are a useful construct that can be used to support table partitioning (Neugebauer et al., 2002). The techniques proposed in this paper are related closely to pushing joins down through UNION ALL views. For example, when a UNION ALL view representing a partitioned table $R = R_1 \cup \ldots \cup R_n$ is joined with a table S, IBM DB2's query optimizer considers pushing the join down to generate a UNION of base-table joins $(R_1 \bowtie S) \cup \ldots \cup (R_n \bowtie S)$ (Neugebauer

et al., 2002). However, unlike our techniques, the join pushdown is considered in the query-rewrite phase. As the authors of (Neugebauer et al., 2002) point out, this step can increase the time and memory overheads of optimization significantly because of the large number of joins generated (especially, if multiple UNION ALL views are joined like in our example query in Figure 9.2). The techniques we propose are designed to keep these overheads in check—even in the presence of hundreds of partitions—while ensuring that good plans can be found.

Partitioning in parallel/distributed Database systems: While we focus on centralized DBMSs, the partition-aware optimization techniques we propose are related closely to *data localization* in distributed DBMSs (Ozsu and Valduriez, 1999). Data localization is a query-rewrite phase where heuristic rules like filter pushdown are used to prune partitions and their joins that will not contribute to the query result. A join graph is created for the partitions belonging to the joining tables, and inference rules are used to determine the empty joins (Ceri and Gottlob, 1986). While our work shares some goals with data localization, a number of differences exist. Instead of heuristic rewrite rules, we propose (provably optimal) cost-based optimization of partitioned tables. In particular, we address the accompanying non-trivial challenge of plan space explosion—especially in the presence of hundreds of partitions per table (e.g., daily partitions for a year)—and the need to incorporate the new optimization techniques into industry-strength cost-based SQL optimizers.

The cost-based optimization algorithms we present are independent of the physical join methods supported by the DBMS. Parallel DBMSs support several partition-aware join methods including collocated, directed, broadcast, and repartitioned joins (Baru et al., 1995). SCOPE is a system for large-scale data analysis that uses cost-based optimization to select the repartitioning of tables and intermediate results (Zhou et al., 2010). Query optimizers in these systems attempt to minimize data

transfer costs among nodes, which is orthogonal to our work.

Dynamic partitioning: *Selectivity-based partitioning* (Polyzotis, 2005), *content-based routing* (Bizarro et al., 2005), and *conditional plans* (Deshpande et al., 2005) are techniques that enable different execution plans to be used for different subsets of the input data. Unlike our work, these techniques focus on dynamic partitioning of (unpartitioned) tables and data streams rather than exploiting the properties of existing partitions. Easy incorporation into widely-used SQL optimizers is not a focus of Bizarro et al. (2005), Deshpande et al. (2005), or Polyzotis (2005).

Predicate optimization: *Predicate move-around* (Levy et al., 1994) is a query transformation technique that moves predicates among different relations, and possibly query blocks, to generate potentially better plans. *Magic sets* (Bancilhon et al., 1986) represent a complementary technique that can generate auxiliary tables to be used as early filters in a plan. Both techniques are applied in the rewrite phase of query optimization, thereby complementing the cost-based optimization techniques we propose.

9.3 Query Optimization Techniques for Partitioned Tables

Our goal is to generate an efficient plan for a SQL query that contains joins of horizontally partitioned tables. We focus on tables that are partitioned horizontally based on conditions specified on one or more *partitioning attributes* (columns). The condition that defines a partition of a table is an expression involving any number of binary subexpressions of the form *Attr Op Val*, connected by *AND* or *OR* logical operators. *Attr* is an attribute in the table, *Val* is a constant, and *Op* is one of $\{=, \neq, <, \leqslant, >, \geqslant\}$.

Joins in a SQL query can be equi or nonequi joins. The joined tables could have different number of partitions and could be partitioned on multiple attributes, like

in Figure 9.1. Furthermore, the partitions between joined tables need not have one-on-one correspondence with each other. For example, a table may have one partition per month while another table has one partition per day.

Our approach for partition-aware query optimization is based on extending bottom-up query optimizers. We will give an overview of the well-known System R bottom-up query optimizer (Selinger et al., 1979) on which a number of current optimizers are based, followed by an overview of the extensions we make.

A bottom-up optimizer starts by optimizing the smallest expressions in the query, and then uses this information to progressively optimize larger expressions until the optimal physical plan for the full query is found. First, the best *access path* (e.g., table or index scan) is found and retained for each table in the query. The best *join path* is then found and retained for each pair of joining tables R and S in the query. The join path consists of a physical join operator (e.g., hash or merge join) and the access paths found earlier for the tables. Next, the best join path is found and retained for all three-way joins in the query; and so on.

Bottom-up optimizers pay special attention to physical properties (e.g., sort order) that affect the ability to generate the optimal plan for an expression e by combining optimal plans for subexpressions of e. For example, for $R \bowtie S$, the System R optimizer stores the optimal join path for each *interesting sort order* (Selinger et al., 1979) of $R \bowtie S$ that can potentially reduce the plan cost of any larger expression that contains $R \bowtie S$ (e.g., $R \bowtie S \bowtie U$).

Our extensions: Consider the join path selection in a bottom-up optimizer for two partitioned tables R and S. R and S can be base tables or the result of intermediate subexpressions. Let the respective partitions be $R_1, \ldots, R_r$ and $S_1, \ldots, S_s$ $(r, s \geqslant 1)$. We call R and S the *parent tables* in the join, and each R_i (S_j) a *child table*. By default, the optimizer will consider a join path corresponding to $(R_1 \cup R_2 \cdots \cup R_r)$

$\bowtie (S_1 \cup S_2 \cdots \cup S_s)$, i.e., a physical join operator that takes the bag unions of the child tables as input. This approach leads to plans like P_1 in Figure 9.2.

Partition-aware optimization must consider joins among the child tables in order to get efficient plans like P_2 in Figure 9.2; effectively, pushing the join below the union(s). Joins of the child tables are called *child joins*. When the bottom-up optimizer considers the join of partitioned tables R and S, we extend its search space to include plans consisting of the union of child joins. This process works in four phases: *applicability testing*, *matching*, *clustering*, and *path creation*.

Applicability testing: We first check whether the specified join condition between R and S match the partitioning conditions on R and S appropriately. Intuitively, efficient child joins can be utilized only when the partitioning columns are part of the join attributes. For example, the $R.a = S.a$ join condition makes it possible to utilize the $R_2 \bowtie (S_{22} \cup S_{23})$ child join in plan P_2 in Figure 9.2.

Matching: This phase uses the partitioning conditions to determine efficiently which joins between individual child tables of R and S can potentially generate output records, and to prune the empty child joins. For $R \bowtie S$ in our running example query Q, this phase outputs $\{(R_1, S_{12}), (R_1, S_{13}), (R_2, S_{22}), (R_2, S_{23})\}$. The remaining possible child joins are pruned.

Clustering: Production deployments can contain tables with many tens to hundreds of partitions that lead to a large number of joins between individual child tables. To reduce the join path creation overhead and execution inefficiencies, we carefully cluster the child tables. For $R \bowtie S$ in our running example, the matching phase's output is clustered such that only the two child joins $R_1 \bowtie (S_{12} \cup S_{13})$ and $R_2 \bowtie (S_{22} \cup S_{23})$ are considered during path creation.

Path Creation: This phase creates and costs join paths for all child joins output by the clustering phase, as well as the path that represents the union of the best

child-join paths. This path will be chosen for $R \bowtie S$ if it costs lower than the one produced by the optimizer without our extensions.

Next we present the details of these phases, and discuss how our techniques can be incorporated into the bottom-up optimization process.

9.3.1 Matching Phase

Suppose the bottom-up optimizer is in the process of selecting the join path for parent tables R and S with respective child tables $R_1, \ldots, R_r$ and $S_1, \ldots, S_s$ $(r, s \geqslant 1)$. The goal of the matching phase is to identify all *partition-wise join pairs* (R_i, S_j) such that $R_i \bowtie S_j$ can produce output tuples as per the given partitioning and join conditions. Equivalently, this algorithm prunes out (from all possible join pairs) partition-wise joins that cannot produce any results.

An obvious matching algorithm would enumerate and check all the $r \times s$ possible child table pairs. In distributed query optimization, this algorithm is implemented by generating a join graph for the child tables (Ceri and Gottlob, 1986). The real inefficiency from this quadratic algorithm comes from the fact that it gets invoked from scratch for each distinct join of parent tables considered throughout the bottom-up optimization process. Recall that R and S can be base tables or the result of intermediate subexpressions.

Partition Index Trees (PITs): We developed a more efficient matching algorithm that builds, probes, and reuses *Partition Index Trees (PITs)*. The core idea is to associate each child table with one or more *intervals* generated from the table's partitioning condition. An interval is specified as a *Low* to *High* range, which can be numeric (e.g., $(0, 10]$), date (e.g., [02-01-10, 03-01-10)), or a single numeric or categorical value (e.g., $[5, 5]$, [*url,url*]). A PIT indexes all intervals of all child tables for one of the partitioning columns of a parent table. The PIT then enables efficient

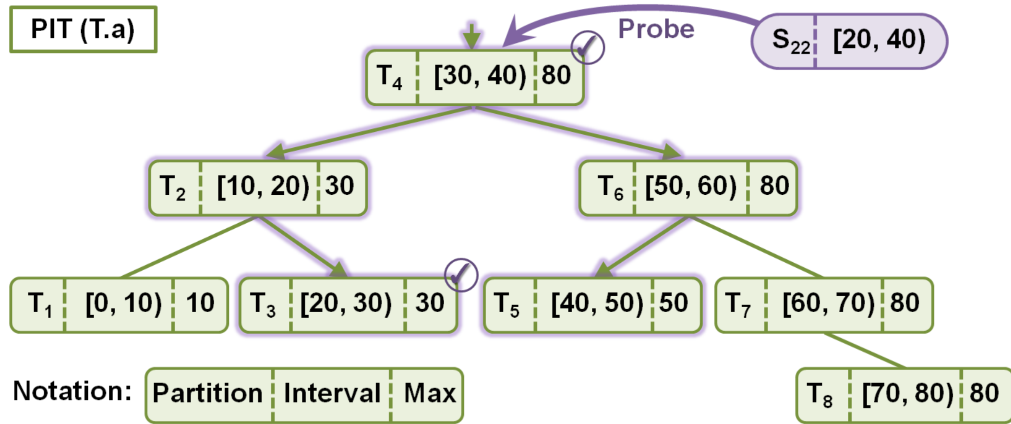


FIGURE 9.3: A partition index tree containing intervals for all child tables (partitions) of T from Figure 9.1.

lookup of the intervals that overlap with a given probe interval from the other table. Use of PITs provides two main advantages:

- For most practical partitioning and join conditions, building and probing PITs has $O(r \log r)$ complexity (for r partitions in a table). The memory needs are $\theta(r)$.
- Most PITs are built once and then reused many times over the course of the bottom-up optimization process.

PIT, at a basic level, is an augmented *red-black tree* (Cormen et al., 2003). The tree is ordered by the *Low* values of the intervals, and an extra annotation is added to every node recording the maximum *High* value (denoted *Max*) across both its subtrees. Figure 9.3 shows the PIT created on attribute $T.a$ based on the partitioning conditions of all child tables of T (see Figure 9.1). The *Low* and *Max* values on each node are used during probing to efficiently guide the search for finding the overlapping intervals. When the interval $[20, 40)$ is used to probe the PIT, five intervals are checked (highlighted in Figure 9.3) and the two overlapping intervals $[20, 30)$ and $[30, 40)$ are returned.

