UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PHYSICAL DESIGN TUNING METHODS FOR EMERGING SYSTEM ARCHITECTURES

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Jeff LeFevre

June 2014

The Dissertation of Jeff LeFevre
is approved:

Neoklis Polyzotis, Chair

Hakan Hacıgümüş

Phokion Kolaitis

Carlos Maltzahn

Wang-Chiew Tan

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures vi

List of Tables ix

Abstract x

Dedication xii

Acknowledgments xiii

1 Introduction 1
- 1.1 Traditional Physical Design Tuning . . . 2
- 1.2 New and Emerging System Architectures . . . 4
- 1.3 Contributions of this Dissertation . . . 6
  - 1.3.1 Divergent Physical Design for Replicated Databases . . . 6
  - 1.3.2 Opportunistic Physical Design for Large-scale Data Analytics . . . 7
  - 1.3.3 Multistore Physical Design for Hybrid Systems . . . 9
  - 1.3.4 Towards a Workload for Evolutionary Analytics . . . 10
- 1.4 Dissertation Outline . . . 10

2 Background 12
- 2.1 Query-Cost Estimation and Workload Modeling . . . 12
- 2.2 The Physical Design Tuning Problem . . . 15
- 2.3 Addressing the Physical Design Tuning Problem for New Architectures . . . 18
  - 2.3.1 Replicated Database Architecture . . . 19
  - 2.3.2 Hadoop Architecture . . . 20
  - 2.3.3 Multistore Architecture . . . 21

3 Divergent Physical Design Tuning for Replicated Databases 22
- 3.1 Divergent Design Tuning: Problem Statement . . . 26
- 3.2 Problem Analysis . . . 33
  - 3.2.1 Theoretical Analysis . . . 34

3.2.2 Empirical Analysis . . . 44
3.3 The DivgDesign Algorithm for Divergent Design Tuning . . . 47
3.3.1 Overview of Our Approach . . . 47
3.3.2 Algorithm Definition . . . 50
3.4 Experimental Study . . . 56
3.4.1 Methodology . . . 57
3.4.2 Behavior of the DivgDesign Algorithm . . . 60
3.4.3 Performance for Query-Only Workloads . . . 62
3.4.4 Results with Mixed Workloads . . . 67
3.4.5 Performance with Concurrent Execution . . . 70
3.5 Implementation Considerations . . . 73
3.6 Related Work . . . 74

4 Opportunistic Physical Design for Big Data Analytics 78
4.1 Preliminaries . . . 81
4.1.1 System Architecture . . . 82
4.1.2 Notations . . . 83
4.1.3 Problem Definition . . . 84
4.2 UDF Model . . . 85
4.2.1 Modeling a UDF . . . 86
4.2.2 Applying the UDF Model and Annotations . . . 89
4.3 Using the UDF Model to Perform Rewrites . . . 92
4.3.1 Equivalence Testing . . . 92
4.3.2 Costing a UDF . . . 94
4.3.3 Lower-bound on Cost of a Potential Rewrite . . . 97
4.4 Problem Overview for Rewriting Queries Containing UDFs . . . 100
4.5 Best-First Rewrite . . . 104
4.5.1 The BfRewrite Algorithm . . . 105
4.5.2 Proof of Correctness and Work-Efficiency . . . 108
4.6 ViewFinder . . . 111
4.6.1 The ViewFinder Algorithm . . . 112
4.6.2 Rewrite Enumeration . . . 114
4.7 Experimental Evaluation . . . 115
4.7.1 Query Workload . . . 117
4.7.2 Experimental Methodology . . . 119
4.7.3 Experimental Results . . . 121
4.7.4 Calibration of Cost Model . . . 135
4.7.5 Comparison with DBMS-X . . . 137
4.7.6 Microbenchmark . . . 139
4.8 Related Work . . . 147

5 MISO: Souping Up Big Data Query Processing with a Multistore System 150
5.1 System Architecture and Multistore Tuning . . . 155
5.1.1 System Components . . . 156
5.1.2 Overview of Multistore Tuning . . . 161
5.2 MISO Tuner . . . 165
5.2.1 Basic Definitions and Problem Statement . . . 166
5.2.2 MISO Tuner Algorithm . . . 169
5.2.3 Handling View Interactions . . . 171
5.2.4 MISO Knapsack Packing . . . 174
5.3 Experimental Study . . . 178
5.3.1 Methodology . . . 179
5.3.2 Main Results . . . 183
5.3.3 Tuning Algorithm Comparisons . . . 189
5.3.4 Utilizing DW with Limited Spare Capacity . . . 194
5.3.5 Comparison with non-Hadoop approaches . . . 199
5.4 Discussion . . . 205
5.5 Related Work . . . 207

6 Towards a Workload for Evolutionary Analytics 212
6.1 Workload characteristics and system metrics . . . 216
6.1.1 Workload Characteristics . . . 216
6.1.2 System Metrics . . . 217
6.2 The Workload . . . 220
6.2.1 Query building blocks . . . 220
6.2.2 Queries . . . 222
6.3 Running the Benchmark . . . 225
6.3.1 Benchmark methodology . . . 225
6.3.2 Example benchmark results . . . 226
6.4 Discussion . . . 228
6.5 Workload Details . . . 229
6.5.1 Datasets . . . 230
6.5.2 User Defined Functions referenced by queries . . . 230
6.5.3 Query Descriptions . . . 232

7 Conclusion and Future Work 247

Bibliography 251

List of Figures

3.1 An illustration of the trade-off between load balancing and performance. 30
3.2 An example of running the DivgDesign algorithm for $m = 1$, $n = 2$ and a workload of three queries: (a) Initial design, (b) Query costs under the corresponding configurations (minimum costs are indicated with bold); (c) Refined designs. 49
3.3 Five independent runs of DivgDesign. 61
3.4 Performance improvement of divergent design with $m = 1$ for (a) TPC-H query-only workload and (b) TPC-DS query-only workload. 63
3.5 Performance of divergent design and uniform design for TPC-DS query-only workload, $m = 1$, $n = 4$. 64
3.6 Performance of divergent designs for query-only workloads (a) TPC-H with $n = 4$, $m = 2$, (b) TPC-DS with $n = 4$, $m = 2$. 66
3.7 (a) Performance of divergent designs for TPC-H with updates and $m = 1$. (b) Percent improvement breakdown for queries and updates in TPC-H-u110, $n = 4$, $m = 1$. 68

4.1 System diagram showing control flows. 83
4.2 A UDF composed of local functions (lf_1, lf_2, $\cdots$, lf_k), showing the end-to-end transformation of input to output. 86
4.3 UDF_FOODIES a) implementation composed of two local functions, b) UDF model showing the end-to-end transformation of input to output. . 88
4.4 (a) Example query to obtain prolific foodies, and (b) corresponding annotated query plan. 90
4.5 Synthesized UDF to perform the fix between a view v and a query q. . . 98
4.6 High level overview of the BfRewrite algorithm. 102
4.7 Query Evolution comparisons for query execution time (log-scale) 122
4.8 Query Evolution comparisons for execution time improvement. 122
4.9 User Evolution comparisons (log-scale) for (a) execution time , and (b) data moved. 123
4.10 Execution time improvement for rewr corresponding to Figure 4.9(a). . 124

4.11 Algorithm comparisons (log-scale) for (a) candidate views considered, (b) rewrite attempts. . . . 126
4.12 Comparison of algorithm runtime (log-scale). . . . 126
4.13 Runtime of BFR and DP for varying number of views. . . . 128
4.14 Execution time analysis of the quality of BFR's rewrite solutions found during its search for the optimal rewrite. . . . 128
4.15 Execution time improvement over ORIG. . . . 132
4.16 MR job execution time vs.estimated cost using our cost model (log-scale). Each point represents an MR job in the 32 query workload. . . . 136
4.17 Dimensions of change (PLUG) between query versions as defined in [61] 142
4.18 Illustration of the microbenchmark queries showing the changes from one version to another. The left-most column shows the original version of each query, the next columns show the revised versions with additions in gray and removals with strike-through. . . . 143
4.19 Microbenchmark results showing original execution time (ORIG) and rewritten query execution time of BfRewrite (BFR) for queries revised along each of the P, L, U, G dimensions. . . . 145

5.1 Typical setup in today's organizations showing (a) two independent stores and (b) multistore system utilizing both stores. . . . 151
5.2 Multistore System Architecture. . . . 156
5.3 A plan with several possible split points. . . . 158
5.4 Execution time profile of all multistore plans for a single query. Plans are ordered by increasing execution time. . . . 162
5.5 Two query workload on several system variants. . . . 165
5.6 Performance results for 5 system variants. . . . 183
5.7 CDF plot showing the progressive TTI (y-axis) as each of the 32 queries in the workload is completed (x-axis). . . . 186
5.8 CDF plot showing the percentage of queries (y-axis) below a given execution time (x-axis). . . . 186
5.9 Breakdown of execution time into DW-EXE, HV-EXE and Transfer time components for queries from Figure 5.7(b); (a) MS-BASIC, (b) MS-MISO for 0.125× storage budget, and (c) MS-MISO for 2× storage budget. . . 189
5.10 TTI Comparison of multistore tuning techniques. . . . 190
5.11 Varying the view storage budgets B_h and B_d, while transfer budget B_t is held constant at 10GB. . . . 191
5.12 TTI for MS-MISO while varying the B_t parameter . . . 192
5.13 TTI for MS-MISO while varying the B_t parameter and reorganization frequency (R). . . . 193
5.14 Impact of multistore workload on DW with 40% spare IO capacity showing (a) IO/CPU resources consumed and (b) average query execution time of the DW queries. . . . 196

5.15 Impact of multistore workload on DW with 20% spare IO capacity showing (a) IO/CPU resources consumed and (b) average query execution time of the DW queries. 196
5.16 Our Pentaho/Kettle ETL flow for loading a JSON log into DW. 201
5.17 Execution time comparison of Hive and Impala for a single query with varying data sizes. 204

6.1 System metrics for evolutionary analytics. 215
6.2 Dimensions of change for query revisions 221
6.3 Sample benchmark reporting of 4 data systems on a user evolution scenario. 227

List of Tables

3.1 Distribution statistics for divergent design costs in the exhaustive experiment. For designs in each cost percentile, we show the ratio of *TotalCost* to the optimal design cost. 46
3.2 Summary of experimental parameters. 57
3.3 Number of distinct indexes in the configurations of p_{unif} and p_{divg}. . . . 63
3.4 Index counts of the ORDERS and LINEITEM tables per replica for p_{unif} and p_{divg} for TPC-H-updates-110, $n = 4$, $m = 1$, $b = 1.0$. 69
3.5 Total execution time improvement of divergent design over uniform design for 17 TPC-H and 20 TPC-DS Queries, $n = 3, m = 2$, $b = 1$, for 15, 30, and 45 concurrent clients. 71

4.1 Improvement in execution time of A_5v_3 as more analysts become present in the system. 125
4.2 Execution time improvement for each analyst's first query (e.g. A_iv_1) after (1) executing all other analysts' queries and (2) discarding from the system all views identical to those required by the first query (i.e., the holdout query). 131
4.3 Execution Time Improvement after reclaiming 10% to 90% of the view storage space. 133

5.1 MISO Tuner variables. 169
5.2 Impact of multistore workload on DW queries and vice-versa. 198

6.1 Business marketing scenarios for 8 analysts, along with the dimensions modified during each of the 4 evolutions. 223

Abstract

Physical Design Tuning Methods for Emerging System Architectures

by

Jeff LeFevre

Physical design tuning (i.e., configuring the physical data model and secondary data structures) is key to obtaining good performance for a database management system. Recently, there have been disruptions to the types of data, queries, and systems used for analytical data management and processing. The advent of the cloud and database-as-a-service along with these recent disruptions have changed the analytical landscape significantly. Physical design tuning methods that were developed to address traditional RDBMS architectures are inadequate for the emerging system architectures and analytics. While tuning the physical design remains crucial for good performance, the problem acquires interesting dimensions in these new contexts.