Algorithm for performing the matching phase
Input: Relation R, Relation S, Join Condition
Output: All partition-wise join pairs (R_i,S_j) that can produce join results

```
For each (binary join expression in Join Condition) {
    Convert all partitioning conditions to intervals;
    Build a PIT with intervals from partitions of R;
    Probe the PIT with intervals from partitions of S;
    Adjust matching result based on logical AND or OR semantics of the Join Condition;
}
```

FIGURE 9.4: Matching algorithm.

Matching algorithm: Figure 9.4 provides all the steps for the matching algorithm. The input consists of the two tables to be joined and the join condition. We will describe the algorithm using our running example query Q. The join condition for $S \bowtie T$ in Q is a simple equality expression: $S.a = T.a$. Later, we will discuss how the algorithm handles more complex conditions involving logical AND and OR operators, as well as nonequi join conditions. Since the matching phase is executed only if the Applicability Test passes, the attributes $S.a$ and $T.a$ must appear in the partitioning conditions for the partitions of S and T respectively.

The table with the smallest number of (unpruned) partitions is identified as the *build* relation and the other as the *probe* relation. In our example, T (with 3 partitions) will be the build relation and S (with 4 partitions) will be the probe one. Since partition pruning is performed before any joins are considered, only the unpruned child tables are used for building and probing the PIT. Then, the matching algorithm works as follows:

- *Build phase:* For each child table T_i of T, generate the interval for T_i's partitioning condition. Build a PIT that indexes all intervals from the child tables of T.

- *Probe phase:* For each child table S_j of S, generate the interval *int* for S_j's

partitioning condition. Probe the PIT on $T.a$ to find intervals overlapping with *int*. Only T's child tables corresponding to these overlapping intervals can have tuples joining with S_j; output the identified join pairs.

For $S \bowtie T$ in our running example query, the PIT on $T.a$ will contain the intervals $[0, 10)$, $[10, 20)$ and $[20, 30)$, which are associated with partitions T_1, T_2, and T_3 respectively (see Figure 9.1). When this PIT is probed with the interval $[20, 40)$ for child table S_{22}, the result will be the interval $[20, 30)$, indicating that only T_3 will join with S_{22}. Overall, this phase outputs $\{(S_{12}, T_1)$, (S_{12}, T_2), (S_{13}, T_1), (S_{13}, T_2), (S_{22}, T_3), $(S_{23}, T_3)\}$; the remaining possible child joins are pruned.

Support for complex conditions: The description so far was simplified for ease of presentation. A number of nontrivial enhancements to PITs and the matching algorithm were needed in order to support complex partitioning and join conditions that can arise in practice. First, PITs need support for multiple types of intervals: open, closed, partially closed, one sided, and single values (e.g., $(1, 5)$, $[1, 5]$, $[1, 5)$, $(-\infty, 5]$, and $[5, 5]$). In addition, supporting nonequi joins required support from PITs to efficiently find all intervals in the tree that are to the left or to the right of the probe interval.

Before building and probing the PIT, we need to convert each partitioning and join condition into one or more intervals. A condition could be any complex combinations of AND and OR subexpressions, as well as involve any operator in $\{=, \neq, <, \leqslant, >, \geqslant\}$. Subexpressions that are ANDed together are used to build a single interval, whereas subexpressions that are ORed together will produce multiple intervals. For example, suppose the partitioning condition is ($R.a \geqslant 0$ AND $R.a < 20$). This condition will create the interval $[0, 20)$. The condition ($R.a > 0$ AND $R.b > 5$) will create the interval $(0, \infty)$, since only $R.a$ appears in the join conditions of query Q. The condition ($R.a < 0$ OR $R.a > 10$) will create the intervals $(-\infty, 0)$ and $(10, \infty)$.

If the particular condition does not involve $R.a$, then the interval created is $(-\infty, \infty)$, as any value for $R.a$ is possible.

Our approach can also support nonequi joins, for example $R.a < S.a$. Suppose $A = (A1, A2)$ is an interval in the PIT and $B = (B1, B2)$ is the probing interval. The interval A is marked as an overlapping interval if $\exists \alpha \in A$, $\beta \in B$ such that $\alpha < \beta$. Note that this check is equivalent to finding all intervals in the PIT that overlap with the interval $(-\infty, B2)$.

Finally, we support complex join expressions involving logical ANDs and ORs. Suppose the join condition is ($R.a = S.a$ AND $R.b = S.b$). In this case, two PITs are built; one for $R.a$ and one for $R.b$. After probing the two PITs, we will get two sets of join pairs. We then adjust the pairs based on whether the join conditions are ANDed or ORed together. In the example above, suppose that R_1 can join with S_1 based on $R.a$, and that R_1 can join with both S_1 and S_2 based on $R.b$. Since the two binary join expressions are ANDed together, we induce that R_1 can join only with S_1. However, if the join condition were ($R.a = S.a$ OR $R.b = S.b$), then we would induce that R_1 can join with both S_1 and S_2.

Complexity analysis: Suppose N and M are the number of partitions for the build and probe relations respectively. Also suppose each partition condition is translated into a small, fixed number of intervals (which is usually the case). In fact, a simple range partitioning condition will generate exactly one interval. Then, building a PIT requires $O(N \times \log N)$ time. Probing a PIT with a single interval takes $O(min(N, k \times \log N))$ time, where k is the number of matching intervals. Hence, the overall time to identify all possible child join pairs is $O(M \times min(N, k \times \log N))$.

The space overhead introduced by a PIT is $\theta(N)$ since it is a binary tree. However, a PIT can be reused multiple times during the optimization process. For example, consider the three-way join condition $R.a = S.a$ AND $R.a = T.a$. The same PIT

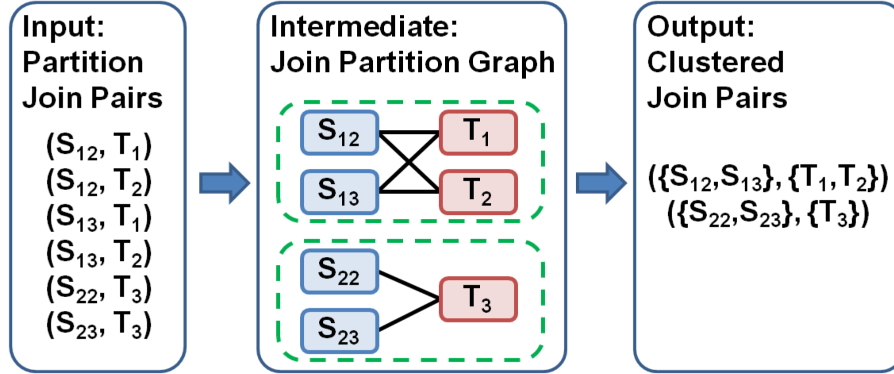


FIGURE 9.5: Clustering algorithm applied to the running example query Q.

on $R.a$ can be (re)used for performing the matching algorithm when considering the joins $R \bowtie S$, $R \bowtie T$, $(R \bowtie S) \bowtie T$, and $(R \bowtie T) \bowtie S$.

9.3.2 *Clustering Phase*

The number of join pairs output by the matching phase can be large, e.g., when each child table of R joins with multiple child tables of S. In such settings, it becomes important to reduce the number of join pairs that need to be considered during join path creation to avoid both optimization and execution inefficiencies. Join path creation introduces optimization-time overheads for enumerating join operators, accessing catalogs, and calculating cardinality estimates. During execution, if multiple child-join paths reference the same child table R_i, then R_i will be accessed multiple times; a situation we want to avoid.

The approach we use to reduce the number of join pairs is to cluster together multiple child tables of the same parent table. Figure 9.5 considers $S \bowtie T$ for our running example query Q from Section 9.1. The six partition-wise join pairs output by the matching phase are shown on the left. Notice that the join pairs (S_{22}, T_3) and (S_{23}, T_3) indicate that both S_{22} and S_{23} can join with T_3 to potentially generate output records. If S_{22} is clustered with S_{23}, then the single (clustered) join $(S_{22} \cup S_{23}) \bowtie T_3$ will be considered in the path creation phase instead of the two

Algorithm for clustering the output of the matching phase
Input: Partition join pairs (output of matching phase)
Output: Clustered join pairs (which will be input to path creation phase)

Build a bipartite join graph from the input partition join pairs where:
    Child tables are the vertices, and
    Partition join pairs are the edges;
Use Breadth-First-Search to identify connected components in the graph;
Output a clustered join pair for each connected component;

FIGURE 9.6: Clustering algorithm.

joins $S_{22} \bowtie T_3$ and $S_{23} \bowtie T_3$. Furthermore, because of the clustering, the child table T_3 will have only one access path (say, a table or index scan) in Q's plan.

Clustering metric: For an $R \bowtie S$ join, two (unpruned) child tables S_j and S_k of S will be clustered together *iff* there exists a (unpruned) child table R_i of R such that the matching phase outputs the join pairs (R_i, S_j) and (R_i, S_k). If S_j and S_k are clustered together when no such R_i exists, then the union of S_j and S_k will lead to unneeded joins with child tables of R; hurting plan performance during execution. In our example in Figure 9.5, suppose we cluster S_{22} with S_{13}. Then, S_{22} will have to be considered unnecessarily in joins with T_1 and T_2.

On the other hand, failing to cluster S_j and S_k together when the matching phase outputs the join pairs (R_i, S_j) and (R_i, S_k) would result in considering join paths separately for $R_i \bowtie S_j$ and $R_i \bowtie S_k$. The result is higher optimization overhead as well as access of R_i in at least two separate paths during execution. In our example, if we consider separate join paths for $S_{22} \bowtie T_3$ and $S_{23} \bowtie T_3$, then partition T_3 will be accessed twice.

Clustering algorithm: Figure 9.6 shows the clustering algorithm that takes as input the join pairs output by the matching phase. The algorithm first constructs the *join partition graph* from the input join pairs. Each child table is a vertex in this bipartite graph, and each join pair forms an edge between the corresponding

vertices. Figure 9.5 shows the join partition graph for our example. Breadth First Search is used to identify all the connected components in the join partition graph. Each connected component will give a (possibly clustered) join pair. Following our example in Figure 9.5, S_{12} will be clustered with S_{13}, S_{22} with S_{23}, and T_1 with T_2, forming the output of the clustering phase consisting of the two (clustered) join pairs $(\{S_{12}, S_{13}\}, \{T_1, T_2\})$ and $(\{S_{22}, S_{23}\}, \{T_3\})$.

9.3.3 Path Creation and Selection

We will now consider how to create and cost join paths for all (clustered) child joins output by the clustering phase, as well as the union of the best child-join paths. We leverage the functionality of a bottom-up query optimizer (Selinger et al., 1979) to create join paths, which are coupled tightly with the physical join operators supported by the database. The main challenge is how to extend the enumeration and path retention aspects of a bottom-up query optimizer in order to find the plan with the least estimated cost (i.e., the *optimal plan*) in the new *extended plan space* efficiently.

Definition 5. Extended plan space: *In addition to the default plan space considered by the bottom-up optimizer for an n-way ($n \geqslant 2$) join of parent tables, the extended plan space includes the plans containing any possible join order and join path for joins of the child tables such that each child table (partition) is accessed at most once.* □

We will discuss three different approaches on how to extend the bottom-up optimizer to find the optimal plan in the extended plan space.

Query Q from Section 9.1 is used as an example throughout. Note that Q joins the three parent tables R, S, and T. For Q, a bottom-up optimizer will consider the three 2-way joins $R \bowtie S$, $R \bowtie T$, $S \bowtie T$, and the single 3-way join $R \bowtie S \bowtie T$.

For each join considered, the optimizer will find and retain the best join path for each interesting order and the best "unordered" path. Sort orders on $R.a$, $S.a$, and $T.a$ are the candidate interesting orders for Q. When the optimizer is considering an n-way join, it only uses the best join paths retained for smaller joins.

Extended enumeration: The first approach is to extend the existing path creation process that occurs during the enumeration of each possible join. The extended enumeration includes the path representing the union of the best child-join paths for the join, in addition to the join over the unions of the child tables. For instance, as part of the enumeration process for query Q, the optimizer will create and cost join paths for $S \bowtie T$.

The conventional join paths include joining the union of S's partitions with the union of T's partitions using all applicable join operators (like hash join or merge join), leading to plans like P_1 in Figure 9.2. At this point, extended enumeration will also create join paths for $(S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)$ and $(S_{22} \cup S_{23}) \bowtie T_3$, find the corresponding best paths, and create the union of the best child-join paths. We will use the notation P_u in this section to denote the union of the best child-join paths.

As usual, the bottom-up optimizer will retain the best join path for each interesting order (a in this case) as well as the best (possibly unordered) overall path. If P_u is the best for one of these categories, then it will be retained. The paths retained will be the only paths considered later when the enumeration process moves on to larger joins. For example, when creating join paths for $(S \bowtie T) \bowtie R$, only the join paths retained for $S \bowtie T$ will be used (in addition to the best access paths retained for R).

Property 6. *Adding extended enumeration to a bottom-up optimizer will not always find the optimal plan in the extended plan space.* □

We will prove Property 6 using our running example. Suppose plan P_u for $S \bowtie T$ is

not retained because it is not a best path for any order. Without P_u for $S \bowtie T$, when the optimizer goes on to consider $(S \bowtie T) \bowtie R$, it will not be able to consider any 3-way child join. Thus, plans similar to P_2 from Figure 9.2 will never be considered; thereby losing the opportunity to find the optimal plan in the extended plan space.

Treating partitions as a physical property: The next approach considers partitioning as a physical property of tables and joins. The concept of *interesting partitions* (similar to interesting orders) can be used to incorporate partitioning as a physical property in the bottom-up optimizer (Rao et al., 2002). In our example query Q, partitions on attributes $R.a$, $S.a$, and $T.a$ are interesting.

Paths with interesting partitions can make later joins and grouping operations less expensive when these operations can take advantage of the partitioning. For example, partitioning on $S.a$ for $S \bowtie T$ could lead to the creation of three-way child joins for $R \bowtie S \bowtie T$. Hence, the optimizer will retain the best path for each interesting partition, in addition to each interesting order. Overall, if there are n interesting orders and m interesting partitions, then the optimizer can retain up to $n \times m$ paths, one for each combination of interesting orders and interesting partitions.

Property 7. *Treating partitioning as a physical property in a bottom-up optimizer will not always find the optimal plan in the extended plan space.* □

Once again we will prove the above property using the example query Q. When the optimizer enumerates paths for $S \bowtie T$, it will consider the union of the best child-join paths P_u. Unlike what happened in extended enumeration, P_u will now be retained, since P_u has an interesting partition on $S.a$. Suppose the first and second child joins of P_u have the respective join paths $(S_{12} \cup S_{13})$ HJ $(T_1 \cup T_2)$ and $(S_{22} \cup S_{23})$ HJ T_3. Also, the best join path for $S \bowtie T$ with an interesting order on $S.a$ is the union of the child-join paths $(S_{12} \cup S_{13})$ MJ $(T_1 \cup T_2)$ and $(S_{22} \cup S_{23})$ MJ T_3.

However, it can still be the case that the optimal plan for Q is plan P_2 shown in

Figure 9.2. Note that P_2 contains $(S_{12} \cup S_{13})$ MJ $(T_1 \cup T_2)$: the interesting order on $S.a$ in this child join led to a better overall plan. However, the interesting order on $S.a$ was not useful in the case of the second child join of $S \bowtie T$, so $(S_{22} \cup S_{23})$ MJ T_3 is not used in P_2. Simply adding interesting partitions alongside interesting orders to a bottom-up optimizer will not enable it to find the optimal plan P_2.

The optimizer was not able to generate plan P_2 in the above example because it did not consider interesting orders independently for each child join. Instead, the optimizer considered interesting orders and interesting partitions at the level of the parent tables (R, S, T) and joins of parent tables ($R \bowtie S$, $R \bowtie T$, $S \bowtie T$, $R \bowtie S \bowtie T$) only. An apparent solution would be for the optimizer to create union plans for all possible combinations of child-join paths with interesting orders. However, the number of such plans is exponential in the number of child joins per parent join, rendering this approach impractical.

Treating partitions as a logical property: Our approach eliminates the aforementioned problems by treating partitioning as a property of the *logical relations* (tables or joins) that are enumerated during the bottom-up optimization process. A logical relation refers to the output produced by either accessing a table or joining multiple tables together. For example, the logical relation (join) RST represents the output produced when joining the tables R, S, and T, irrespective of the join order or the join operators used in the physical execution plan. Figure 9.7 shows all logical relations created during the enumeration process for our example query Q.

As illustrated in Figure 9.7, each logical relation (table or join) maintains: (i) a list of logical child relations (child tables or child joins), (ii) the partitioning conditions, which are propagated up the enumeration lattice when the child joins are created, and (iii) the best paths found so far for each interesting order and the best unordered path. A logical child table is created for each unpruned partition during partition pruning,

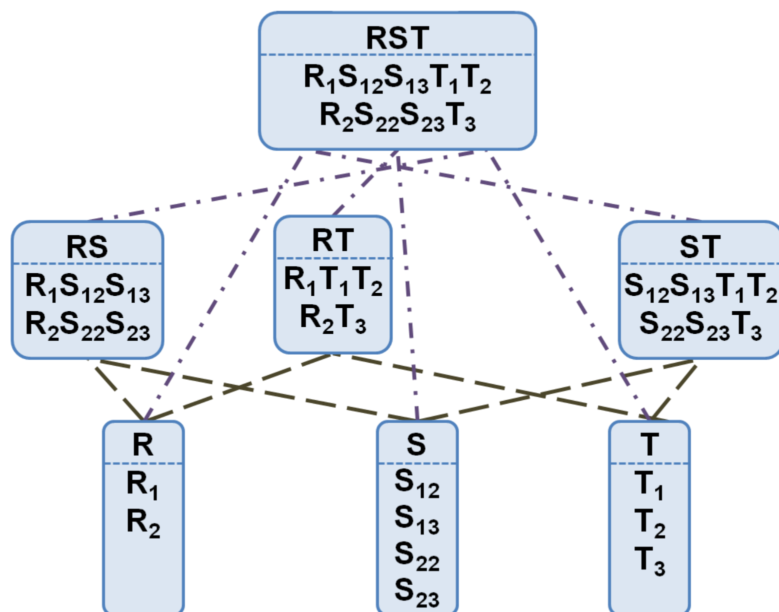


FIGURE 9.7: Logical relations (with child relations) enumerated for query Q by our partition-aware bottom-up optimizer.

whereas logical child joins are created based on the output of the clustering phase. For our example query Q, the child-join pairs $(\{S_{12}, S_{13}\}, \{T_1, T_2\})$ and $(\{S_{22}, S_{23}\}, \{T_3\})$ output by the clustering phase are used to create the respective logical child joins $S_{12}S_{13}T_1T_2$ and $S_{22}S_{23}T_3$. Note that both the matching and clustering phases work at the logical level, independent of physical plans (paths).

The logical relations are the entities for which the best paths found so far during the enumeration process are retained. The logical child joins behave in the same way as their parent joins, retaining the best paths for each interesting order and the best unordered path. Hence, the number of paths retained is *linear* in the number of child joins per parent join (instead of exponential as in the case when partitions are treated as physical properties). The optimizer considers all child-join paths with interesting orders during path creation for higher child joins, while ensuring the property:

Property 8. *Paths with interesting orders for a single child join can be used later up the lattice, independent from all other child joins of the same parent relation.* □

Suppose, the optimizer is considering joining ST with R to create paths for RST. The output of the clustering phase will produce the two child-join pairs $(S_{12}S_{13}T_1T_2, R_1)$ and $(S_{22}S_{23}T_3, R_2)$. Join paths for these two child joins will be created and costed independently from each other, using any paths with interesting orders and join operators that are available. The best join paths for $((S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)) \bowtie R_1$ and $((S_{22} \cup S_{23}) \bowtie T_3) \bowtie R_2)$ will be retained in the logical relations $R_1S_{12}S_{13}T_1T_2$ and $R_2S_{22}S_{23}T_3$ respectively (see Figure 9.7).

For each parent relation, the path representing the union of the best child-join paths is created only at the end of each enumeration level[2] and it is retained only if it is the best path. Hence, the optimizer will consider all join orders for each child join before creating the union, leading to the following property:

Property 9. *The optimizer will consider plans where different child joins of the same parent relation can have different join orders and/or join operators.* □

We have already seen how the optimizer created join paths $((S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)) \bowtie R_1$ and $((S_{22} \cup S_{23}) \bowtie T_3) \bowtie R_2$ when joining ST with R. Later, the optimizer will consider joining RS with T, creating join paths for $(R_1 \bowtie (S_{12} \cup S_{13})) \bowtie (T_1 \cup T_2)$ and $(R_2 \bowtie (S_{22} \cup S_{23})) \bowtie T_3$. It is possible that the best join path for $((S_{12} \cup S_{13}) \bowtie (T_1 \cup T_2)) \bowtie R_1$ is better than that for $(R_1 \bowtie (S_{12} \cup S_{13})) \bowtie (T_1 \cup T_2)$, while the opposite occurs between $((S_{22} \cup S_{23}) \bowtie T_3) \bowtie R_2$ and $(R_2 \bowtie (S_{22} \cup S_{23})) \bowtie T_3$; which leads to the plan P_2 in Figure 9.2.

Property 10. Optimality guarantee: *By treating partitioning as a logical property, our bottom-up optimizer will find the optimal plan in the extended plan space.*

□

This property is a direct consequence of Properties 8 and 9. We have extended the plan space to include plans containing unions of child joins. Each child join

[2] Enumeration level n refers to the logical relations representing all possible n-way joins.

is enumerated during the traditional bottom-up optimization process in the same way as its parent; the paths are built bottom-up, interesting orders are taken into consideration, and the best paths are retained. Since each child join is optimized independently, the top-most union of the best child-join paths is the optimal union of the child joins. Finally, recall that the union of the best child-join paths is created at the end of each enumeration level and retained only if it is the best plan for its parent join. Therefore, the full extended plan space is considered and the optimizer will be able to find the optimal plan (given the current database configuration, cost model, and physical design).

Traditionally, grouping (and aggregation) operators are added on top of the physical join trees produced by the bottom-up enumeration process (Selinger et al., 1979). In this case, interesting partitions are useful for pushing the grouping below the union of the child joins, in an attempt to create less expensive execution paths. With our approach, paths with interesting partitions on the grouping attributes can be constructed at the top node of the enumeration lattice, and used later on while considering the grouping operator.