This dissertation introduces new physical design tuning methods for emerging system architectures. Our methods exploit the unique characteristics of several different architectures, leveraging them toward good physical design. For replicated database architectures, we introduce the concept of divergent design that exploits database replication. Our method specializes the design of replicas to efficiently process subsets of the workload while still affording the opportunity to load balance the workload across replicas. For MapReduce architectures, we introduce the concept of opportunistic design that exploits the by-products of query processing in MapReduce. Our method

includes a semantic model for user-defined functions (UDFs) and a novel query rewriting algorithm that together enable the effective reuse of previous results. For hybrid MapReduce–RDBMS architectures, we introduce the concept of multistore design that exploits the unique strengths of both stores. Our method periodically reorganizes the data in each store by transferring data between them, adapting their physical designs as the workload changes dynamically. Lastly, based on our experiences with the changing analytical landscape, we examine big data exploratory queries in current real-world scenarios and present a workload modeled upon our findings. We then present a benchmark along with a set of system performance metrics that highlight the various design tradeoffs made by different architectures. By better understanding these tradeoffs, the benchmark can be used to help guide the design of future hybrid architectures.

For Angie.

Acknowledgments

I have been very fortunate to work with and learn from many great people while developing this research. First and foremost, I would like to express my gratitude to my advisor Neoklis Polyzotis for all of his help, guidance, and support throughout my time here at UC Santa Cruz. Next I would like to thank all of my terrific co-authors, who also helped make this work possible and have been great mentors as well: Michael J. Carey, Mariano P. Consens, Hakan Hacıgümüş, Junichi Tatemura; and especially Jagan Sankaranarayanan and Kleoni Ioannidou. Thanks also to my other committee members: Wang-Chiew Tan, Phokion Kolaitis, and Carlos Maltzahn for their guidance and contributions. Thank you to Jeffrey F. Naughton and Hector Garcia-Molina who also provided helpful comments on some of this work. Furthermore, I would also like to thank the folks at NEC Labs for their support. Thanks to all of my many colleagues at UCSC in the Database Group and Systems Group who have been great to work with, in particular Ivo Jimenez, Quoc Trung Tran, Joe Buck, and Noah Watkins; and thanks also to Lynne Sheehan at UCSC for always helping me with my many computing needs. There are many others who have helped me along this journey that I would like to thank as well, including Walt Burkhard, Darrell Long, Grant Grundler, Radha Ramachandran, Leslie Leland, and Craig Blevins. Last but not least I would like to thank my family for all of their love and support, including Olivia Smith, Cole LeFevre, and Evan LeFevre for inspiring me to try harder; and most importantly Angie Smith, who is always there for me and for us – I could not have done this without you.

Chapter 1

Introduction

A wide variety of choices for data management and processing have recently become available with the advent of the cloud and database-as-a-service. New system architectures have emerged, affording users many options beyond traditional relational database management systems (RDBMS). These new architectures can differ significantly from a traditional stand-alone RDBMS in the way they store and process data. The ability to rent *multiple* data management systems as well as multiple distinct *types* of systems on demand in the cloud creates a rich landscape of possibilities.

For these new architectures, database physical design (i.e., configuring the physical data model and secondary data structures) remains key to obtaining good performance. Secondary data structures such as indexes and materialized views can reduce the amount of I/O and CPU resources consumed during query processing. This impacts query execution time, and a well-tuned design can significantly improve performance.

Database physical design tuning has a long history of research for traditional

RDBMSs. However, given the unique characteristics of these new architectures, the physical design problem acquires new and interesting dimensions. Hence, traditional methods are inadequate and there is a need for new techniques. The goal of this dissertation is to introduce new physical design tuning methods for emerging system architectures.

In the following sections we give a brief overview of traditional methods for physical design, describe the salient features of emerging system architectures, then introduce our new methods that form the core of this dissertation.

1.1 Traditional Physical Design Tuning

The physical design tuning problem focuses on identifying a set of indexes or materialized views that will reduce the cost to process a given workload. While indexes and materialized views can provide significant benefit for query processing, they also incur system overhead in terms of storage space, creation cost, and maintenance cost. Storage space refers to the amount of disk space consumed by the indexes and views. Creation cost refers to the cost for materializing the data structures, i.e., building the indexes or views. Maintenance cost refers to the cost of updating the relevant indexes and views whenever an update is made to the base data. Thus, the physical design tuning problem requires cost-benefit considerations, leading to a large space of possible designs to consider.

There are two general approaches toward physical design tuning and a key

distinction between them is the choice of a representative workload for which to tune. *Offline tuning* techniques are applicable when the workload is known, e.g., a batch of reporting queries executed at the end of each sales day. Because the workload is fixed, the system is tuned once. *Online tuning* techniques are applicable when the workload is dynamic and cannot be known in advance, e.g., queries issued by a data analyst looking for new patterns in sales data. Because the workload is changing, the workload is monitored and the system is retuned periodically.

There has been much previous work on offline tuning methods [14,16,22,34,102] while more recent work has focused on online tuning methods [15, 66, 83, 84]. Some of this work has been incorporated into commercial database management systems in the form of a *design advisor* tool that automates the tuning process [25, 33, 96]. All of the previous methods have addressed physical design tuning for a traditional, monolithic RDBMS where a single solution is computed and deployed to a centralized data management system. However, new architectures are not restricted to such an environment, necessitating new approaches toward physical design. Additionally, the types of workloads considered by these methods have only included relational queries such as those represented by standard benchmarks (e.g., TPC-H and TPC-DS [95]) for analytical workloads on relational databases containing highly structured data. In contrast, the ever-increasing data from new non-traditional sources such as social media, sensors, and web logs has low-structure and is often referred to as "big data". This has prompted new types of data management systems for storing and processing this data. Indeed, many organizations now maintain multiple distinct data management systems for this

reason.

1.2 New and Emerging System Architectures

With the cloud, database-as-a-service (DaaS) enables users to choose their systems on-demand, thus users can potentially avoid an up-front decision of a single, fixed data management system. These options provide much flexibility in system architectures but also introduce unique problems for database physical design. For instance, one common option is the ability to rent additional systems as needed. This enables users to scale-out their data processing systems, for example during periods of increased demand. Full database replication is easily implemented in the cloud: multiple database replicas can be instantly configured and launched from an existing system. With replication, a workload can be load-balanced across multiple replicas, leading to better performance than a single system. Improved query processing speeds as well as easy scalability on-demand make database replication an attractive feature of the cloud.

Another option commonly available for data management in the cloud is the MapReduce [35] framework for large-scale data analytics. Recently, MapReduce, or its open-source implementation Hadoop [9], has become the de-facto standard for processing "big data". One of Hadoop's desirable properties is its high fault-tolerance due to n-way data replication over distributed nodes. This property allows Hadoop to scale easily, and since it can run on inexpensive commodity hardware, Hadoop is an attractive option for cloud-based big data processing.

Hadoop systems differ significantly from RDBMSs not only in their architecture and type of data they store but also the way data is processed. While Hadoop natively executes Java code, data analysts often write their big data queries in higher-level query languages such as Pig [73] and Hive [93] that are supported by the Hadoop framework. Furthermore, big data queries often include domain-specific processing tasks (e.g, statistical methods such as classifiers) in the form of user-defined functions (UDFs) which contain arbitrary user-code. Since the business value of this data is often unknown, data analysts may issue a sequence of related complex queries as they explore the data in order to find value. In these ways, a workload of big data analysis queries can differ significantly from a workload of relational reporting queries (e.g., TPC-DS).

One reason Hadoop has become increasingly popular is that big data fits more naturally in a Hadoop system since it only requires copying the data directly into the distributed file system (i.e., HDFS) as flat files. In contrast, an RDBMS requires an extract-transform-load (ETL) process to store the data in highly structured form. This process can be expensive and time consuming, particularly with a large volume of big data. Increasingly, organizations maintain two separate data management systems: a Hadoop processing system for their big data and an RDBMS system (e.g., data warehouse) for their highly-structured organizational data.

Furthermore, an RDBMS and Hadoop system can be combined into a hybrid or *multistore* system whereby a query can be co-processed by both systems. This is done by enabling a query to access data and functionality in either store, migrating data and computation as needed between the stores during query processing. Such emerging

multistore systems represent a natural evolution for big data analytics, and the cloud easily enables users to choose the system type best-suited to their data processing needs as well as combine systems for workload co-processing.

1.3 Contributions of this Dissertation

Next we introduce our new physical design methods for these newly emerging architectures and analytics. Our specific contributions in each area are as follows.

1.3.1 Divergent Physical Design for Replicated Databases

Our first contribution is the concept of *divergent physical design* for replicated databases. With a replicated database architecture, a single system is replicated into multiple stand-alone systems, each with a full copy of the base data such that a query can be answered by any replica. At run time, the workload is load-balanced across the replicas to increase performance. Previous work has not considered replicated scenarios, and our work on divergent design introduces a new paradigm for database systems that employ replication. We exploit database replication, tuning the physical design of replicas such that the design of each replica is specialized to process a subset of the query workload. In contrast, the current state of the practice is to apply single-system tuning techniques to design one replica for the full workload, and then copy the resulting design to all replicas. This results in an inefficient use of the total storage space (i.e., disk space on each replica) by not considering the interplay of replication, missing the opportunity to specialize designs in order to increase performance. Moreover, the current practice

can also result in inefficient designs for the replicas since each replica's design is identical and based upon the full workload — yet at run-time the workload will be distributed across the replicas.

In our work, we formalize the divergent design problem, analyze its complexity, and characterize the properties of good designs. Our problem formulation captures the trade-off between load balancing among all n replicas vs. $m \leq n$ specialized replicas, where a tension occurs due to replica specialization. Furthermore, we show that identical replica designs can be sub-optimal since their designs reflected a workload that is not actually observed at run time. We present a heuristic algorithm to compute effective designs along the trade-off spectrum, and provide an experimental study showing divergent designs can significantly improve workload execution time for analytical workloads on replicated databases. Our concept of divergent design is presented in Chapter 3.

1.3.2 Opportunistic Physical Design for Large-scale Data Analytics

Our second contribution is the concept of *opportunistic physical design* for big data systems. A Hadoop architecture aggressively materializes intermediate results in order to support fault-tolerance. We exploit this query processing behaviour of Hadoop by retaining these results and treating them as materialized views forming what we term an opportunistic physical design, which we then use for query optimization by rewriting queries using the available views. Because UDFs are common with big data queries, it is likely that opportunistic views will include computations performed by UDFs. While there has been a long line of research in rewriting queries using views [41,

46, 58, 59, 64, 79, 101] previous methods have only considered relational operators and have not addressed queries that contain UDFs. This creates a challenge for physical design tuning methods that have previously only considered relational queries. Several recent efforts [37, 72] have introduced methods to reuse computation in Hadoop systems by caching and sharing intermediate results. However, because current approaches lack a semantic understanding of UDFs, these methods are limited to syntactic-matching only, which can restrict the opportunities for reuse significantly.

In our work, we first present a semantic model that captures an important class of MapReduce UDFs, allowing us to reason about views that contain UDFs. Our opportunistic design approach has low overhead since the design is obtained simply as a by-product of query processing. However, both of these factors complicate the physical design problem since there can be a large number of views retained in the system due to the policy of retaining these materializations and there can also be many UDFs added to the system, creating a large space of possible rewrites. To address this, we introduce a novel query rewriting algorithm along with a method that allows us to quickly compute a lower bound on the cost of a potential rewrite given a query and a view, which we use to incrementally explore the large space of possible rewrites. Our algorithm provably finds the minimum cost rewrite under certain assumptions. We evaluate our algorithm with an experimental study showing the algorithm's performance and the resulting improvement in query execution time for a workload of big data queries. Our concept of opportunistic design is presented in Chapter 4.

1.3.3 Multistore Physical Design for Hybrid Systems

Our third contribution is the concept of *multistore physical design* for hybrid RDBMS and Hadoop systems. We exploit this type of system, where data and computation are migrated between the stores during query processing, by dynamically tuning the physical design of both stores through actively transferring data between them in the form of materialized views. Online physical design tuning has been well-studied, but previous works have not considered the multistore setting. Recent work [36, 89] has studied single-query optimization for a multistore system but has not considered physical design tuning or a workload of queries. Other work [2] improves access to base data by piggybacking RDBMS loading onto Hadoop query processing when the systems are co-located on the same node, but does not explicitly consider physical design tuning.

In our work, we tune the physical design of the stores together in an online fashion as the workload changes dynamically. By utilizing opportunistic materialized views and tuning the physical design of both stores, our approach enables sharing computation across queries and stores instead of only improving access to base data. We present a heuristic method to tune both stores that observes constraints on the physical design storage budgets and the maximum amount of data transferred between the stores. We then provide an experimental study showing that our approach can significantly improve performance for big data workloads, even with relatively small view storage and transfer budget size constraints. Our concept of multistore design is presented in Chapter 5.

1.3.4 Towards a Workload for Evolutionary Analytics

Our final contribution is a characterization of big data analytics along with a proposed workload and benchmark. We first describe the way in which the queries, the users, and the data all evolve in big data analytics, a concept we term *evolutionary analytics*. We examine real-world big data queries and present a workload of analytical queries modeled upon our findings. For each query in the workload, we provide several versions, representing the query's revisions during data exploration. We then identify several relevant system performance metrics in this analytical space: physical design storage overhead, physical design tuning time, data-arrival-to-query time, workload execution time, and monetary cost. We describe the tradeoffs among these metrics, and propose a benchmarking methodology to quantify the tradeoffs. By evaluating the tradeoffs, this benchmark can be used to help guide the design of future hybrid systems. Using our benchmark, we then give a brief comparison of the performance of several system architectures in order to highlight the various tradeoffs made by each system. Our concept of evolutionary analytics is presented in Chapter 6.

1.4 Dissertation Outline

The organization of this dissertation is as follows. Chapter 2 provides a background and some of our notation along with the general problem formulation for physical design tuning. Chapter 3 presents our work on divergent physical design for replicated databases, which applies to RDBMSs. A version of this work appears in SIGMOD

2012 [32]. Chapter 4 presents our work on opportunistic physical design for big data analytics systems, such as MapReduce/Hadoop. A version of this work appears in SIGMOD 2014 [63]. Chapter 5 presents our work on multistore physical design for hybrid systems which combine RDBMS and Hadoop for co-processing queries. A version of this work appears in SIGMOD 2014 [62]. Chapter 6 presents our workload and benchmark for evolutionary analytics, based on our experiences with physical design tuning for RDBMS, Hadoop, and multistore systems. A version of this work appears in the 2^{nd} SIGMOD Workshop on Data Analytics in the Cloud 2013 [61]. Chapter 7 provides a conclusion along with some directions for future work.

Chapter 2

Background

In this chapter we introduce introduce some of our notation and provide a brief background for the physical design tuning problem.

2.1 Query-Cost Estimation and Workload Modeling

Database query optimizers estimate the execution cost of a query using an internal cost model, typically based upon on expected CPU, I/O, and network resource consumption. Typically, these internal units do not directly correspond to execution time, rather they are only used by the optimizer to compare relative costs among alternative execution plans. In order to obtain reasonable cost estimates, the optimizer collects and maintains statistics about the data such as the number of rows in each table, the average size of each tuple, and distribution statistics on the data values for each column (e.g., histograms). This metadata is retained in the system catalog and may be updated periodically.

Given a statement q and a physical design D, we use $cost(q, D)$ to denote to the best (i.e., lowest-cost) execution plan for q found by the query optimizer that considers only design D. We note that in the literature, the design is often referred to as a "configuration". In our work we will sometimes use I or v in place of D, when the design includes only indexes or views respectively. In a modern RDBMS, we can utilize its query optimizer to obtain expected costs through the "what-if" optimizer interface [26]. This interface invokes the RDBMS optimizer to compute a cost estimate for a statement given a *hypothetical* design D, allowing us to reason about the expected performance of different designs under consideration. This is convenient because cost estimates can be obtained without actually materializing the design (i.e., building the indexes and materialized views).

To tune a database physical design, a representative workload is required. We use W to denote the representative workload, where W may contain both query and update statements. Each statement in W may have an associated weight or frequency, indicating its relative importance in the workload. We define total workload cost as the sum of the cost to process all statements in W, given as

$$\sum_{q \in W} cost(q, D).$$

This metric represents the total units of work performed by the system to evaluate W as if executing each statement independently. Our workload cost formulation does not consider concurrent execution of statements or run-time system effects such as caching

or buffering that may impact query execution time. Our metric is commonly used for physical design tuning, and allows us to reason concretely among potential design choices. In each chapter, we will expand this metric where relevant for the specific version of the physical design tuning problem being formulated.

The workload may be fixed or changing dynamically, and as noted in Chapter 1.1, physical design tuning approaches fall into two categories depending on the workload scenario. For an offline tuning scenario, W is provided and the design is computed and installed before executing W, possibly during a period of low system utilization. The cost to materialize the design (i.e., creation cost) is not explicitly included in the offline formulation of the offline physical design problem as creating the design is assumed to be a one-time effort. Hence the goal of offline physical design tuning is to minimize the cost to process W.

In contrast, for an online tuning scenario the system observes a continuous stream of statements $\langle q_1, q_2, q_3 \cdots \rangle$ from which W is drawn. Typically, the most recent n statements are used under the assumption that the future workload will be similar to the recent workload, thus the physical design always reflects the recent workload history. Tuning is done periodically; perhaps after a specified number of statements or time intervals, and is installed during (or interleaved with) query processing. The cost to materialize the design is explicitly included in the online formulation of the physical design problem. Hence the goal of online physical design tuning is to minimize the cost to process W as well as the cost to transition between designs.

2.2 The Physical Design Tuning Problem

The general physical design tuning problem is to identify a physical configuration that reduces workload processing cost. The design may include indexes, materialized views, or physical data layout strategies (e.g., horizontal table partitioning). In this dissertation we focus on indexes and materialized views, and do not address physical data layout. The input to the problem is the representative workload W and a set of constraints, and the output is a design (e.g., a set of indexes and views) that minimizes the workload cost metric under the given constraints. A commonly used constraint is a disk space budget, denoted as b, representing the maximum allowable storage space for the data structures in the design solution.

There are many possible candidate indexes and views to consider for the physical design. For instance, one could consider all single-column indexes, all multi-column indexes, in all possible column orders for every relation. Making matters worse, the set of all possible views is a superset of all possible indexes, as views may be defined over multiple relations.

To obtain the expected benefit of a given set of candidate indexes or views, we use the previously described what-if optimizer functionality. Informally, the benefit for a set of indexes (or views) is measured as the change in cost for executing a statement with and without the set. Selecting a subset of all possible candidates that provides the most benefit requires searching a space of designs that is exponential in the number of candidates. When the elements of the design are restricted to include only indexes, this

is known as the *index selection problem*, a special case of the general physical design problem.

The index selection problem is significantly complicated by the fact that the elements of the design can interact with one another, with respect to their benefit for a given query. This means that the benefit of a given index is dependent upon the other indexes present, hence index benefits cannot be calculated independently. For instance, consider an index I_x that provides some benefit for a query q, and another index I_y that can be used in place of I_x and provides slightly more benefit for q. In this instance, if I_y is present the optimizer will prefer to use I_y for q instead of I_x, rendering I_x useless. Then if two designs $D = \{I_x\}$ and $D' = \{I_x, I_y\}$ are under consideration, notice that the benefit of I_x for q is not the same in both designs — in D the benefit of I_x is positive and non-zero, while in D' where I_y is present, the benefit of I_x drops to zero. This is the reason that all possible subsets of candidates must be considered in order to identify the most beneficial set. A formal model and analysis of index interactions is given in [85], while earlier work such as [15] accounts for interactions using a heuristic course-grained approach.

Due to interactions, the index selection problem with a storage constraint has been shown to be NP-hard as well as hard to approximate [23] (via reduction from the *k-densest subgraph* problem). Since the problem is computationally hard and there is a large space of possible designs for any realistic problem size, heuristics are used to prune the search space.

The problem is typically solved in two phases. First, a set of *relevant* candi-

dates (indexes/views) are identified based on the representative workload. Pruning is typically done here by restricting these to include only a subset of all possible candidates that could be defined. For instance, by considering only single-column indexes [83] or by "merging" candidates [6, 25, 27] in order to reduce redundancy and hence conserve disk space.

Next, the process of selecting a beneficial subset from the full candidate set can begin, using additional pruning heuristics. Some approaches [96] use a variant of the 0-1 knapsack problem, where the knapsack *value* of an index is its benefit and the *weight* of an index is its storage size. After knapsack packing in [96], small random permutations are applied to the solution in an effort to possibly identify a better solution. Others [25] take a greedy approach to traverse the space, exhaustively searching a small subspace to find the best set and then greedily growing that set. These two methods represent bottom-up approaches, growing the solution from an empty initial set. Other work [14] takes a top-down approach, starting with a very beneficial but too-large set of candidates then removing elements until the set fits within the storage budget constraint.

Recent work [84] proposes a divide-and-conquer approach that partitions the full candidate set into non-overlapping subsets, where indexes in one subset do not interact with indexes in another subset (or interact very weakly). With partitioning, the isolation of indexes into non-interacting sets reduces the number of interactions that must be considered. This significantly reduces the size of the search space from exponential in the cardinality of the full candidate set to exponential in the cardinality of the (likely much smaller) partitioned subsets. Furthermore, because invoking the

what-if optimizer to evaluate the benefit of each potential design has a non-zero cost, some approaches have proposed ways to mitigate this overhead by reducing the number of calls to the what-if optimizer [25, 83] or computing the benefit for a set of indexes using integer linear programming [34, 74].

Similar to the index selection problem, choosing a good set of views is the *view selection problem*, which is also known to be computationally hard [30, 53]. Whereas indexes are restricted to projections over a single relation and may contain selection predicates, views can include additional predicates such as join or group-by, allowing views to be defined over multiple relations with more complex operations. Since an index is a special case of a view, there are at least as many views as indexes to consider for selection [6]. Furthermore, views can also have interactions. A recent paper [68] provides a good survey of view selection techniques.

2.3 Addressing the Physical Design Tuning Problem for New Architectures

In this section, we briefly describe the impact of each new architecture as it relates to the physical design tuning problem, introducing some of the specific challenges for each case. Our approach toward solving each problem is presented in turn in the chapters that follow.

2.3.1 Replicated Database Architecture

With a replicated database, the space of possible physical designs is increased significantly. This is because the workload can be divided into subsets that get routed to replicas, where each replica can have a different physical design. Thus the physical design problem for replicated databases can be conceptually modeled as finding the right subset of the workload and right physical design per replica. The complication is that there is an obvious interaction between these two components — the subset affects the replica's physical design and the physical design affects which subset should be routed to the replica. Furthermore, due to replica specialization there can be an asymmetry in a query's cost among each replica. With a replicated design that is "overspecialized", a query may have a low cost on one specific replica, but a much higher cost on all other replicas. This significantly diminishes the ability to load-balance the workload across replicas at run-time. For example, if a query's "best" (i.e., lowest-cost) replica is not available, it can be routed to its "next-best" replica, where the query may have significantly higher cost. However, with no specialization, a query will have the same cost on every replica. While this provides the maximum opportunity for load-balancing, it misses the opportunity to increase performance by specializing each replica's design to process a specific subset of the workload. Hence, there exists a tension between replicated designs chosen for best-case performance and those for load-balancing, further complicating the space of designs. Indeed there exists a vast space of divergent designs, and ours is the first work to address the physical design tuning problem for replicated

databases.

2.3.2 Hadoop Architecture

With a Hadoop system and our opportunistic design approach, we can consider those views generated as a by-product of query processing as the candidate views for physical design tuning. Since these views have no additional creation cost and encapsulate prior computation performed by the system, they are practically free and reflect the recent workload history. Hence, they are likely to be useful views. However, leveraging these views for physical design is not straightforward due to the presence of UDFs, which create a significant challenge. The previously mentioned approaches [37, 72] that use a syntactic model of the views (based on the syntax of their query plans) are not general enough to address the physical design tuning problem effectively. For instance, if two plans produce the same result using the same operations but apply those operations in different orders, then the results will not match syntactically. Hence, syntactic methods miss the opportunity to reuse results (i.e., opportunistic views) in this case. Moreover, there is no way to re-purpose the results (i.e., perform additional transformations/compensations) even if they are a "close match" to a new query since there is not a semantic understanding of *what* is contained in the results. To overcome the limitations of previous approaches, we first introduce a semantic UDF model that expands the set of possible rewrites for queries with UDFs, and then use this model to build a system that produces rewrites using opportunistic views.