Treating partitions as a property of the logical relations allows for a clean separation between the enumeration process of the logical relations and the construction of the physical plans. Hence, our algorithms are applicable to any Database system that uses a bottom-up optimizer. Moreover, they can be adapted for non-database data processing systems like SCOPE and Hive that offer support for table partitioning and joins.

9.3.4 Extending our Techniques to Parallel Database Systems

While this paper focuses on centralized DBMSs, our work is also useful in parallel DBMSs like Aster nCluster (AsterData, 2012), Teradata (Teradata, 2012), and HadoopDB (Abouzeid et al., 2009), which try to partition tables such that most

queries in the workload need intra-node processing only. A common data placement strategy in parallel DBMSs is to use hash partitioning to distribute tuples in a table among the nodes $N_1, \ldots, N_k$, and then use range/list partitioning of the tuples within each node. Our techniques extend to this setting: if two joining tables R and S have the same hash partitioning function and the same number of partitions, then a partition-wise join $R_i \bowtie S_i$ is created for each node N_i. If a secondary range/list partitioning has been used to further partition R_i and S_i at an individual node, then our techniques can be applied directly to produce child joins for $R_i \bowtie S_i$.

Another data placement strategy popular in data warehouses is to replicate the dimension tables on all nodes, while the fact table is partitioned across the nodes. The fact-table partition as well as the dimension tables may be further partitioned on each node, so our techniques can be used to create child joins at each node. In such settings, multi-dimensional partitioning of the fact table can improve query performance significantly as we show in Section 9.4.

9.4 Experimental Evaluation

The purpose of this section is to evaluate the effectiveness and efficiency of our optimization techniques across a wide range of factors that affect table partitioning. We have prototyped our techniques in the PostgreSQL 8.3.7 optimizer. All experiments were run on Amazon EC2 nodes of m1.large type. Each node runs 64-bit Ubuntu Linux 10.04 and has 7.5GB RAM, dual-core 2GHz CPU, and 850GB of storage. We used the TPC-H benchmark with scale factors ranging from 10 to 40, with 30 being the default scale. Following directions from the TPC-H Standard Specifications (TPC, 2009), we partitioned tables only on primary key, foreign key, and/or date columns. We present experimental results for a representative set of 10 out of the 22 TPC-H queries, ranging from 2-way up to the maximum possible 8-way joins. All results presented are averaged over three query executions.

Table 9.2: Optimizer categories considered in the experimental evaluation.

Name	Features
Basic	Per-table partition pruning only (like MySQL and PostgreSQL). The PostgreSQL 8.3.7 optimizer is used as the Basic optimizer.
Intermediate	Per-table partition pruning and one-to-one partition-wise joins (like Oracle and SQLServer). The Intermediate optimizer is implemented as a variant of the Advanced optimizer that checks for and creates one-to-one partition-wise join pairs in place of the regular matching and clustering phases.
Advanced	Per-table partition pruning and all the join optimizations for partitioned tables as described in Section 9.3.

For evaluation purposes, we categorized query optimizers into three categories—*Basic*, *Intermediate*, and *Advanced*—based on how they exploit partitioning information to perform optimization. Details are given in Table 9.2. We compare the optimizers on three metrics used to evaluate optimizers (Giakoumakis and Galindo-Legaria, 2008): (i) query execution time, (ii) optimization time, and (iii) optimizer's memory usage.

9.4.1 Results for Different Partitioning Schemes

The most important factor affecting query performance over partitioned tables is the partitioning scheme that determines which tables are partitioned and on which attribute(s). We identified two cases that arise in practice:

1. The DBA has full control in selecting and deploying the partitioning scheme to maximize query-processing efficiency.

2. The partitioning scheme is forced either partially or fully by practical reasons beyond query-processing efficiency.

Results from DBA-controlled schemes: Given the capabilities of the query optimizer, the DBA has a spectrum of choices regarding the partitioning scheme (Zilio

Table 9.3: Partitioning schemes for TPC-H databases.

Partition Scheme	**Table**	**Partitioning Attributes**	**Number of Partitions**
PS-P	orders	o_orderdate	28
	lineitem	l_shipdate	85
PS-J	orders	o_orderkey	48
	lineitem	l_orderkey	48
	partsupp	ps_partkey	12
	part	p_partkey	12
PS-B	orders	o_orderkey, o_orderdate	72
	lineitem	l_orderkey, l_shipdate	120
	partsupp	ps_partkey	12
	part	p_partkey	6
PS-C	orders	o_orderkey, o_orderdate	36
	lineitem	l_orderkey, l_shipdate	168
	partsupp	ps_partkey	30
	part	p_partkey	6
	customer	c_custkey	6

et al., 1994). In one extreme, the DBA can partition tables based on attributes appearing in filter conditions in order to take maximum advantage of partition pruning. At the other extreme, the DBA can partition tables based on joining attributes in order to take maximum advantage of one-to-one partition-wise joins; assuming the optimizer supports such joins (like the Intermediate optimizer in Table 9.2). In addition, our techniques now enable the creation of multi-dimensional partitions to take advantage of both partition pruning and partition-wise joins. We will refer to the three above schemes as partitioning schemes respectively for pruning (PS-P), for joins (PS-J), and for both (PS-B). Table 9.3 lists all partitioning schemes used in our evaluation.

Figure 9.8(a) shows the execution times for the plans selected by the three query optimizers for the ten TPC-H queries running on the database with the PS-J scheme. The Intermediate and Advanced optimizers are able to generate a better plan than the Basic optimizer for all queries, providing up to an order of magnitude benefit

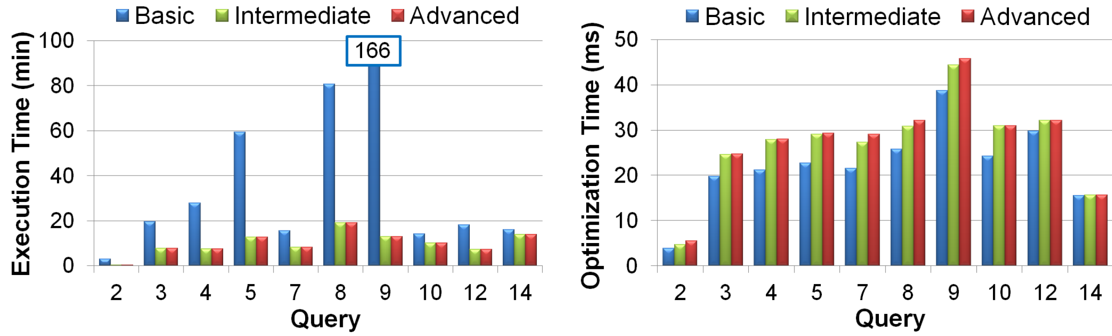


FIGURE 9.8: (a) Execution times and (b) optimization times for TPC-H queries over partitioning scheme PS-J.

for some of them. Note that the Intermediate and Advanced optimizers produce the same plan in all cases, since one-to-one partition-wise joins are the only join optimization option for both optimizers for the PS-J scheme.

Figure 9.8(b) presents the corresponding optimization times for the queries. The Intermediate optimizer introduces some overhead on optimization time—average of 17% and worst case of 21% due to the creation of child-join paths—compared to Basic. The additional overhead introduced by the Advanced optimizer over Intermediate is on average less than 3%. This overhead is due to the matching and clustering algorithms. Overall, the optimization overhead introduced by Advanced is low, and is most definitely gained back during execution as we can see by comparing the y-axes of Figures 9.8(a) and 9.8(b) (execution time is in minutes whereas optimization time is in milliseconds). The memory overheads follow the same trend: average memory overhead of Advanced over Basic is around 7%, and the worst case is 10%.

Query performance is related directly to the optimizer capabilities and the partitioning scheme used in the database. Figure 9.9 shows the performance results for TPC-H queries 5 and 8 for the three optimizers over databases with different partitioning schemes. (Results for other queries are similar.) Since a database using the PS-P scheme only allows for partition pruning, all three optimizers behave in an

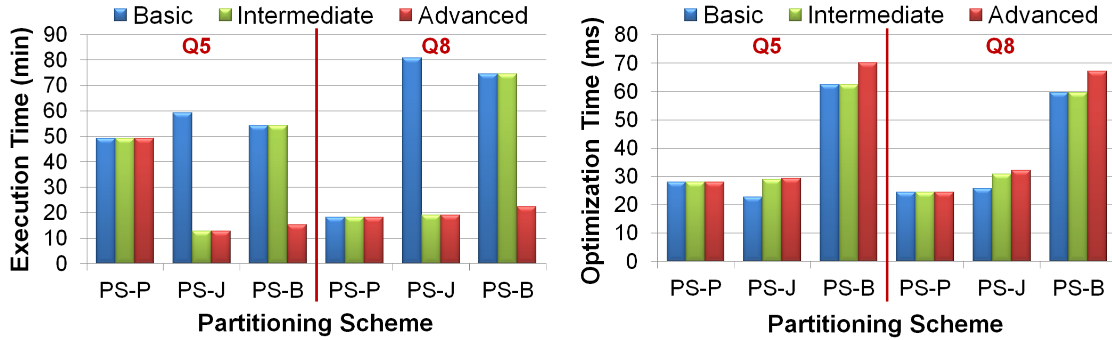


FIGURE 9.9: (a) Execution times and (b) optimization times for TPC-H queries 5 and 8 over three partitioning schemes.

identical manner. A PS-J scheme on the other hand, does not allow for any partition pruning since join attributes do not appear in filter conditions in the queries. Hence, the Basic optimizer performs poorly in many cases, whereas the Intermediate and Advanced optimizers take advantage of partition-wise joins to produce better plans with very low overhead.

The presence of multi-dimensional partitions in a PS-B scheme prevents the Intermediate optimizer from generating any one-to-one partition-wise joins, but it can still perform partition pruning, just like the Basic optimizer can. The Advanced optimizer utilizes both partition pruning and partition-wise joins to find better-performing plans. Consider the problem of picking the best partitioning scheme for a given query workload. The best query performance can be obtained either from (a) partition pruning (PS-P is best for query 8 in Figure 9.9), or (b) from partition-aware join processing (PS-J is best for query 5 in Figure 9.9), or (c) from a combination of both due to some workload or data properties. In all cases, the Advanced optimizer enables finding the plan with the best possible performance.

Results from constrained schemes: In many cases, external constraints or objectives can limit the partitioning scheme that can be used. For instance, data arrival rates may require the creation of daily or weekly partitions; file-system properties

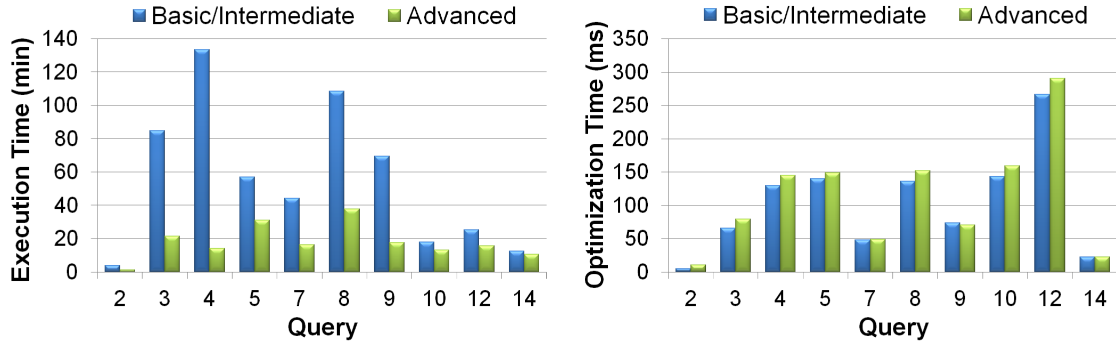


FIGURE 9.10: (a) Execution times and (b) optimization times for TPC-H queries over partitioning scheme PS-C with partition size 128MB.

may impose a maximum partition size to ensure that each partition is laid out contiguously; or optimizer limitations may impose a maximum number of partitions per table.