2.3.3 Multistore Architecture

Tuning a multistore system consisting of an RDBMS and a Hadoop system requires solving two physical designs as a single combined system. Unlike the replicated RDBMS architecture that we addressed with divergent design, the multistore data management systems are not of the same type, and a single query can be evaluated in part by both systems, each of which has very different characteristics. The design of one system affects the design of the other, and moreover the way in which a single query plan is split across both systems affects the new opportunistic views it creates. Each system may have different storage budget constraints, and there is a cost to transfer views between the stores (network and data loading) when tuning the designs. Additionally, since the query optimizers of each store use their own internally-defined cost units, some unit normalization is required in order to make meaningful comparisons of query plan costs for potential designs. These issues unique to multistore architectures create a significant challenge for physical design timing. To our knowledge, ours is the first work to address the physical design tuning problem for multistore architectures.

Chapter 3

Divergent Physical Design Tuning for Replicated Databases

Full database replication is commonly used with Database-as-a-Service in the cloud. These cloud-based systems run on commodity hardware and rely on replication to cope with failures as well as achieve scalability. With a replicated database, the system maintains several copies (or replicas) of the data at different nodes, and ensures that they remain synchronized when updates are applied to the database. Two such examples are Microsoft SQL Azure [12,69], which employs 3-way replication, and Amazon's Relational Database Service [8], where up to 5 read replicas of MySQL databases can be launched on-demand to provide scalability. These systems fully replicate the database on several nodes, thus allowing a query to be evaluated on any replica. While the main motivation

behind replication is to cope with node failures and ensure high availability, quite often replication is also used to enable the parallel servicing of a workload in a load-balanced fashion.

In this work, we introduce the paradigm of a *divergent design*, which leverages replication for the purpose of tuning the database system more effectively. Specifically, a divergent design installs a different physical design (e.g., indexes and materialized views) on each replica in order to specialize it for a subset of the workload. In this fashion, a query statement in the incoming workload is routed to the replica that can evaluate it most efficiently. Our proposed approach is in stark contrast to the current state of the practice — a *uniform design* — where each replica has the exact same physical design. By allowing the physical design of each replica to diverge, a divergent design utilizes the aggregate storage in the system more effectively (i.e., more indexes and materialized views are built overall) and consequently enables higher benefits for more queries in the workload together with faster initialization and maintenance of replicas.

A divergent design allows replicas to specialize for subsets of the workload and thus improve query processing performance for a replicated database. Similarly, divergent design may improve the performance of parallel database systems that employ replication for high availability, failure recovery, and performance, e.g., the new Teradata Unity feature. In addition, divergent designs can be particularly beneficial in the context of ad-hoc data analytics. For instance, a data scientist may procure several machines from a private or public cloud (e.g., Amazon EC2), spin up several database instances for the same data set, and then use divergent designs to efficiently evaluate a heavy

analytics workload in parallel. In all these examples, using a divergent design does not require any changes to the underlying query processor. It is only a matter of installing a different design on each replica and taking care of routing queries to the appropriate replicas.

The focus of our work is the problem of *divergent design tuning*, i.e., how to compute a beneficial divergent design for a system that employs database replication. The problem involves numerous technical challenges. Choosing a good divergent design requires reasoning about possible subsets of the workload and how they can benefit from possible configurations, which already leads to a doubly-exponential space of solutions. In particular, divergent design tuning can be viewed as a generalization of physical design tuning for a single node, which is already known to be a hard problem [23]. Another source of complication is the tension between best-case performance and load balancing. Specifically, the specialization of replicas to different subsets of the workload may cause a query to have really good performance on one replica but really bad performance everywhere else. In turn, this variability can result in load imbalance and bad overall performance, e.g., in the case where a query cannot be routed to its "best" replica. Our work shows that it is non-trivial to determine a good trade-off point between specialization and load balancing.

With a replicated database, the current state of the practice is to apply traditional physical design techniques that were developed for a single, centralized database [15, 25, 34] to tune a single replica, and then copy the resulting design to every other replica — resulting in a uniform design. In contrast, a divergent design

installs a "richer" design (i.e., different sets of indexes and materialized views) on each replica. Experimentally, we observe that divergent designs have several benefits over uniform designs: (a) the available space budget for physical configurations is utilized more effectively, (b) the time to materialize the design of each replica is often reduced, (c) update statements are executed faster, and (d) query performance improves significantly without compromising the ability to load-balance the system. Existing systems can already reap some benefits from divergent designs. However, to realize the full potential of divergent design tuning, it becomes necessary to modify the policies for query routing and load balancing. Later, we perform such a experiment to evaluate divergent designs in a real-world scenario, implementing load balancing for several replicas and many concurrent users in the system.

The outline of this chapter is as follows. We formally introduce the concept of a divergent design and define the problem of divergent design tuning in Section 3.1. Our formalization casts the problem as computing a partition of the workload across replicas, and then utilizing existing tools (e.g., DB2's Design Advisor, or the MS SQL's Index Tuning Wizard) to compute the corresponding designs. We next analyze the complexity of the problem and the properties of good divergent designs in Section 3.2. Not surprisingly, our analysis reveals that finding the optimal divergent design is an NP-Hard problem; furthermore we prove that the current state of the practice (uniform design) can be sub-optimal. In Section 3.3 we propose an efficient algorithm to compute effective divergent designs. In Section 3.4 we present an extensive experimental study using a commercial RDBMS to evaluate the potential of divergent designs and also the

effectiveness of our tuning algorithm. Lastly, Section 3.5 provides a short discussion followed by a detailed review of related work in Section 3.6.

3.1 Divergent Design Tuning: Problem Statement

Preliminaries. We first review some basic concepts from the conventional problem of physical design tuning over a single database. The problem of divergent design tuning extends these concepts to a parallel environment where the database is replicated.

In conventional physical design tuning, we are given a representative workload W and a space budget b, and the goal is to compute a configuration (i.e., design) that minimizes the cost to evaluate W and fits within the space budget b. The computed design typically comprises materialized views and (primary or secondary) indexes over tables and views. Formally, let $W = Q \cup U$ where Q is the *query workload* and U is the *update workload*. Since W plays the role of a representative workload, it is common to provide a weight function $f : W \rightarrow \Re$, such that $f(x)$ corresponds to the importance of query or update statement x in W. W and f are typically specified by the administrator or they can be obtained automatically, e.g., by mining the query logs of the database system.

We use $cost(x, I)$ to denote the cost of evaluating (query or update) statement x assuming that I is the physical design of the system. Given a design I, we can define

the cost of evaluating W as follows:

$$SingleCost(W, f, I) = \sum_{q \in Q} f(q) cost(q, I) + \sum_{u \in U} f(u) cost(u, I)$$

Building on these definitions, the physical design tuning problem is defined as computing the design I such that I has size not greater than b and $SingleCost(W, f, I)$ is minimal.

As noted previously, the cost function can be evaluated efficiently in modern systems (i,e., without materializing I) using a *what-if optimizer* [26], and most modern commercial database management systems come with a *design advisor* tool that automates the tuning process [25, 33, 102]. We model a design advisor as a function $DBAdv$ that takes as input the workload W, the weight function f, and the space budget b and outputs a design I, i.e., $I = DBAdv(W, f, b)$. Since computing the optimal design is an NP-Hard problem, the advisor relies on heuristics to output a good solution that is hopefully close to optimal.

Divergent Physical Design Tuning. We extend the previous concepts to a system where the database is replicated across n machines. We consider the cases described in Section 3, where each replica $i \in [1, n]$ holds a full copy of the database. Moreover, we assume that replicas have the same space budget b for their design. Our techniques are readily extensible to the case where the unit of replication is a partition of the database, or when replicas may have different space budgets.

In this particular setting, each query statement q can be evaluated by any

replica in the system. (Of course, the performance of query evaluation may be different depending on which replica is chosen.) On the other hand, each update statement u has to be evaluated at each replica, in order to ensure consistency. The system may use different strategies to ensure synchronization for updates, e.g., eager vs lazy, but this choice is completely orthogonal to our techniques.

The high-level idea of a divergent physical design is to allow each replica to have a different physical design, tailored to a particular subset of the workload. Formally, let $\{Q_1, \ldots, Q_n\}$ denote a partition of the query workload Q to n (potentially overlapping) subsets. The partition of Q induces a partition of W in corresponding subsets $W_1, \ldots, W_n$, that we call *sub-partitions* such that $W_i = Q_i \cup U$. We use *partitions*(W) to denote the set of such partitions. Essentially, W_i represents the subset of the workload for which replica i is specialized. Note that each W_i contains U, since all updates have to be applied to all replicas. We couple the partition $\{W_1, \ldots, W_n\}$ with a set of corresponding weight functions $\{f_1, \ldots, f_n\}$, where $f_i()$ sets the weights of queries and updates in W_i. (We will impose certain restrictions on f_i later.) In what follows, we use $p = \{(W_1, f_1), \ldots, (W_n, f_n)\}$ to denote the combined information of workload subsets and weight functions. Note that the *uniform design* $p_{unif} = \{(W, f), \ldots, (W, f)\}$ captures the current practice in real systems, where each replica is equipped with the same physical design $I_{unif} = DBAdv(W, f, b)$ that is computed based on the complete workload.

Definition 3.1.1 (Divergent Design)**.** *Given a workload $W = Q \cup U$, a weight function f, and a number of replicas n, a divergent design corresponds to a set*

$p = \{(W_1, f_1), \ldots, (W_n, f_n)\}$, *such that:*

- $W_i = Q_i \cup U$ *for all* $i \in [1, n]$, *where* $Q_1 \cup \cdots \cup Q_n = Q$.
- $f_i(u) = f(u)$ *for all* $u \in U$.
- $\sum_{i \in [1,n]} f_i(q) = f(q)$.

The design of each replica is computed as $I_i = DBAdv(W_i, f_i, b)$.

Our definition places three constraints on p. The first one corresponds to our earlier observation that each W_i contains all the updates. The second and third constraints regulate the weight functions $f_i()$. Specifically, $f_i(u) = f(u)$ preserves the importance of an update *on each replica*, whereas $\sum_i f_i(q) = f(q)$ preserves the weight of a query *across all replicas*. These properties are aligned with the setting that we consider: each occurrence of u has to be routed to every replica, whereas each occurrence of a query q is routed to one replica. The definition also states that each replica will be tuned with a design I_i that results from invoking $DBAdv()$ on the corresponding subset W_i and weight vector f_i.

An implication of our definitions is that we essentially view the function $DBAdv()$ as a black box. We chose this modeling for two reasons. First, we reduce the problem of divergent design tuning to that of finding the right partition of the workload and corresponding weight functions that maximize a suitable performance metric (to be detailed shortly). Second, we can leverage the existing literature on automated tuning techniques, and in particular the existing advisor tools, in order to compute the design of each replica.

To model the performance of a divergent design p we could extend the *SingleCost* metric, where each query q in W is now evaluated by the replica with the best physical design for q. Formally, let $I_1, \ldots, I_n$ denote the replica designs corresponding to p. We define $J_1^q, \ldots, J_n^q$ as the permutation of $I_1, \ldots, I_n$ such that $cost(q, J_1^q) \leq cost(q, J_2^q) \leq \cdots \leq cost(q, J_n^q)$, i.e., J_j^q is the physical design of the j-th best replica for q. Hence, the best-case total evaluation cost for W can be expressed as follows:

$$BestCost(p) = \sum_{q \in Q} f(q) cost(q, J_1^q) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i)$$

This metric reflects the total work performed by the system when each query is routed to its best replica and each update is routed to all replicas.

The problem with the previous metric is that it ignores completely the issues of load-balancing and parallel processing for queries. Specifically, the metric does not guard against the case where $cost\ (q, J_1^q) << cost(q, J_2^q)$ and hence q performs badly on every replica except the one with design J_1^q. A schematic representation is given in Figure 3.1, which depicts $cost(q, J_j^q)$ for a system with $n = 3$ replicas. Design p_A ensures a low cost for the best replica of q, but a much higher cost for other replicas.

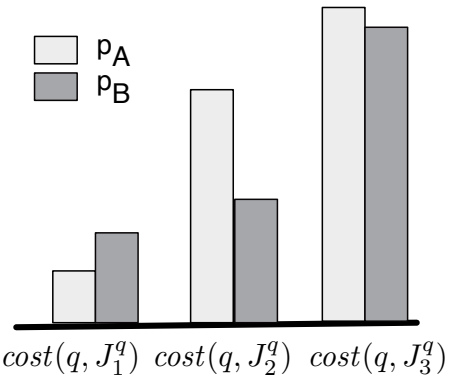

Figure 3.1: An illustration of the trade-off between load balancing and performance.

On the other hand, design p_B ensures that two replicas can evaluate q with low cost, albeit higher than the best-case replica of p_A. In a sense, design p_B trades off best-case performance for flexibility in routing queries. The uniform design p_{unif} stands at the extreme point of this trade-off, since every query has the same cost on every replica.

To address load-balancing in our problem statement, we introduce a load-balancing factor $m \leq n$ that specifies the number of "low-cost" replicas in p for each query in Q. We expect the DBA to set m based on the desired behavior of the system. If parallel processing and load balancing are not important, then $m = 1$ will optimize the choice of p for the best-case cost (essentially, the previous metric). On the other hand, $m = n$ will lead to a design p such that q has similar performance on all replicas. Values of m between these two extremes will trade off best-case performance for the ability to load balance. We incorporate factor m in the performance metric as follows:

$$TotalCost(p, m) = \sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, J_j^q) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i)$$

The second term is simply the cost to push updates to all the replicas. The first term reflects the total work to evaluate the workload, assuming that the cost of each query q is "distributed" across the m replicas with the best physical designs for q. Hence, this metric favors a divergent design that ensures at least m good options for each query. Our proposed metric is a generalization of the cost metric for conventional physical design tuning, to the case where m replicas equally share the cost of each query. Returning to the example of Figure 3.1, design p_B outperforms p_A for $m = 2$, but p_A is better for

$m = 1$.

We now formulate the problem that we tackle in this work, namely computing the optimal divergent design.

Definition 3.1.2 (Optimal Divergent Design)**.** *Given a workload W, a weight function f, a number of replicas n and a load-balancing factor $m \leq n$, compute the divergent design p that minimizes $TotalCost(p, m)$.*

We can also formulate an alternative problem statement, where we optimize m based on a desired trade-off between best-case performance and load balancing.

Definition 3.1.3 (Optimal m and Design)**.** *Given a workload W, a weight function f, a number of replicas n, a threshold $\tau \in (0, 1]$ and the uniform design $p_{unif} = \{(W, f), \ldots, (W, f)\}$, compute the maximal m and corresponding divergent design p such that $TotalCost(p, m) \leq \tau\, TotalCost(p_{unif}, n)$.*

In this case, we seek to find the divergent design that offers the most opportunities for load balancing and improves on the baseline uniform design at least by a factor of τ. In our theoretical analysis provided in the next section, we prove that any PTIME algorithm to solve the first problem yields a PTIME algorithm to solve the second problem, by using binary search to identify the maximal value for m. Hence, from this point onward, we focus on the first problem.

For purposes that will become clear in the next section, we distinguish the class of *l-balanced divergent designs*. The formal definition follows.

Definition 3.1.4 (l-Balanced Design)**.** *A divergent design* $p = \{(W_1, f_1), \ldots, (W_n, f_n)\}$ *is called* l*-balanced iff the following two conditions hold:*

- *Each* $q \in W$ *appears in exactly* l *sub-partitions of* p*, i.e.,* $\{i \mid i \in [1, n] \wedge \ q \in W_i\}$ *has exactly* l *elements for any* $q \in W$*.*
- $f_i(q) = f(q)/l$ *if* $q \in W_i$ *and* $f_i(q) = 0$ *otherwise, for any* $q \in W$ *and* $i \in [1, n]$*.*

In an l-balanced design p, each query q directly affects the design of exactly l replicas. Moreover, the original weight of q is uniformly distributed among the l sub-partitions that contain q. A special case of an l-balanced design is the *fully-balanced design* $p_{FB} = \{(W, f_{FB}), \ldots, (W, f_{FB})\}$ where $l = n$, i.e., each query q appears in every sub-partition with weight $f_{FB}(q) = f(q)/n$. Similar to the uniform design, the fully-balanced design installs the same design $I_{FB} = DBAdv(W, f_{FB}, b)$ in each replica. The difference is that queries have their weights deflated by n, which intuitively reflects the fact that q can be evaluated by any of the n replicas with equal cost $cost(q, I_{FB})$. The counter-part to the fully-balanced design p_{FB} is a 1-balanced design p, where each query q directly affects the design of exactly one replica.

3.2 Problem Analysis

Having defined the problem of divergent design tuning, our next step is to develop some intuition about the space of possible solutions. Our approach is two-pronged. First, we present a theoretical analysis on the complexity of the problem and the properties of good divergent designs. We couple these theoretical results with an ex-

haustive investigation of the space of divergent designs for a small-scale instance of the problem. Overall, the theoretical results and our empirical observations demonstrate the challenges behind the problem: there is a vast search space of divergent designs; identifying the optimal design is computationally hard; there exists a very big performance gap between good and bad designs; and, there are relatively few good designs. These insights contribute directly to the design of the tuning algorithm presented in the next section.

3.2.1 Theoretical Analysis

For the purpose of our theoretical analysis, we assume that $DBAdv(W, f, b)$ returns the *optimal* configuration for any W, f and b. This assumption allows us to analyze the properties of divergent design tuning without having to model the imperfections of the advisor. We stress that the tuning algorithm we present in Section 3.3 *does not* rely on this assumption.

First, we investigate the computational complexity of identifying the optimal divergent design. The following proof shows that the decision problem of divergent design tuning is NP-Hard, through a reduction from the NP-Complete subset sum problem.

Theorem 3.2.1. *The optimal divergent design cannot be computed in polynomial time unless* $P = NP$.

Proof. We formulate the corresponding decision problem, termed DivergentPartition, which takes as input an instance (W, f, n, m) of the optimal design problem and a threshold ρ, and asks whether there exists a divergent design p such that

$TotalCost(p, m) \leq \rho$. We then proceed to show that DivergentPartition is an NP-Hard problem by reducing the subset sum problem to the optimal divergent design problem.

The subset sum problem takes as input a set of integers and another integer called the target and asks whether there exists a subset of the integers such that their sum equals to the target.

We will create an instance of DivergentPartition as follows: Consider a workload with k queries. We map each of the queries to some integer. We pick one query, that we call the target, to be mapped to the target integer of the subset sum problem. Hence, from the set of queries, we have created a set of $k-1$ integers and a target. Note that there can be many ways to map a query to an integer. For example we could use a binary representation of the query and translate that into a positive integer. Also the target can be identified by many ways. For example, we can assign all $k-1$ queries to positive integers as explained above but assign the target to some negative integer whose absolute value will be the target integer of the subset sum problem.

Next, we will define the cost function of the instance of the problem we are creating. We define $cost(q, I) = 1$ for all queries and index sets unless the following conditions hold: Query q belongs to a subpartition W_i of a partition p of the workload W and index set I is calculated given W_i as input; subpartition W_i contains the target query and at least one more query (i.e., the size of W_i is at least 2); the sum of the values mapped to the queries in W_i without the target query is equal to the absolute value of the integer mapped to the target query. If all conditions above hold for some

q and I then $cost(q, I) = 0.5$.

We pick the threshold τ to be equal to k. We will show that by solving the divergent design problem we also solve the subset sum. First, we note that the total cost of the workload will be either equal to k (i.e., the number of queries) or less than k if we can satisfy the conditions above and reduce the cost by half for some queries.

If the optimal cost of the divergent design problem returns k then the answer for the subset sum problem should be negative. This is because if the TotalCost calculated is equal to k then there is no subset of the queries in W whose corresponding values sum up to the value corresponding to the target query. Hence $cost(q, I) = 1$ for all q and I and the total cost equals to the number of queries which is k.

If the optimal cost returned by the divergent design problem is less than k, then the answer for the subset sum problem should be positive. This is because to have less cost than k, there is at least one query q and an index I such that $cost(q, I) = 0.5$. But then, for this to happen according to the conditions above, q is part of a partition p whose corresponding values add up to the target of the subset problem. □

Instead of looking for an optimal algorithm to solve the problem, our goal is to develop an efficient algorithm that computes good divergent designs. In this direction, we analyze further the space of possible divergent designs in order to build some intuition on the properties of good solutions.

Recall that the fully-balanced design p_{FB} installs the same configuration in all replicas, computed based on the complete workload W and a modified weight function

$f_{FB}(q) = f(q)/n$. The intuition behind p_{FB} is that each query q can be evaluated by any replica with the same cost, and hence the original weight $f(q)$ is equally spread across all replicas. This intuition makes the fully-balanced design a natural choice for $m = n$, and in fact we prove that it is an optimal solution.

Theorem 3.2.2. *The fully-balanced design p_{FB} is optimal if $m = n$, i.e., $TotalCost(p_{FB}, n) \leq TotalCost(p, n)$ for any other design p.*

Proof. Consider a partition $p = \{(W_i, f_i)\}$ of workload W that is used to create configurations $\{I_1, \ldots, I_n\}$. Because $m = n$, we sum over all n replicas and obtain:

$$\sum_{j \in [1,n]} \frac{f(q)}{n} cost(q, J_j^q) = \sum_{i \in [1,n]} \frac{f(q)}{n} cost(q, I_i) \tag{3.1}$$

By Equation 3.1 and definition of *TotalCost*, we substitute and obtain

$$TotalCost(p, n) = \sum_{i \in [1,n]} [\sum_{q \in Q} \frac{f(q)}{n} cost(q, I_i) + \sum_{u \in U} f(u) cost(u, I_i)] \tag{3.2}$$

Let $f'(q) = \frac{f(q)}{n}$ and $f'(u) = f(u)$. Then,

$$TotalCost(p, n) = \sum_{i \in [1,n]} SingleCost(W, f', I_i) \tag{3.3}$$

Let I' be the configuration in $\{I_1, \ldots, I_n\}$ that causes the smallest cost for (W, f') (ie., $SingleCost(W, f', I') \leq SingleCost(W, f', I_i)$ for all $i \in [1, n]$). If we replace

$SingleCost(W, f', I_i)$ by $SingleCost(W, f', I')$ in Equation 3.3 then

$$TotalCost(p, n) \geq n\, TotalCost(W, f', I') \tag{3.4}$$

Let $I = DBAdv(W, f', b)$. By definition of the design advisor, I is the best possible index set for workload (W, f'), hence $SingleCost(W, f', I') \geq SingleCost(W, f', I)$ and from Equation 3.4

$$TotalCost(p, n) \geq nSingleCost(W, f', I) \tag{3.5}$$

The theorem follows because as we show next, $TotalCost(p_{FB}, n) = nSingleCost(W, f', I)$:

$$TotalCost(p_{FB}, n) = \sum_{q \in Q} f(q) cost(q, I) + n \sum_{u \in U} f(u) cost(u, I)$$
$$= n[\sum_{q \in Q} \frac{f(q)}{n} cost(q, I) + \sum_{u \in U} f(u) cost(u, I)]$$
$$= nSingleCost(W, f', I)$$ □

Our next step is to consider the other extreme point of $m = 1$, where we only care about best-case performance. A 1-balanced design is a natural choice for $m = 1$, since it naturally associates each query with a single replica. However, proving optimality is far from trivial, given that there are many such designs and also because the best cost for q may not come from the configuration that has considered q. Still, we are able to show an important result: the fully-balanced design cannot beat *any* 1-balanced design for $m = 1$. In fact, we prove the following generalization:

Theorem 3.2.3. *$TotalCost(p, m) \leq TotalCost(p_{FB}, m)$ for any $m \in [1, n]$ and any*

m-balanced design p.