For a TPC-H scale factor of 30, biweekly partitions of the fact table lead to a 128MB partition size. We will impose a maximum partition size of 128MB to create the partitioning scheme PS-C used in this section (see Table 9.3). Figure 9.10 shows the results for the TPC-H queries executed over a database with the PS-C scheme. The constraint imposed on the partitioning scheme does not allow for any one-to-one partition-wise joins. Hence, the Intermediate optimizer produces the same plans as Basic, and is excluded from the figures for clarity. Once again, the Advanced optimizer was able to generate a better plan than the Basic optimizer for all queries, providing over $2x$ speedup for 50% of them. The average optimization time and memory overheads were just 7.9% and 3.6% respectively.

9.4.2 Studying Optimization Factors on Table Partitioning

In this section, we study the effects of our optimization techniques on execution time, optimization time, and memory usage by varying (i) the number and size of partitions used to split each table, (ii) the amount of data residing in the Database system, and (iii) the use of the clustering algorithm.

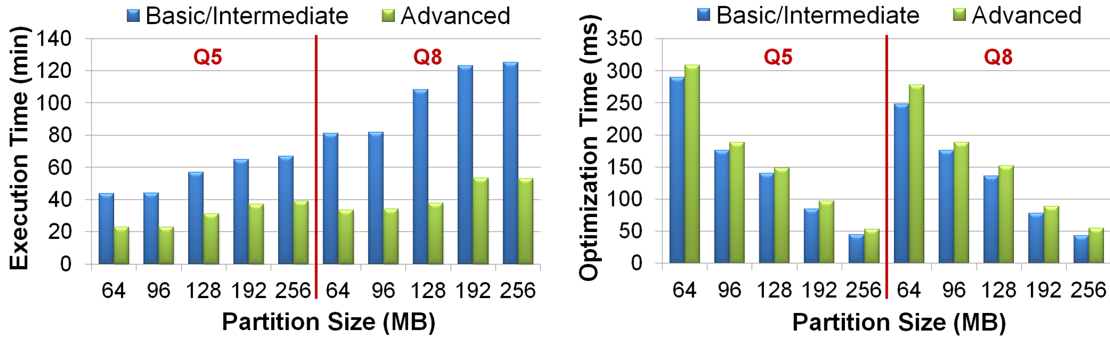


FIGURE 9.11: (a) Execution times and (b) optimization times as we vary the partition size for TPC-H queries 5 and 8.

Effect of size and number of partitions: We evaluate the performance of the optimizers as we vary the size (and thus the number) of partitions created for each table, using the PS-C scheme. As we vary the partition size from 64MB to 256MB, the number of partitions for the fact table vary from 336 to 84. Figure 9.11(b) shows the optimization times taken by the two optimizers for TPC-H queries 5 and 8. As the partition size increases (and the number of partitions decreases), the optimization time decreases for both optimizers. We observe that (i) the optimization times for the Advanced optimizer scale in a similar way as for the Basic optimizer, and (ii) the overhead introduced by the creation of the partition-wise joins remains small (around 12%) in all cases.

The overhead added by our approach remains low due to two reasons. First, Clustering bounds the number of child joins for $R \bowtie S$ to *min(number of partitions in R, S)*; so we cause only a linear increase in paths enumerated per join. Second, optimizers have other overheads like parsing, rewrites, scan path enumeration, catalog and statistics access, and cardinality estimation. Let us consider Query 5 from Figure 9.11(b). Query 5 joins five tables, including *orders* and *lineitem* with 72 and 336 partitions respectively. In this case, Basic enumerated 2317 scan and join paths in total, while Advanced enumerated 2716 paths. The extra 17% paths are for the

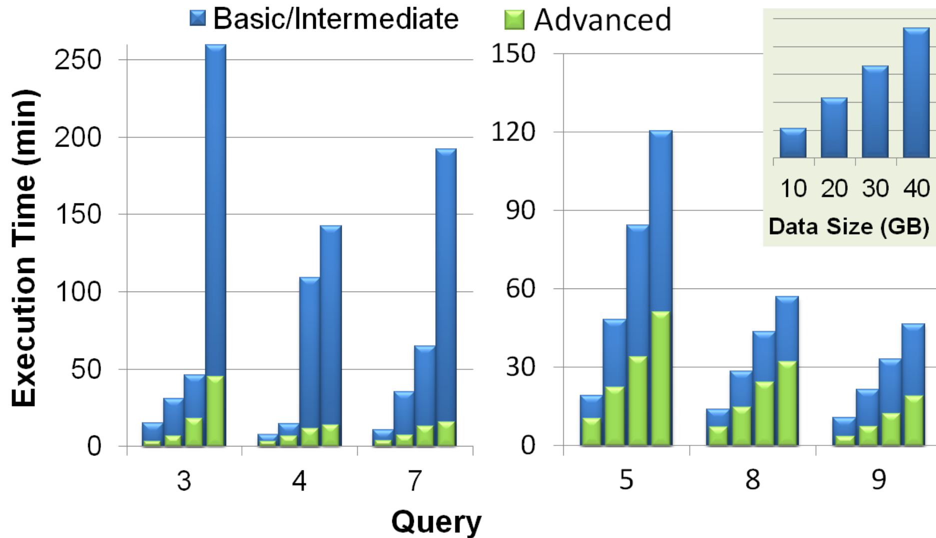


FIGURE 9.12: Execution times as we vary the total data size.

72 partition-wise joins created by Advanced. The trends are similar for the memory consumption of the optimizers. Hence, our algorithms scale well with the number of table partitions.

Decreasing the partition size for the same total data size has a positive effect on plan execution times as seen in Figure 9.11(a): smaller partition sizes force finer-grained partition ranges, leading to better partition pruning and join execution. Looking into execution times at the subplan level, we observed that PostgreSQL was more effective in our experimental settings when it accessed partitions in the 64MB range. It is worth noting that current partitioning scheme recommenders (Agrawal et al., 2004; Rao et al., 2002; Zilio et al., 1994) do not consider partition size tuning.

Effect of data size: We used the PS-C scheme with a partition size of 128 MB to study the effects of the overall data size on query performance. Figure 9.12 shows the query execution times as the amount of data stored in the database increases. For many queries, the plans selected by the Basic optimizer lead to a quadratic or exponential increase in execution time as data size increases linearly. We observed that joins for large data sizes cause the Basic optimizer to frequently resort to index

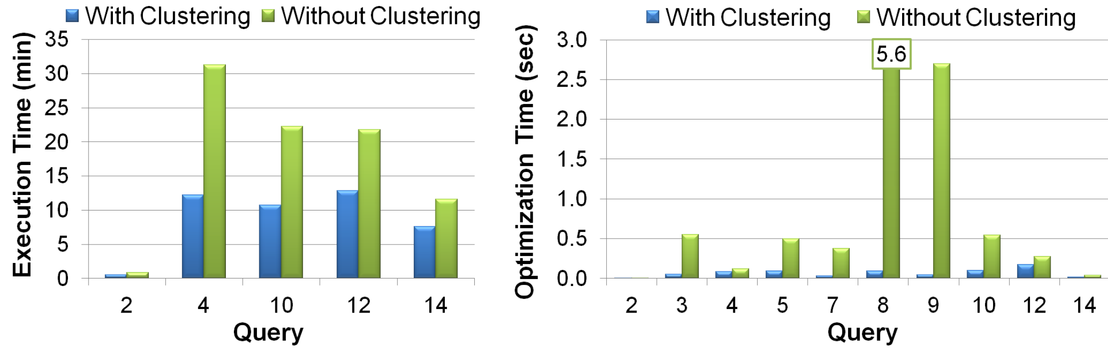


FIGURE 9.13: (a) Execution times and (b) optimization times for enabling and disabling clustering.

nested loop joins (the system has 7.5GB RAM only).

On the other hand, the Advanced optimizer is able to generate smaller partition-wise joins that use more efficient join methods (like hash and merge joins); leading to the desired linear increase in execution time as data size increases linearly. For the queries where the Basic optimizer is also able to achieve a linear trend, the slope is much higher compared to the Advanced optimizer. Figure 9.12 shows that the benefits from our approach become more important for larger databases. Note that optimization times and memory consumption are independent of the data size.

Effect of the clustering algorithm: Clustering (Section 9.3.2) is an essential phase in our overall partition-aware optimization approach that is missing from the data localization approach discussed in Section 9.2. When matching is applied without clustering, our optimizer implements a rough equivalent of the four-phase approach to distributed query optimization (Ozsu and Valduriez, 1999). Figure 9.13(b) compares the optimization time of the optimizer when clustering is enabled and disabled in a database with the PS-C scheme. Disabling clustering causes high overhead since the optimizer must now generate join paths for each child join produced by the matching phase. This issue has not come up in the distributed optimization literature because the implementation and evaluation there considers a small number of partitions (<10

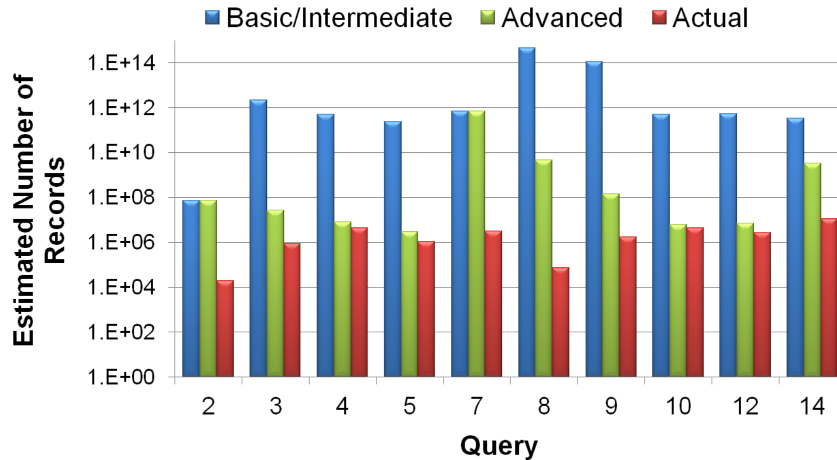


FIGURE 9.14: Estimated and actual number of records produced by TPC-H queries over partitioning scheme PS-C.

partitions), while our optimizer has to perform well under hundreds of partitions per table (e.g., daily partitions for 2 years).

Figure 9.13(a) shows the execution times for the plans generated when enabling and disabling clustering. In all cases shown, the plan generated without clustering is worse than the plan generated when clustering is used, since the generated plans scan the same partitions multiple times (in different joins). The queries that are not shown failed to complete because the system runs out of memory during plan execution. Note that with multi-dimensional partitioning and without clustering, literally thousands of child join paths are created, each requiring a small amount of memory during their initialization phase. We conclude that the use of clustering is crucial for finding good execution plans.