Proof. First we write the expression for $TotalCost(p_{FB}, m)$

$$TotalCost(p_{FB}, m) =$$

$$\sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, I_{FB}) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_{FB})$$
$$= \sum_{q \in Q} f(q) cost(q, I_{FB}) + n \sum_{u \in U} f(u) cost(u, I_{FB})$$

Next we consider the expression for $TotalCost(p, m)$:

$$TotalCost(p, m) =$$

$$\sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, J_j^q) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i)$$

Given a query q, let $I_1^q, \ldots, I_m^q$ denote the index sets of the m sub-partitions that contain q. The definition of the sequence $J_1^q, \ldots, J_n^q$ implies the following inequality:

$$\sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, J_j^q) \leq \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, I_j^q)$$

When applied to $TotalCost(p, m)$ we obtain the following:

$$TotalCost(p, m) \leq$$

$$\sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, I_j^q) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i) =$$
$$\sum_{i \in [1,n]} \sum_{q \in W_i} \frac{f(q)}{m} cost(q, I_i) + \sum_{i \in [1,n]} \sum_{u \in W_i} f(u) cost(u, I_i)$$

The last equality results from a rearrangement of the query cost terms and the fact that p is an m-balanced design, which means that each q appears in exactly m sub-partitions. Doing a unification of the two sums, we arrive at the inequality shown below:

$$TotalCost(p, m) \leq \sum_{i \in [1,n]} SingleCost(W_i, f', I_i)$$

where W_i is the workload $Q_i \cup U$ with the weight function $f'(q) = f(q)/m$ and $f'(u) = f(u)$. The optimality of the configuration advisor and because $I_i = DBAdv(W_i, f', b)$ (where b is a space budget) implies that $SingleCost(W_i, f', I_i) \leq SingleCost(W_i, f', I_{FB})$. This leads to the following inequality and subsequent expansion:

$$\begin{aligned} TotalCost(p, m) &\leq \sum_{i \in [1,n]} SingleCost(W_i, f', I_{FB}) \\ = \sum_{i \in [1,n]} \sum_{q \in W_i} \frac{f(q)}{m} cost(q, I_{FB}) &+ \sum_{i \in [1,n]} \sum_{u \in W_i} f(u) cost(u, I_{FB}) \\ = \sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, I_{FB}) &+ \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_{FB}) \end{aligned}$$

The last expression results again from a rearrangement of terms and the fact that p is an m-balanced design. The final step is to factor out the common query cost and update cost terms.

$$\begin{aligned} &TotalCost(p, m) \leq \\ &\sum_{q \in Q} f(q) cost(q, I_{FB}) + n \sum_{u \in U} f(u) cost(u, I_{FB}) = \\ &TotalCost(p_{FB}, m) \end{aligned}$$

□

This result reflects the tension between load-balancing and performance. The fully-balanced design offers the greatest flexibility for load-balancing (and is in fact optimal for $m = n$), since each query q has the same cost on each replica. On the other hand, an m-balanced design tries to specialize m replicas per query, which reduces the flexibility for load-balancing but improves the cost of workload evaluation.

The previous two theorems lead us to the following intuition about the properties of a "good" divergent design p: *The overlap between sub-partitions should increase as m increases.* Indeed, the fully-balanced design p_{FB}, whose sub-partitions W_i overlap completely, is optimal for $m = n$. Conversely, for any $m < n$, p_{FB} cannot beat any m-balanced design where each q appears in exactly m sub-partitions. These observations play a crucial role in the divergent design algorithm that we describe in the next section.

We conclude our analysis by comparing divergent designs against the current practice in real systems, which is the *uniform design*. We expect p_{unif} to be mostly relevant for $m = n$, and hence a natural starting point for our analysis is to compare p_{unif} to the optimal design p_{FB}. We prove below that we should always prefer the fully balanced design compared to the uniform design for any value of the load balancing factor.

Theorem 3.2.4. *The uniform design cannot outperform the fully balanced design for*

any load balancing factor, i.e.,

$$TotalCost(p_{unif}, m) \geq TotalCost(p_{FB}, m) \quad for\ any \quad m \in [1, n].$$

Proof. By definition of *TotalCost*, we obtain:

$$TotalCost(p_{unif}, m) = \sum_{q \in Q} f(q) cost(q, I_{unif}) + n \sum_{u \in U} f(u) cost(u, I_{unif}).$$

We can rewrite the above equation as follows:

$$TotalCost(p_{unif}, m) = n[\sum_{q \in Q} \frac{f(q)}{n} cost(q, I_{unif}) + \sum_{u \in U} f(u) cost(u, I_{unif})].$$

By definition of *SingleCost* and the fully balanced design, we obtain:

$$TotalCost(p_{unif}, m) = n[SingleCost(W, f_{FB}, I_{unif})]$$

where f_{FB} is the weight function used by the fully balanced design. Since I_{FB} is optimal for (W, f_{FB}), then

$SingleCost(W, f_{FB}, I_{FB}) \leq SingleCost(W, f_{FB}, I_{unif})$ and from the above, we obtain that: $TotalCost(p_{unif}, m) \geq nSingleCost(W, f_{FB}, I_{FB})$. The theorem follows because we can easily obtain:

$$TotalCost(p_{FB}, m) = nSingleCost(W, f_{FB}, I_{FB}).$$ □

Intuitively, this result can be explained by the asymmetry between queries and updates. Recall that a query q can be evaluated by *any single replica*, whereas an update u has to be evaluated by *all n replicas*. Hence, the contribution of queries to the total work metric decreases by a factor of n compared to the updates. The fully balanced design p_{FB} uses $f_{FB}(q) = f(q)/n$ for query weights, and therefore the configuration

$I_{FB} = DBAdv(W, f_{FB}, b)$ takes directly into account this decrease in contribution. On the other hand, p_{unif} ignores the deflation of query importance, as it uses the unmodified query weights. The resulting configuration $I_{unif} = DBAdv(W, f, b)$ is basically oblivious to the increased importance of updates.

We have already seen that p_{FB} cannot lose to p_{unif} (Theorem 3.2.4), and by Theorem 3.2.3, we conclude that given a fixed value for m, the uniform design p_{unif} cannot improve on *any* m-balanced design p.

Corollary 3.2.5. *$TotalCost(p_{unif}, m) \geq TotalCost(p, m)$ for any $m \in [1, n]$ and any m-balanced design p.*

This general result provides even stronger evidence in favor of substituting p_{unif} (again, the current state of the practice) with a divergent design. Note that *any* m-balanced design has the potential to improve on p_{unif}. The reason can be traced again to the asymmetry between queries and updates, but also to the semantics of the load-balancing factor m. Recall that m specifies that each query q should have m replicas on which it has a low execution cost. An m-balanced design p attempts to achieve this goal by placing q in exactly m sub-partitions and thus having q affect the design of m replicas. This placement still affords some specialization per replica, since each W_i contains a subset of the queries. In contrast, p_{unif} corresponds to a single configuration I_{unif} that is tuned for the whole workload, which means that no replica is specialized. Moreover, p_{unif} may over-provision in terms of load balancing, since each query has the same cost on all $n \geq m$ replicas. These two factors contribute to the sub-optimality of

p_{unif} compared to an m-balanced design.

Next, we show how to solve the optimal load balancing problem given any solution to the optimal divergent design problem. The key behind our solution is that *TotalCost* monotonically increases with the load balancing factor m, as shown in Theorem 3.2.6. Therefore, we could search for the solution to the optimal load balancing problem using binary search on m among the solutions given by the optimal divergent design.

Theorem 3.2.6. *Let p_x be a divergent design given by a solution to the optimal divergent design problem for $m = x$. We will show that $TotalCost(p_x, x) \leq TotalCost(p_{x+1}, x+1)$.*

Proof. First, we observe that $TotalCost(p_x, x) \leq TotalCost(p_{x+1}, x)$. That is because design p_x is optimal for $m = x$ since it has been produced by a solution to the optimal divergent design problem. Hence, any other design (eg. p_{x+1}) will cause the same or worse *TotalCost*. The theorem follows because $TotalCost(p_{x+1}, x) \leq TotalCost(p_{x+1}, x+1)$, which is easy to verify by simple mathematical manipulations and the definition of *TotalCost*. □

3.2.2 Empirical Analysis

The second part of our analysis is empirical, and aims to provide further insights on the space of good divergent designs.

We conduct a small-scale empirical study of l-balanced designs with a workload of five queries from the TPC-DS benchmark [95], using IBM DB2 as the underlying

DBMS. The details of the experimental setup are given in Section 3.4. We focused on balanced designs in order to contain the scope of the study and also due to the nice theoretical properties of these designs. We handpicked the queries and tuned the available space budget to simulate the following realistic scenario: different subsets of queries can benefit from similar indexes, but the indexes for different subsets are too large to fit in a single configuration. This is representative of what we expect to see in practice. Intuitively, a good divergent design will specialize a different replica for each subset.

We set $n = 3$ and then exhaustively enumerate all possible l-balanced designs, for $l \in [1, 3]$. Overall, this yielded 486 distinct designs. For each design p, we computed the corresponding configurations I_1, I_2, and I_3 by invoking the DB2 Design Advisor on each sub-partition W_i. Finally, we used DB2's what-if query optimizer to compute $\mathit{TotalCost}(p, m)$ for $m = 1$ and $m = 2$. (We do not consider $m = 3$ since we already know that p_{FB} is optimal.)

We performed the whole computation on Amazon EC2 using ten separate IBM DB2 instances so that we could parallelize the process. Overall, it required many hours of computation time in order to fully explore the search space. The main overhead was the cost of invoking DB2's Design Advisor (several seconds) to compute the physical design for each sub-partition W_i, even with our toy workload. Since an exhaustive search will be impractical for real workloads with tens of queries, any reasonable divergent design tuning algorithm should explore only a small part of the search space.

The results revealed several interesting properties for the space of balanced

	25-th Perc.	Median	75-th Perc.	Max
$m = 1$	$\times 2.54$	$\times 4.47$	$\times 25.1$	$\times 26.8$
$m = 2$	$\times 2.98$	$\times 5.07$	$\times 5.24$	$\times 5.42$

Table 3.1: Distribution statistics for divergent design costs in the exhaustive experiment. For designs in each cost percentile, we show the ratio of *TotalCost* to the optimal design cost.

designs for this particular setup. First, we observed that the optimal design for a fixed value of m is an m-balanced design, for $m \in [1, 3]$. Intuitively, an l-balanced design with $l > m$ provides more flexibility for load balancing than specified by m, and hence misses the opportunity to tighten the specialization of replicas. On the other hand, $l < m$ leads to over-specialization and hence to imbalances among the first m replicas for each query. Based on these empirical observations, we can formulate the following conjecture.

Conjecture 3.2.7. *Let p_m denote an m-balanced divergent design and p_l denote an l-balanced design, for $l \neq m$. Then, it holds that* $\min_{p_m} TotalCost(p_m, m) \leq \min_{p_l} TotalCost(p_l, m)$.

As a next step in our analysis, we examine the distribution statistics of the *TotalCost*() metric within each class of m-balanced designs, for $m = 1, 2$. In each case, there are 6 designs, out of the 243 designs in the class, which achieve the minimal value for *TotalCost*(). Table 3.1 further shows the distribution of the *TotalCost*() values within each class in relation to these optimal designs. For instance, for $m = 1$, the 25-th percentile (that is, the *TotalCost* value that bounds 25% of the designs) is $\times 2.54$ worse than the optimal cost within the class. Overall, the results show that non-optimal

designs have significantly higher cost, and hence a random selection is not likely to yield a good solution to the divergent tuning problem.

It is also interesting to examine the cost of p_{unif} with respect to the optimal in each class. Specifically, $TotalCost(p_{unif}, 1)$ is $\times 26.8$ higher than the optimal cost for the class of 1-balanced designs. The same ratio for $m = 2$ is $\times 5.42$. Hence, the current practice of employing a uniform design may lead to a huge loss in performance compared to a divergent design.

3.3 The DivgDesign Algorithm for Divergent Design Tuning

In this section, we present an efficient algorithm called DivgDesign that computes effective divergent designs. We first provide an overview of the key ideas behind the algorithm, and then present the detailed pseudocode. We conclude with a theoretical analysis that demonstrates the nice theoretical properties of DivgDesign.

3.3.1 Overview of Our Approach

The first design choice behind DivgDesign is to output m-balanced designs, for the value of m in the problem specification. This choice is justified by the theoretical evidence for the nice properties of such designs: the m-balanced design for $m = n$ is optimal; and any m-balanced design has the potential to improve on the baseline p_{unif} partition. The key challenge therefore is to identify an m-balanced design p such that

$TotalCost(p, m)$ is small.

Intuitively, a good design should have the property that $J_1^q, \ldots, J_m^q$ (the m best configurations for q among $I_1, \ldots, I_n$) correspond to the configurations of the sub-partitions that contain q. Otherwise, this implies that q affects directly the computation of some configuration $I_i = DBAdv(W_i, f_i, b)$ (by being part of the workload W_i), without however using I_i in the $TotalCost$ metric. In that sense, it is better to take q out of W_i, in order to allow I_i to be tuned even further for the remaining statements in W_i.

Enforcing the aforementioned property is quite challenging, due to the black-box nature of function $DBAdv()$. Essentially, it is infeasible to partition queries based on some analysis of their features, given that we do not make any assumptions about the implementation of $DBAdv()$. Instead, we have to rely on a trial-and-error approach to identify a good partition p, but with a smart strategy that avoids enumerating an exponential number of partitions.

The proposed DivgDesign algorithm employs a strategy that is inspired by the well-known k-means clustering algorithm. The latter is designed to solve clustering problems that employ a black-box distance function. The idea is simple and yet quite effective: start with a random selection of points as the cluster medoids, assign the remaining points to clusters based on the closest medoid, identify the new medoid per cluster, and then iterate in the same fashion until the clustering converges to a stable state. DivgDesign works in a similar fashion. First, it computes a random partition of the workload $\{W_1^0, \ldots, W_n^0\}$ and the corresponding configurations $I_i^0 = DBAdv(W_i^0, f_i^0, b)$, for $i \in [1, n]$. Subsequently, the algorithm computes $cost(q, I_i^0)$ for

	q_1	q_2	q_3
W_1^0	✓	✓	
W_2^0			✓

(a)

	q_1	q_2	q_3
I_1^0	**10**	20	10
I_2^0	20	**10**	**5**

(b)

	q_1	q_2	q_3
W_1^1	✓		
W_2^1		✓	✓

(c)

Figure 3.2: An example of running the DivgDesign algorithm for $m = 1$, $n = 2$ and a workload of three queries: (a) Initial design, (b) Query costs under the corresponding configurations (minimum costs are indicated with bold); (c) Refined designs.

all $q \in Q$ and $i \in [1, n]$, and "moves" each query q to the sub-partitions that correspond to the lowest evaluation costs. This process yields a new partition $\{W_1^1, \ldots, W_n^1\}$, which is used to compute new configurations $I_1^1, \ldots, I_n^1$ and another assignment of queries to sub-partitions. This iterative process continues until there is convergence.

Figure 3.2 illustrates an iteration of DivgDesign with a simple example. Assume that we have $n = 2$ replicas, $m = 1$ and the workload contains three queries q_1, q_2, q_3 and no updates. The algorithm starts with an initial random partition $p = \{W_1^0, W_2^0\}$, shown in Figure 3.2(a). We show the partition in the form of a matrix, where a cell (i, q) is checked if q appears in W_i^0. This partition yields configurations $I_1^0 = DBAdv(W_1^0, f_1^0, b)$ and $I_2^0 = DBAdv(W_2^0, f_2^0, b)$, which are computed by invoking the configuration advisor on the corresponding sub-partitions. Subsequently, the algorithm computes the execution cost of each query under these configurations. An example of resulting costs are shown in Figure 3.2(b). Note that query q_2 appears in W_1^0 and yet has the least execution cost with configuration I_2^0. This may happen because q_2 is more "similar" to q_3 in terms of beneficial configurations, e.g., both benefit from the same indexes or materialized views, and hence I_2^0, which is computed based on q_3, is more effective for q_2 compared to I_1^0. Based on the computed costs, the algorithm moves

each query to the sub-partition corresponding to the least cost. The resulting partition $\{W_1^1, W_2^1\}$ is shown in Figure 3.2(c), where q_2 now appears in the same sub-partition with q_3. This new partition becomes the input of the next iteration, and the iterations continue until a convergence condition (which we discuss later) is satisfied.

Overall, DivgDesign tries to create partitions that group queries that are highly "similar" with respect to the configuration elements (e.g., indexes or materialized views) that benefit them. This strategy ensures that each resulting configuration $I_i = DBAdv(W_i, f_i, b)$ is beneficial for all the queries in W_i. Equally importantly, this similarity is computed indirectly through the output of the black-box function $DBAdv()$, without the need to model the logic of the design advisor.

3.3.2 Algorithm Definition

Figure 1 shows the pseudocode of the DivgDesign algorithm that we introduce in this work. The algorithm receives as input the workload W, the number of replicas n, the space budget b, and the load balancing factor m, and outputs an m-balanced design p such that $TotalCost(p, m)$ is small.

The algorithm picks (line 2) an initial design $p^0 = \{(W_1^0, f_1^0), \ldots, (W_n^0, f_n^0)\}$ and then refines it in an iterative fashion (lines 4–17). Keep in mind that the weight functions $f_1^0, \ldots, f_n^0$ are fully specified through the sub-partitions $W_1^0, \ldots, W_n^0$ given that p is an m-balanced design (Definition 3.1.4). Hence, it suffices to pick a random partition $\{W_1^0, \ldots, W_n^0\}$ in order to initialize p^0. We use x as the variable that tracks the current iteration number, and use $p^x = \{(W_1^x, f_1^x), \ldots, (W_n^x, f_n^x)\}$ for the design at

Algorithm 1 DivgDesign

Input: Workload W, number of replicas n, load-balancing factor m, space budget b.
Output: A divergent design p that is m-balanced.

1: **function** DivgDesign(W,n,m,b) ▷ Knobs: Improvement threshold ϵ, Max number of iterations *IterMax*
2: Pick a random m-balanced design $p^0 = \{(W_1^0, f_1^0), \ldots, (W_n^0, f_n^0)\}$
3: $x \leftarrow 0$
4: **repeat**
5: **for each** $i \in [1, n]$ **do**
6: $I_i^x \leftarrow DBAdv(W_i^x, f_i, b)$
7: **end for**
8: Initialize $W_i^{x+1} = U$ for all $i \in [1, n]$
9: **for each** $q \in Q$ **do**
10: Let $J_1^q, \ldots, J_n^q$ be a permutation of $I_1^l, \ldots, I_n^l$ such that $cost(q, J_1^q) \leq cost(q, J_2^q) \cdots \leq cost(q, J_n^q)$
11: **for each** $j \in [1, m]$ **do**
12: add q to W_i^{x+1} such that $I_i^x = J_j^q$
13: **end for**
14: **end for**
15: $p^{x+1} \leftarrow \{(W_1^{x+1}, f_1^{x+1}), \ldots, (W_n^{x+1}, f_n^{x+1})\}$
16: $x \leftarrow x + 1$
17: **until** $|TotalCost(p^{x-1}, m) - TotalCost(p^x, m)| < \epsilon$ or $x > IterMax$
18: **return** p
19: **end function**

the beginning of the current iteration, and $p^{x+1} = \{(W_1^{x+1}, f_1^{x+1}), \ldots, (W_n^{x+1}, f_n^{x+1})\}$ for the design at the end of the current iteration. (As always, the weight functions are determined based on the sub-partitions according to Definition 3.1.4.) Each iteration comprises two steps. In the first step, the algorithm computes the configuration I_i^x of replica i by invoking the design advisor on W_i^x, i.e., $I_i^x = DBAdv(W_i^x, f_i^x, b)$. In the second step, the algorithm computes the new design based on these configurations. Specifically, for each query $q \in Q$, DivgDesign computes $cost(q, I_i^x)$ for all $i \in [1, n]$, and then identifies the m configurations $J_1^q, \ldots, J_m^q$ from $I_1^x, \ldots, I_n^x$ that yield the m least costs for q. Then, q is added to W_i^{x+1} if I_i is among $J_1^q, \ldots, J_m^q$. Hence, each query

is assigned to the m sub-partitions that correspond to the m most efficient evaluation plans for the query.

How do we know that this iterative process converges to a good design? One of our key contributions is to show that each iteration of DivgDesign cannot worsen the quality of p under certain assumptions.

Theorem 3.3.1. *Let p^x be the design at the beginning of iteration x (line 4 in Figure 1) and p^{x+1} be the design at the end of the iteration (line 15 in Figure 1). Assuming that DBAdv() returns optimal configurations, it holds that $TotalCost(p^x, m) \geq TotalCost(p^{x+1}, m)$, for all $x \geq 0$.*

Proof. Let J_j^{qx} be the j^{th} best choice of index set for query q among index sets $\{I_1^x, \ldots, I_n^x\}$. Then by definition of *TotalCost*, we obtain:

$$TotalCost(p^x, m) = \sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, J_j^{qx}) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i^x) \tag{3.6}$$

$$TotalCost(p^{x+1}, m) = \sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, J_j^{q(x+1)}) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i^{x+1}) \tag{3.7}$$

Because the algorithm ensures that in p^{x+1} each query is part of exactly m replicas, we can rearrange those queries by assigning them to partitions as follows:

- For each copy of query q that belongs to some partition in p^{x+1} where it also has its j^{th} best cost (for any $j \in [1, m]$), we assign query q to this partition. The index set of the corresponding partition is $J_j^{q(x+1)}$ (i.e., the j^{th} best index set for query q).

- Otherwise, there is at least one copy of q that does not belong in any partition in p^{x+1} used to create any of its m best index sets. Each such copy of q is assigned to a partition where it belongs as long as no other copy of q is also assigned to that partition. This assignment is always possible because there are exactly m such copies and m partitions where they belong. Note that the cost of q using the index set created by its assigned partition is larger than or equal to any of its m best choices.

Let us replace each index set $J_j^{q(x+1)}$ in $TotalCost(p^{x+1}, m)$ (given by Equation 3.7) by the index set produced by the partition in p^{x+1} where q is assigned to. If we denote $I_j^{q(x+1)}$ the index set that was created by the j^{th} partition where q belongs to then Equation 3.7 can rewritten as follows:

$$TotalCost(p^{x+1}, m) \leq \sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, I_j^{q(x+1)}) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i^{x+1}) \tag{3.8}$$

By regrouping the sums in Equation 3.8, we can create costs per group of queries and updates where each group is a partition in p^{x+1} and each cost is calculated with the corresponding index sets of the partition. In particular,

$$\sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, I_j^{q(x+1)}) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i^{x+1}) = \sum_{i \in [1,n]} SingleCost(W_i^{x+1}, f', I_i^{x+1}) \tag{3.9}$$

where $f'(q) = \frac{f(q)}{m}$ and $f'(u) = f(u)$. Hence, from Equations 3.8 and 3.9, we get that:

$$TotalCost(p^{x+1}, m) \leq \sum_{i \in [1,n]} SingleCost(W_i^{x+1}, f', I_i^{x+1}) \tag{3.10}$$

By definition of the optimal design advisor, we know that $SingleCost(W_i^{x+1}, f', I_i^{x+1})$ is the smallest cost possible comparing to using any other possible index set because index set I_i^{x+1} is created (W_i^{x+1}, f') (i.e., $I_i^{x+1} = DBAdv(W_i^{k+1}, f', b)$). So if we would replace the index sets in the above sum with any other index set we could not decrease the resulting sum of costs. Next, we will replace I_i^{x+1} by I_i^x and as explained above, the total cost has to either increase or stay the same. Recall that I_i^x is the index set created for workload W_i^x. More formally, by Equation 3.10 and the fact that $SingleCost(W_i^{x+1}, f' I_i^{x+1}) \leq SingleCost(W_i^{x+1}, f', I_i^x)$, we get that:

$$TotalCost(p^{x+1}, m) \leq \sum_{i \in [1,n]} SingleCost(W_i^{x+1}, f', I_i^x) \tag{3.11}$$

Let Q_i^{stay} be the set of queries that belong to both W_i^x and W_i^{x+1} (i.e., the ones that will stay in that workload partition). Let Q_i^{move} be the set of queries that do not belong in W_i^x but they do belong in W_i^{x+1} (queries in Q_i^{move} "moved into" the workload partition producing index sets for replica i).