9.4.3 Impact on Cardinality Estimation

An additional benefit that child joins bring is better cardinality estimation for costing during path creation. Cardinality estimation for filter and join conditions is based on data-level statistics kept by the Database system for each table (e.g., distribution histograms, minimum and maximum values, number of distinct values). For

partitioned tables, databases like Oracle and PostgreSQL collect statistics for each individual partition. When the optimizer considers joining unions of partitions, the cardinality estimates needed are derived by aggregating statistics over partitions. This process, however, can give inaccurate estimates since many typical statistics kept by the system cannot be readily aggregated. For instance, estimating the number of unique values for all partitions is not possible by simply combining the number of unique values for each partition.

Figure 9.14 shows the estimated and actual number of records of TPC-H queries over the PS-C scheme. For the Basic Optimizer, we observe large cardinality errors. In contrast, partition-wise joins provide much more accurate cardinality estimation because these joins increase the chances of using partition-level statistics directly for costing. The same pattern was observed with all the partitioning schemes and queries used.

10

The Future of Big Data Analytics

Over the next decade, the number of servers (virtual and physical) worldwide are predicted to grow by a factor of 10, the amount of information managed by enterprise data centers to grow by a factor of 50, and the number of files in a given data center to grow by a factor of 75 (Gantz and Reinsel, 2011). However, the number of Information Technology (IT) professionals in the world that are responsible for managing and capitalizing on this explosive growth of information is predicted to grow by less than a factor of 1.5. It is imperative that new, flexible, scalable, and, most importantly, *self-tuning* systems are developed to face the challenges in the Big Data era.

The thesis of this work is that the proposed *profile-predict-optimize* approach can form the basis for automatically tuning both Database and Dataflow systems used to analyze Big Data. To support this thesis, we designed, implemented, and evaluated solutions for systems belonging in both categories. *Starfish*, a system that is built on top of Hadoop MapReduce, uses the profile-predict-optimize approach for tuning repeatedly-run as well as ad-hoc MapReduce workloads. *Xplus*, on the other hand,

employs this technique repeatedly to perform fine-grained tuning of SQL queries executing in Database systems. In the process, we have made numerous contributions in the areas of automated management, dynamic optimization, and tuning, while creating a large number of future challenges and opportunities to address.

10.1 Starfish: Present and Future

Starfish builds on the Hadoop platform while adapting to user needs and system workloads to provide good performance automatically throughout the data lifecycle in analytics; without any need for users to understand and manipulate the many tuning knobs available. A system like Starfish is essential as Hadoop usage continues to grow beyond companies like Facebook and Yahoo! that have considerable expertise in Hadoop to new application domains as well as to organizations with few expert users. The Starfish source code has been released publicly and the Starfish system has gained several external users in both academia and industry.

Starfish implements, to the best of our knowledge, the *first* cost-based optimization framework for MapReduce that can handle simple to arbitrarily complex MapReduce programs. Our approach is applicable to optimizing the execution of MapReduce jobs and workflows regardless of whether they are submitted directly by the user or come from a higher-level system like Hive, Jaql, or Pig.

We focused on the optimization opportunities presented by the large space of configuration parameters in Hadoop as well as the space of cluster resources. The supported optimization space can be extended in the future to include logical decisions such as selecting the best partitioning function, join operator, and data layout. In these cases, the *profiles* proposed in Chapter 4 will form the basic unit of statistical information that higher-level optimizers can use while making their optimization decisions.

We proposed a lightweight *Profiler* to collect detailed statistical information from

unmodified MapReduce programs. The Profiler, with its task-level sampling support, can be used to collect profiles online while MapReduce jobs are executed on the production cluster. Novel opportunities arise from storing these job profiles over time, including tuning the execution of MapReduce jobs adaptively within a job execution and across multiple executions. New policies are needed to decide when to turn on dynamic instrumentation and which stored profile to use as input for a given what-if question or optimization request.

We also proposed a *What-if Engine* for the fine-grained cost estimation needed by the *Cost-based Optimizers*. A promising direction for future work is to integrate the What-if Engine with tools like data layout advisors, dynamic and elastic resource allocators, resource-aware job schedulers, and progress estimators for complex MapReduce workflows.

In addition to optimizing configuration parameters for individual MapReduce jobs, we showed the importance of optimizing them for MapReduce workflows in an automated and interaction-aware manner. Efficient solutions for this problem improve the performance of the entire MapReduce stack, irrespective of the higher-level interface used to specify workflows. Furthermore, the growing usage of data-intensive workflows beyond large Web companies to smaller groups with few human tuning experts makes such automatic optimization timely.

The primary challenge we faced in the workflow optimization problem was that the space of configuration parameter settings for a workflow W comprises the huge cartesian product of the individual configuration spaces of all jobs in W. We developed optimizers that traverse this space efficiently based on a characterization of the dataflow-based and resource-based interactions that can arise in W. We also developed interaction-aware techniques for profiling as well as answering what-if questions on workflow performance.

Infrastructure-as-a-Service (IaaS) cloud platforms allow nonexpert users to pro-

vision clusters of any size on the cloud to run their MapReduce workloads, and pay only for the resources used. However, these users are now faced with complex *cluster sizing problems* that involve determining the cluster resources, in addition to MapReduce configuration settings, to meet desired requirements on execution time and cost for a given analytic workload. Users can express their cluster sizing problems as queries in a declarative fashion to Starfish. Starfish will provide reliable answers to these queries using an automated technique; providing nonexpert users with a good combination of cluster resource and job configuration settings to meet their needs. The automated technique is based on a careful mix of job profiling, estimation using black-box and white-box models, and simulation.

Multi-tenancy is a key characteristic of IaaS cloud platforms that can cause variations in the performance of MapReduce workloads. In other words, the execution time of a particular MapReduce job j can vary based on what other MapReduce jobs and background processes run concurrently with j in the cluster. It would be interesting to study the effects of multi-tenancy and to enable Starfish to recommend robust configuration settings for running a MapReduce workload under different conditions. This work will require extensions to the simulation process used by the What-if Engine. Another interesting avenue for future work is to add support for auction-based resource provisioning, for example, spot instances on Amazon EC2.

Finally, all Starfish components were prototyped and evaluated for the popular Hadoop MapReduce system using representative MapReduce programs from various application domains, as well as a variety of benchmarks from Facebook, Yahoo!, and TPC. The results showed that, not only is our automated cost-based optimization able to match manual rule-based optimization used in real-life MapReduce deployments, but also routinely outperforms manual tuning by 1.2-2.5x for complex workflows.

10.2 Xplus: Present and Future

As a novel query optimizer, Xplus is the right entity to automate the important task of SQL tuning. An Xplus user can mark a repeatedly-run query for which she is not satisfied with the performance of the plan being picked; and Xplus will try to find a new plan that gives the desired performance. Xplus differs from regular query optimizers in its ability to run plans proactively, and to collect monitoring data from these runs to diagnose its mistakes as well as to identify better plans.

A key contributor to the effectiveness and efficiency of Xplus is the abstraction of *plan neighborhoods* in the physical plan space. Plan neighborhoods are used to capture useful relationships among plans that simplify information tracking; allowing Xplus to overcome the nontrivial challenge of choosing a small set of plans to run from the huge plan space in order to improve efficiency in SQL tuning. Efficiency is further improved through features like subplan selection and parallel execution.

We showed how the two conflicting objectives of exploitation and exploration need to be balanced in effective SQL tuning. Xplus uses an architecture based on multiple SQL-tuning experts with different goals, and an arbitration policy to achieve this balance. Perhaps one of the most important features that Xplus has to offer is the strong optimality guarantee: after completing the tuning process of a query Q, Xplus produces Q's optimal plan for the current database configuration and optimizer cost model; with all plans costed using accurate cardinalities. Finally, we validated the promise of Xplus through an extensive evaluation based on tuning scenarios that arise in practice.

As Xplus has to run subplans to collect information, it uses an *experiment-driven* approach to address the problem of SQL tuning. The use of experiments for SQL tuning (or any other tuning task) poses a tradeoff between costs and benefits. Emerging mechanisms, like *cloud computing*, enable the use of experiments on a larger scale.

For example, Amazon EC2 provides cheap resources that can be leveraged for experiments. At the same time, it creates challenging research questions like how to get the data into the cloud, and how to leverage parallelism in this context.

In the Database world, query optimization technology has not kept pace with the growing usage and user control of table partitioning. We addressed this gap by developing novel partition-aware optimization techniques to generate efficient plans for SQL queries over partitioned tables. We extended the search space to include plans with multiway partition-wise joins, and provided techniques to find the optimal plan efficiently. Our techniques are designed for easy incorporation into the bottom-up query optimizers that are in widespread use today. An extensive experimental evaluation showed that our optimizer, with low optimization-time overhead, can generate plans that are an order of magnitude better than plans generated by current optimizers. Integrating our optimizer with physical design advisors is a promising avenue for future work. Another promising future direction is to apply partition-aware query optimization techniques to parallel Database systems—both multicore-based and shared-nothing like Aster nCluster, Greenplum Database, HadoopDB, and Teradata—as well as to MapReduce-oriented systems like Pig and Hive.

Closing remarks: Currently, the technology for cost-based optimization and self-tuning in Dataflow systems lags behind the corresponding technology in Database systems. Our work on Starfish has made significant contributions and raised the bar for Dataflow systems. At the same time, our work on Xplus and partition-aware query optimization has raised the bar even higher for Database systems. Most importantly, we have introduced a unified theme—the profile-predict-optimize approach—to move cost-based optimization and self-tuning in these two systems forward, especially as these systems themselves are evolving in accordance with the MADDER principles. I look forward to future work from myself as well as others to further bridge the gap in cost-based optimization and self-tuning between these systems.

Bibliography

Aboulnaga, A. and Chaudhuri, S. (1999), "Self-Tuning Histograms: Building Histograms without Looking at Data," *ACM SIGMOD Record*, 28, 181–192.

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Rasin, A., and Silberschatz, A. (2009), "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *Proc. of the VLDB Endowment*, 2, 922–933.

Afrati, F. and Ullman, J. D. (2009), "Optimizing Joins in a MapReduce Environment," in *Proc. of the 13th Intl. Conf. on Extending Database Technology*, pp. 99–110, ACM.

Agrawal, S., Chaudhuri, S., and Narasayya, V. R. (2000), "Automated Selection of Materialized Views and Indexes in SQL Databases," in *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pp. 496–505, Morgan Kaufmann Publishers Inc.

Agrawal, S., Narasayya, V., and Yang, B. (2004), "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design," in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 359–370, ACM.

Agrawal, S., Chaudhuri, S., Kollar, L., Marathe, A., Narasayya, V., and Syamala, M. (2005), "Database Tuning Advisor for Microsoft SQL Server 2005," in *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 930–932, ACM.

Agrawal, S., Chu, E., and Narasayya, V. (2006), "Automatic Physical Design Tuning: Workload as a Sequence," in *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 683–694, ACM.

Amazon EMR (2012), "*Amazon Elastic MapReduce*," `http://aws.amazon.com/elasticmapreduce`.

Andrei, M. and Valduriez, P. (2001), "User-Optimizer Communication using Abstract Plans in Sybase ASE," in *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pp. 29–38, Morgan Kaufmann Publishers Inc.

Antoshenkov, G. and Ziauddin, M. (1996), "Query Processing and Optimization in Oracle Rdb," *The VLDB Journal*, 5, 229–237.

AsterData (2012), "*Aster Data nCluster*," `http://www.asterdata.com/product/ncluster_cloud.php`.

Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. (1976), "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems (TODS)*, 1, 97–137.

Avnur, R. and Hellerstein, J. M. (2000), "Eddies: Continuously Adaptive Query Processing," in *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 261–272, ACM.

Azkaban (2011), "*Azkaban: Simple Hadoop Workflow*," `http://sna-projects.com/azkaban/`.

Babcock, B. and Chaudhuri, S. (2005), "Towards a Robust Query Optimizer: A Principled and Practical Approach," in *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 119–130, ACM.

Babu, S. (2010), "Towards Automatic Optimization of MapReduce Programs," in *Proc. of the 1st Symposium on Cloud Computing*, pp. 137–142, ACM.

Babu, S., Bizarro, P., and DeWitt, D. (2005), "Proactive Re-Optimization," in *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 107–118, ACM.

Baldwin, C., Eliassi-Rad, T., Abdulla, G., and Critchlow, T. (2003), "The Evolution of a Hierarchical Partitioning Algorithm for Large-Scale Scientific Data: Three Steps of Increasing Complexity," in *Proc. of the 15th Intl. Conf. on Scientific and Statistical Database Management*, pp. 225–228, IEEE.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986), "Magic Sets and Other Strange Ways to Implement Logic Programs," in *Proc. of the 5th ACM Symp. on Principles of Database Systems*, pp. 1–15, ACM.

Baru, C. K., Fecteau, G., Goyal, A., Hsiao, H., Jhingran, A., Padmanabhan, S., Copeland, G. P., and Wilson, W. G. (1995), "DB2 Parallel Edition," *IBM Systems Journal*, 34.

Belknap, P., Dageville, B., Dias, K., and Yagoub, K. (2009), "Self-Tuning for SQL Performance in Oracle Database 11g," in *Proc. of the 25th IEEE Intl. Conf. on Data Engineering*, pp. 1694–1700, IEEE.

Bent, J., Denehy, T. E., Livny, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2009), "Data-Driven Batch Scheduling," in *Proc. of the 2nd Intl. Workshop on Data-Aware Distributed Computing*, pp. 1–10, ACM.

Bizarro, P., Babu, S., DeWitt, D. J., and Widom, J. (2005), "Content-Based Routing: Different Plans for Different Data," in *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pp. 757–768, VLDB Endowment.

Blanas, S., Patel, J. M., Ercegovac, V., Rao, J., Shekita, E. J., and Tian, Y. (2010), "A Comparison of Join Algorithms for Log Processing in MapReduce," in *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 975–986, ACM.

Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., and Patterson, D. (2009), "Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters," in *Proc. of the 1st USENIX Conf. on Hot Topics in Cloud Computing*, USENIX Association.

Bodkin, R. (2010), "*Facebook on Hadoop, Hive, HBase, and A/B Testing*," `http://rbodkin.wordpress.com/2010/07/14/facebook-on-hadoop-hive-hbase-and%C2%A0ab%C2%A0testing/`.

Bruno, N. and Chaudhuri, S. (2005), "Automatic Physical Database Tuning: A Relaxation-Based Approach," in *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 227–238, ACM.

Bruno, N., Chaudhuri, S., and Ramamurthy, R. (2009), "Power Hints for Query Optimization," in *Proc. of the 25th IEEE Intl. Conf. on Data Engineering*, pp. 469–480, IEEE.

BTrace (2012), "*BTrace: A Dynamic Instrumentation Tool for Java*," `http://kenai.com/projects/btrace`.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. (2010), "HaLoop: Efficient Iterative Data Processing on Large Clusters," *Proc. of the VLDB Endowment*, 3, 285–296.

Cafarella, M. J. and Ré, C. (2010), "Manimal: Relational Optimization for Data-Intensive Programs," in *Proc. of the 13th Intl. Workshop on the Web and Databases*, pp. 10:1–10:6, ACM.

Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. (2004), "Dynamic Instrumentation of Production Systems," in *Proc. of the USENIX Annual Technical Conference*, USENIX Association.

Cascading (2011), "*Cascading*," `http://www.cascading.org/`.

Ceri, S. and Gottlob, G. (1986), "Optimizing Joins Between two Partitioned Relations in Distributed Databases," *Journal of Parallel and Distributed Computing*, 3, 183–205.

Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. (2008), "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," *Proc. of the VLDB Endowment*, 1, 1265–1276.

Chaudhuri, S. and Narasayya, V. R. (2007), "Self-Tuning Database Systems: A Decade of Progress," in *Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pp. 3–14, VLDB Endowment.

Chaudhuri, S., Ganesan, P., and Narasayya, V. R. (2003), "Primitives for Workload Summarization and Implications for SQL," in *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, pp. 730–741, VLDB Endowment.

Chaudhuri, S., Narasayya, V., and Ramamurthy, R. (2008), "A Pay-As-You-Go Framework for Query Execution Feedback," in *Proc. of the 34th Intl. Conf. on Very Large Data Bases*, pp. 1141–1152, VLDB Endowment.

Chen, C. M. and Roussopoulos, N. (1994), "Adaptive Selectivity Estimation using Query Feedback," *ACM SIGMOD Record*, 23, 161–172.

Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A., and Krintz, C. (2010), "See Spot Run: Using Spot Instances for MapReduce Workflows," in *Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing*, pp. 7–7, USENIX Association.

Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J. M., and Welton, C. (2009), "MAD Skills: New Analysis Practices for Big Data," *Proc. of the VLDB Endowment*, 2, 1481–1492.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2003), *Introduction to Algorithms*, The MIT Press and McGraw-Hill Osborne Media, 2nd edn.

Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., and Ziauddin, M. (2004), "Automatic SQL Tuning in Oracle 10g," in *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pp. 1098–1109, VLDB Endowment.

Dai, D. (2011), "*PigMix Benchmark*," `https://cwiki.apache.org/confluence/display/PIG/PigMix`.

Dean, J. and Ghemawat, S. (2004), "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. of the 6th Conf. on Operating Systems Design and Implementation*, pp. 137–149, USENIX Association.

Dean, J. and Ghemawat, S. (2008), "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, 51, 107–113.

Deshpande, A., Guestrin, C., Hong, W., and Madden, S. (2005), "Exploiting Correlated Attributes in Acquisitional Query Processing," in *Proc. of the 21st IEEE Intl. Conf. on Data Engineering*, pp. 143–154, IEEE.

Deshpande, A., Ives, Z. G., and Raman, V. (2007), "Adaptive Query Processing," *Foundations and Trends in Databases*, 1, 1–140.

Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. (2010), "Hadoop++: Making a Yellow Elephant Run Like a Cheetah," *Proc. of the VLDB Endowment*, 3, 515–529.

Duan, S., Thummala, V., and Babu, S. (2009), "Tuning Database Configuration Parameters with iTuned," *Proc. of the VLDB Endowment*, 2, 1246–1257.

Friedman, E., Pawlowski, P., and Cieslewicz, J. (2009), "SQL/MapReduce: A Practical Approach to Self-Describing, Polymorphic, and Parallelizable User-Defined Functions," *Proc. of the VLDB Endowment*, 2, 1402–1413.

Gantz, J. and Reinsel, D. (2011), "*Extracting Value from Chaos*," `http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf`.

Gates, A. (2010), *Comparing Pig Latin and SQL for Constructing Data Processing Pipelines*, `http://developer.yahoo.com/blogs/hadoop/posts/2010/01/comparing_pig_latin_and_sql_fo/`.

Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U. (2009), "Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience," *Proc. of the VLDB Endowment*, 2, 1414–1425.

Giakoumakis, L. and Galindo-Legaria, C. (2008), "Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences," *IEEE Data Engineering Bulletin*, 31, 37–44.

Gittins, J. C. and Jones, D. M. (1974), "A Dynamic Allocation Index for the Sequential Design of Experiments," *Progress in Statistics (European Meeting of Statisticians)*, 1, 241.

Goldberg, D. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley.

Graefe, G. and DeWitt, D. J. (1987), "The EXODUS Optimizer Generator," in *Proc. of the 1987 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 160–172, ACM.

Greenplum (2012), "*Greenplum*," `http://www.greenplum.com`.

Gunda, P. K., Ravindranath, L., Thekkath, C. A., Yu, Y., and Zhuang, L. (2010), "Nectar: Automatic Management of Data and Computation in Datacenters," in *Proc. of the 9th Conf. on Operating Systems Design and Implementation*, pp. 1–8, USENIX Association.

Haas, P. J., Ilyas, I. F., Lohman, G. M., and Markl, V. (2009), "Discovering and Exploiting Statistical Properties for Query Optimization in Relational Databases: A Survey," *Statistical Analysis and Data Mining*, 1, 223–250.

Hadoop (2012), "*Apache Hadoop*," `http://hadoop.apache.org/`.

Hadoop Perf UI (2011), "*Hadoop Performance Monitoring UI*," `http://code.google.com/p/hadoop-toolkit/wiki/HadoopPerformanceMonitoring`.

Hadoop Tutorial (2011), "*Hadoop MapReduce Tutorial*," `http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html`.

Hadoop Vaidya (2011), "*Hadoop Vaidya*," `http://hadoop.apache.org/mapreduce/docs/r0.21.0/vaidya.html`.

Hamilton, J. (2008), "*Resource Consumption Shaping*," `http://perspectives.mvdirona.com/2008/12/17/ResourceConsumptionShaping.aspx`.

Herodotou, H. and Babu, S. (2009), "Automated SQL Tuning through Trial and (Sometimes) Error," in *Proc. of the 2nd Intl. Workshop on Testing Database Systems*, pp. 1–6, ACM.

Herodotou, H. and Babu, S. (2010), "Xplus: A SQL-Tuning-Aware Query Optimizer," *Proc. of the VLDB Endowment*, 3, 1149–1160.

Herodotou, H. and Babu, S. (2011), "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs," *Proc. of the VLDB Endowment*, 4, 1111–1122.

Herodotou, H., Dong, F., and Babu, S. (2011a), "MapReduce Programming and Cost-based Optimization? Crossing this Chasm with Starfish," in *Demonstration at the 37th Intl. Conf. on Very Large Data Bases*, VLDB Endowment.

Herodotou, H., Dong, F., and Babu, S. (2011b), "No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics," in *Proc. of the 2nd Symposium on Cloud Computing*, ACM.

Herodotou, H., Borisov, N., and Babu, S. (2011c), "Query Optimization Techniques for Partitioned Tables," in *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 49–60, ACM.

Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. (2011d), "Starfish: A Self-tuning System for Big Data Analytics," in *Proc. of the 5th Biennial Conf. on Innovative Data Systems Research*.

Herodotou, H., Babu, S., Reed, B., and Chen, J. (2012), "Interaction-Aware Optimization of MapReduce Workflows for Improved Cluster Utilization," Tech. Rep. CS-2012-02, Duke Computer Science.

IBM Corp. (2007), *Partitioned tables*, IBM Corporation2, `http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.partition.doc/doc/c0021560.html`.

IBM Corp. (2009), *Configuring DB2 to Use an Optimization Profile*, IBM Corporation, `http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.admin.doc/doc/t0024533.htm`.

IBM Corp. (2010), *DB2 SQL Performance Analyzer*, IBM Corporation, `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2tools.anl.doc.iug/anlhome.htm`.

IBM Corp. (2011a), *Giving Optimization Hints to DB2*, IBM Corporation, `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2.doc.admin/p9li375.htm`.

IBM Corp. (2011b), *IBM DB2 Database for Linux, UNIX, and Windows Information Center*, IBM Corporation, `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp`.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007), "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating Systems Review*, 41, 59–72.

Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. (2009), "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proc. of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pp. 261–276, ACM.

Iu, M.-Y. and Zwaenepoel, W. (2010), "HadoopToSQL: A MapReduce Query Optimizer," in *Proc. of the 5th European Conf. on Computer Systems*, pp. 251–264, ACM.

Jiang, D., Ooi, B. C., Shi, L., and Wu, S. (2010), "The Performance of MapReduce: An In-depth Study," *Proc. of the VLDB Endowment*, 3, 472–483.

Jindal, A., Quian-Ruiz, J.-A., and Dittrich, J. (2011), "Trojan Data Layouts: Right Shoes for a Running Elephant," in *Proc. of the 2nd Symposium on Cloud Computing*, ACM.

Kabra, N. and DeWitt, D. J. (1998), "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans," *ACM SIGMOD Record*, 27, 106–117.

Kambatla, K., Pathak, A., and Pucha, H. (2009), "Towards Optimizing Hadoop Provisioning in the Cloud," in *Proc. of the 1st USENIX Conf. on Hot Topics in Cloud Computing*, USENIX Association.

Karjoth, G. (2003), "Access Control with IBM Tivoli Access Manager," *ACM Transactions on Information and System Security (TISSEC)*, 6, 232–257.

Kemper, A., Moerkotte, G., and Peithner, K. (1993), "A Blackboard Architecture for Query Optimization in Object Bases," in *Proc. of the 19th Intl. Conf. on Very Large Data Bases*, pp. 543–543, Morgan Kaufmann Publishers Inc.

Kreps, J. (2009), "*TeraByte-scale Data Cycle at LinkedIn*," http://tinyurl.com/lukod6.

Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. (2010), "Skew-Resistant Parallel Processing of Feature Extracting Scientific User-Defined Functions," in *Proc. of the 1st Symposium on Cloud Computing*, ACM.

Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., and Zhang, X. (2011), "YSmart: Yet Another SQL-to-MapReduce Translator," in *Proc. of the 31st Intl. Conf. on Distributed Computing Systems*, pp. 25–36, IEEE.

Levy, A. Y., Mumick, I. S., and Sagiv, Y. (1994), "Query Optimization by Predicate Move-Around," in *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pp. 96–107, Morgan Kaufmann Publishers Inc.

Li, A., Yang, X., Kandula, S., and Zhang, M. (2010), "CloudCmp: Shopping for a Cloud Made Easy," in *Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing*, USENIX Association.

Lin, J. and Dyer, C. (2010), *Data-Intensive Text Processing with MapReduce*, Morgan and Claypool.

Lipcon, T. (2009), "*Cloudera: 7 tips for Improving MapReduce Performance*," http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.

Louth, W. (2009), "*OpenCore Vs. BTrace*," http://opencore.jinspired.com/?page_id=588.

Macbeth, S. (2011), "*Why YieldBot Chose Cascalog over Pig for Hadoop Processing*," http://tech.backtype.com/52456836.

Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., and Cilimdzic, M. (2004), "Robust Query Processing through Progressive Optimization," in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 659–670, ACM.

Markl, V., Haas, P. J., Kutsch, M., Megiddo, N., Srivastava, U., and Tran, T. M. (2007), "Consistent Selectivity Estimation via Maximum Entropy," *The VLDB Journal*, 16, 55–76.

Morales, T. (2007), *Oracle Database VLDB and Partitioning Guide 11g Release 1 (11.1)*, Oracle Corporation, `http://docs.oracle.com/cd/B28359_01/server.111/b32024.pdf`.

Neugebauer, C. Z. R., Sutyanyong, N., Qian, X., and Berger, R. (2002), "Partitioning in DB2 Using the UNION ALL View," *IBM DeveloperWorks*, `http://www.ibm.com/developerworks/data/library/techarticle/0202zuzarte/0202zuzarte.pdf`.

Nykiel, T., Potamias, M., Mishra, C., Kollios, G., and Koudas, N. (2010), "MRShare: Sharing Across Multiple Queries in MapReduce," *Proc. of the VLDB Endowment*, 3, 494–505.

Olken, F. and Rotem, D. (1995), "Random Sampling from Databases: A Survey," *Statistics and Computing*, 5, 25–42.

Olston, C., Reed, B., Silberstein, A., and Srivastava, U. (2008a), "Automatic Optimization of Parallel Dataflow Programs," in *Proc. of the 2008 USENIX Annual Technical Conference*, pp. 267–273, USENIX Association.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008b), "Pig Latin: A Not-So-Foreign Language for Data Processing," in *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1099–1110, ACM.

Oozie (2010), "*Oozie: Workflow Engine for Hadoop*," `http://yahoo.github.com/oozie/`.

Ozsu, T. M. and Valduriez, P. (1999), *Principles of Distributed Database Systems*, Prentice Hall.

Papadomanolakis, S. and Ailamaki, A. (2004), "AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning," in *Proc. of the 2004 Intl. Conf. on Scientific and Statistical Database Management*, pp. 383–392, IEEE.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., and Stonebraker, M. (2009), "A Comparison of Approaches to Large-Scale Data Analysis," in *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 165–178, ACM.

Polyzotis, N. (2005), "Selectivity-based Partitioning: A Divide-and-union Paradigm for Effective Query Optimization," in *Proc. of the 14th ACM Intl. Conf. on Information and Knowledge Management*, pp. 720–727, ACM.

Quinlan, R. J. (1992), "Learning with Continuous Classes," in *Proc. of the 5th Australian Joint Conference on Artificial Intelligence*, pp. 343–348, Springer Berlin / Heidelberg.

Qureshi, A., Weber, R., Balakrishnan, H., Guttag, J. V., and Maggs, B. V. (2009), "Cutting the Electric Bill for Internet-scale Systems," in *Proc. of the ACM SIGCOMM Conf. on Data Communication*, pp. 123–134, ACM.

Rao, J., Zhang, C., Megiddo, N., and Lohman, G. M. (2002), "Automating Physical Database Design in a Parallel Database," in *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 558–569, ACM.

Ratzesberger, O. (2010), "*Agile Enterprise Analytics*," SMDB 2010 Keynote by Oliver Ratzesberger.

Romeijn, H. and Smith, R. (1994), "Simulated annealing and adaptive search in global optimization," *Probability in the Engineering and Informational Sciences*.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979), "Access Path Selection in a Relational Database Management System," in *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pp. 23–34, ACM.

Shankar, S. and Dewitt, D. J. (2007), "Data Driven Workflow Planning in Cluster Management Systems," in *Proc. of the 16th Intl. Symposium on High Performance Distributed Computing*, pp. 127–136, ACM.

Sheers, K. R. (1996), "HP OpenView Event Correlation Services," *Hewlett-Packard Journal*, 47, 31–33.

Sood, A. (2010), "*How to dynamically assign reducers to a Hadoop Job at runtime*," http://www.hadoop-blog.com/2010/12/how-to-dynamically-assign-reducers-to.html.

Stillger, M., Lohman, G. M., Markl, V., and Kandil, M. (2001), "LEO - DB2's Learning Optimizer," in *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pp. 19–28, Morgan Kaufmann Publishers Inc.

Talmage, R. (2009), *Partitioned Table and Index Strategies Using SQL Server 2008*, Microsoft, http://msdn.microsoft.com/en-us/library/dd578580.aspx.

Tang, H. (2009), "*Mumak: Map-Reduce Simulator*," https://issues.apache.org/jira/browse/MAPREDUCE-728.

Teradata (2012), "*Teradata*," `http://www.teradata.com`.

Thain, D., Tannenbaum, T., and Livny, M. (2005), "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, 17, 323–356.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009), "Hive: A Warehousing Solution over a Map-Reduce Framework," *Proc. of the VLDB Endowment*, 2, 1626–1629.

TPC (2009), *TPC Benchmark H Standard Specification*, `http://www.tpc.org/tpch/spec/tpch2.9.0.pdf`.

Urhan, T., Franklin, M., and Amsaleg, L. (1998), "Cost-based Query Scrambling for Initial Delays," *ACM SIGMOD Record*, 27, 130–141.

Wang, G., Butt, A., Pandey, P., and Gupta, K. (2009), "A Simulation Approach to Evaluating Design Decisions in MapReduce Setups," in *Proc. of the IEEE Intl. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 1–11, IEEE.

White, T. (2010), *Hadoop: The Definitive Guide*, Yahoo! Press.

Wu, S., Li, F., Mehrotra, S., and Ooi, B. C. (2011), "Query Optimization for Massively Parallel Data Processing," in *Proc. of the 2nd Symposium on Cloud Computing*, ACM.

Ye, T. and Kalyanaraman, S. (2003), "A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration," in *Proc. of the 2003 ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pp. 196–205, ACM.

Yu, Y., Gunda, P. K., and Isard, M. (2009), "Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations," in *Proc. of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pp. 247–260, ACM.

Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010), "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proc. of the 5th European Conf. on Computer Systems*, pp. 265–278, ACM.

Zeller, B. and Kemper, A. (2002), "Experience Report: Exploiting Advanced Database Optimization Features for Large-Scale SAP R/3 Installations," in *Proc. of the 28st Intl. Conf. on Very Large Data Bases*, pp. 894–905, VLDB Endowment.

Zheng, W., Bianchini, R., Janakiraman, J., Santos, J. R., and Turner, Y. (2009), "JustRunIt: Experiment-Based Management of Virtualized Data Centers," in *Proc. of the 2009 USENIX Annual Technical Conference*, pp. 18–18, USENIX Association.

Zhou, J., Larson, P.-Å., and Chaiken, R. (2010), "Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer," in *Proc. of the 26th IEEE Intl. Conf. on Data Engineering*, pp. 1060–1071, IEEE.

Zilio, D., Jhingran, A., and Padmanabhan, S. (1994), "Partitioning Key Selection for a Shared-Nothing Parallel Database System," *IBM Research Report RC 19820*.

Zilio, D. C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., and Fadden, S. (2004), "DB2 Design Advisor: Integrated Automatic Physical Database Design," in *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pp. 1087–1097, VLDB Endowment.

Biography

Herodotos Herodotou was born on September 5th, 1983 in Nicosia, Cyprus. He received his Ph.D. degree in Computer Science from Duke University in May 2012. His dissertation work focused on the ease-of-use, manageability, and automated tuning of both centralized and distributed data-intensive computing systems. He received his M.S. degree in Computer Science from Duke University in May 2009. His M.S. thesis focused on automating the process of SQL tuning in an efficient manner using an experiment-driven approach. His research work resulted in two public software releases for (i) Xplus, a SQL-tuning-aware query optimizer, and (ii) Starfish, a self-tuning system for Big Data analytics.

While at Duke, he was a recipient of the Steele Endowed Fellowship in 2008. In addition, he has had extensive industry exposure as both a researcher and a software engineer through multiple internships, with esteemed companies like Yahoo! Research, Microsoft Corporation, and Aster Data.

Before joining Duke, he completed his undergraduate studies as a double major in Computer Science and Mathematics at the University of Maryland, Baltimore County (UMBC). He was the recipient of the Cyprus-America Scholarship (awarded by the Cyprus Fulbright Commission) and the UMBC President's Scholar Award.

Upon completion of his graduate studies at Duke University, Herodotos will join the eXtreme Computing Group (XCG) at Microsoft Research, where he will engage in research related to cloud computing and large-scale data analytics.