Since queries in Q_i^{stay} did not "move" but instead they stayed in the partition used for index set of replica i, that means that their j^{th} (for some $j \in [1, m]$) smallest cost was using index set I_i^x and as a result, $I_i^x = J_j^{qx}$ for all $q \in Q_i^{stay}$. Queries in

Q_i^{move} will "move" during iteration x into the partition used for the index set of replica i. This only happens by the algorithm if they have their j^{th} smallest cost given I_i^x for some $j \in [1, m]$. Hence, $I_i^x = J_j^{qx}$ for some $j \in [1, m]$ and for all $q \in Q_i^{move}$. Hence by Equation 3.6, the above arguments, and because $f'(q) = \frac{f(q)}{m}$ we conclude that: $\sum_{i \in [1,n]} SingleCost(W_i^{x+1}, f', I_i^x) = TotalCost(p^x, m)$ and by equation 3.11, we get that $TotalCost(p^{x+1}, m) \leq TotalCost(p^x, m)$ and the theorem follows. □

This theoretical result is interesting, since it relies solely on the optimality of $DBAdv()$ and does not make further assumptions about this black-box function. In practical terms, where $DBAdv()$ is imperfect but still identifies effective configurations, we have observed that each iteration yields a design p^{x+1} whose performance is either significantly better or close to p^x. (The details appear in Section 3.4.2.) Overall, the theoretical and empirical evidence suggest that the algorithm converges to a good design.

The convergence condition (line 17) checks that the $TotalCost()$ value of the new design does not differ from the previous design beyond a threshold ϵ, which is a parameter of the algorithm. Intuitively, this is a point where the algorithm has discovered a good divergent design, and subsequent iterations refine this design with diminishing returns. However, certain input instances may cause the algorithm to terminate only after an exponential number of iterations. To avoid such problematic cases, the convergence condition also imposes an upper bound on the number of iterations, based on a second parameter $IterMax$. Clearly, the two parameters control the trade-off between total execution time of the algorithm and performance of the output partition p. Our

experimental study indicates $\epsilon = 1.0\%$ and $IterMax = 10$ yields a good balance point in practice.

To improve the chances of discovering a good divergent design, we invoke the algorithm several times, each time starting with a different initial design. In this fashion we collect several designs and return the one with the minimum $TotalCost()$ value.

An important detail of DivgDesign is the choice of the initial partition p^0. In the current definition p^0 is set as a random partition (for each restart). There may be other methods of higher overhead and perhaps of higher potential to improve the final output which are worth investigating as future work. Another important point is that the design p^{x+1} may have sub-partitions that do not have any queries. This can happen in the generation of $W_1^{x+1}, \ldots, W_n^{x+1}$ at the end of the iteration, if no query in a previous part W_i^x has I_i^x among its best m configurations (line 12). Such designs are not desirable, as they reduce the effective number of replicas for query processing. Our current implementation of DivgDesign simply aborts the current iteration if this happens and restarts the algorithm with a different random design. Our experiments (Section 3.4) show that random initialization and restarts work well in practice.

3.4 Experimental Study

This section presents the results of an experimental study that we conducted in order to evaluate the effectiveness of the DivgDesign algorithm and the performance of the divergent designs that it computes. We first discuss the methodology we followed

Parameter	Values
n	2, 3, 4
m	1, 2
Space budgets	0.25×, 0.5×, 1.0×, 2.0×, INF (infinite)

Table 3.2: Summary of experimental parameters.

for the experiments and then present our findings.

3.4.1 Methodology

Data Sets and Workloads. We employ data sets and workloads from the standard TPC analytical decision-support benchmarks [95] TPC-H and its successor TPC-DS. These benchmarks comprise analytic workloads consisting of 22 and 99 query templates respectively. As a newer benchmark, TPC-DS is intended to model business reporting queries for a modern *data warehouse* and thus includes more diverse and complex queries compared to TPC-H.

For both benchmarks, we create 10GB databases and corresponding query workloads using the supplied (data and query) generators with the default parameters. Each workload is created as a single query stream with all frequencies set to one, thus assigning equal weight to each template. For TPC-H, we create a workload with updates (termed TPC-H-u) using the 22 queries with *refresh* streams RF1 and RF2 at the rate of 0.1% as prescribed by the benchmark. These streams contain inserts and deletes respectively that are applied to both the LINEITEM and ORDERS tables. We create an additional update workload (termed TPC-H-u110) that has an increased proportion of queries to updates. Specifically, we concatenate five streams of 22 queries each, in

addition to the refresh streams RF1 and RF2 mentioned above. We do not consider update workloads for TPC-DS, as their implementation according to the benchmark is quite complicated.

During our evaluation, we observed that TPC-DS queries 4 and 11 were so expensive that they effectively obscured all other queries in the workload. It appeared that the the DB2 Design Advisor was unable to recommend good indexes to improve those two queries, therefore their costs would remain the same in a uniform or divergent design. Moreover, each query of the two had a cost that exceeded the combined costs of the remaining 97 queries. For these reasons we exclude those two queries from our TPC-DS workload.

Experimental Parameters. Our experiments vary the following parameters: number of replicas n, load balancing factor m, and index configuration space budget b. Table 3.2 shows the parameter values that we use in the experiments. The storage space budget is measured as a multiple of the base data size, i.e., given our 10 GB base data size, a space budget of 1.0× indicates a 10 GB storage space budget.

DivgDesign Implementation. We completed a prototype implementation of DivgDesign over IBM DB2 (Express-C version 9.7 for 64-bit Linux). As presented in Section 3.3.2, DivgDesign utilizes the system's design advisor (IBM DB2 Design Advisor, in this case) to compute per-replica configurations, and the what-if optimizer to estimate query costs under these configurations. In our implementation, we invoked the DB2 Design Advisor to compute index-only configurations. We note that our methods are applicable for materialized views as well. Since DivgDesign treats the *DBAdv*

and what-if query optimizer as black boxes, our implementation can be easily ported to another database system with these features.

We set the convergence parameters of DivgDesign to $\epsilon = 1\%$ and $IterMax = 10$. We actually performed minimal tuning for these settings, and therefore they are not particularly tailored to the workloads that we employ. Given input parameters W, f, b, n, and m, we run the DivgDesign algorithm five times and output the lowest cost design found by all five independent runs (see also Section 3.3.2). We denote this final design as p_{divg}.

Metrics. We measure the performance of the divergent design p_{divg} (computed by DivgDesign) primarily through the value of the total cost metric $TotalCost(p_{divg}, m)$. We compute this metric by invoking the what-if optimizer to estimate the costs of queries and updates under the configurations implied by p_{divg}. This methodology, which is consistent with previous studies on physical design tuning, allows us to gauge the effectiveness of DivgDesign in isolation from any estimation errors in the optimizer's cost models. In several cases, we report the improvement of $TotalCost(p_{divg}, m)$ compared to $TotalCost(p_{unif}, m)$, which measures the performance of the uniform design (the current practice in real-world systems).

We also perform several experiments where we measure the wall-clock time to execute workloads using the configurations corresponding to p_{divg} (again, with IBM DB2 as the DBMS). In these experiments, we actually build the indexes corresponding to each replica. To ensure statistical robustness, we repeat these experiments three times and report the average execution time.

Experimental Platforms. In all experiments, DivgDesign and IBM DB2 run on Ubuntu Linux 10.04 *LTS* inside a VMWare virtual machine. The virtual machine infrastructure provides the convenience of setting up identical environments over different host machines. We employ two experimental platforms for the host machines. All experiments that measure the $TotalCost(p_{divg}, m)$ metric run in a single host machine with a dual-core Intel 2.6GHz CPU and 8GB of memory. All experiments that measure wall-clock time run on a local cluster of host machines, each having two dual-core AMD 2.0GHz CPUs and 8GB of RAM. The host machines in the local cluster are connected with a gigabit network and switch. In both platforms, the virtual machine running IBM DB2 is assigned 2 cores and 4GB of RAM.

3.4.2 Behavior of the DivgDesign Algorithm

The first set of experiments examines the behavior of DivgDesign and the characteristics of the divergent designs that it computes.

Iterative improvement in DivgDesign. We first examine how each iteration of DivgDesign improves the currently computed design. Theorem 3.3.1 states that each iteration cannot lead to a worse design if the DBMS advisor is optimal, so our goal is to examine the validity of this result under the imperfect DB2 Design Advisor. For this experiment, we employ the TPC-DS workload with $b = 0.25$, $n = 4$ and $m = 1$, but we obtained similar results for other values.

Figure 3.3 shows the $TotalCost(p, m)$ metric at each iteration of the algorithm, where p is the design at the end of the iteration. Each of the five curves represents an

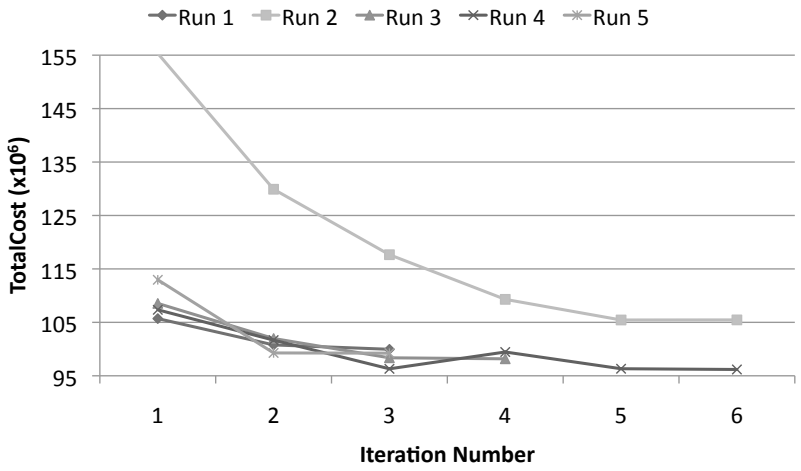

Figure 3.3: Five independent runs of DivgDesign.

independent run of DivgDesign with a different initial (random) partition. (Recall that p_{divg} is the best design out of these five runs, which is run 4 in this example.)

The results show that most iterations successfully improve the currently computed design, which matches the theoretical result of Theorem 3.3.1 even though the advisor is imperfect. In fact, run 4 shows a slight upward trend at the fourth iteration, which is due precisely to this imperfection. Still, DivgDesign was able to recover from this setback in subsequent iterations, and this run resulted in the best design overall.

In all the experiments that we present, DivgDesign converged to a solution within six iterations, well before the *IterMax* cutoff of ten iterations. This also had a beneficial effect on running time. As an example, for the TPC-DS workload of the previous experiment (97 queries), DivgDesign required around three minutes per iteration, leading to a maximum of 18 minutes per run. We observed similar run times for all other experiments.

Diversity of output design p_{divg}. Next, we evaluate the diversity of the divergent design p_{divg} computed by DivgDesign. Diversity is a primary differentiator for divergent designs and one of the key motivations behind their usage. Since our prototype implementation invokes the DB2 Design Advisor to compute index-only configurations, we measure the diversity of p_{divg} in terms of the number of distinct indexes contained in the corresponding configurations $I_1, \ldots, I_n$. As a yardstick for comparison, we also measure the number of distinct indexes in the configuration I_{unif} of the uniform design p_{unif} for the same number of replicas. Clearly, our expectation is that p_{divg} should contain a significant number of additional indexes compared to p_{unif}.

Table 3.3 reports the total count of distinct indexes for p_{divg} and p_{unif}, for input instances with $m = 1$. We select $m = 1$ because it corresponds to maximum specialization and hence maximal diversity. As shown, the number of additional indexes varies across the experiments, but overall the results demonstrate that p_{divg} contains significantly more indexes compared to p_{unif}. The value of these additional indexes will become evident in the experiments that follow. For a few cases we examined the indexes in p_{unif} that do not appear in p_{divg} and found that they were mostly indexes on smaller tables. Somewhat surprisingly, p_{divg} has a higher diversity even for the infinite space budget, but this is due to the imperfection of the DB2 Design Advisor.

3.4.3 Performance for Query-Only Workloads

The next set of experiments evaluates the performance of p_{dvg} on query-only workloads based on the *TotalCost*() metric. Results with mixed workloads of queries

	Space Budget				
	0.25×	0.5×	1.0×	2.0×	INF
TPC-H, $n = 2$					
p_{divg}	27	31	39	42	42
p_{unif}	22	23	35	40	42
TPC-H, $n = 4$					
p_{divg}	35	39	42	43	43
p_{unif}	22	23	35	40	42
TPC-DS, $n = 2$					
p_{divg}	138	165	191	229	275
p_{unif}	119	134	154	185	269
TPC-DS, $n = 4$					
p_{divg}	172	197	237	275	279
p_{unif}	119	132	152	184	268

Table 3.3: Number of distinct indexes in the configurations of p_{unif} and p_{divg}.

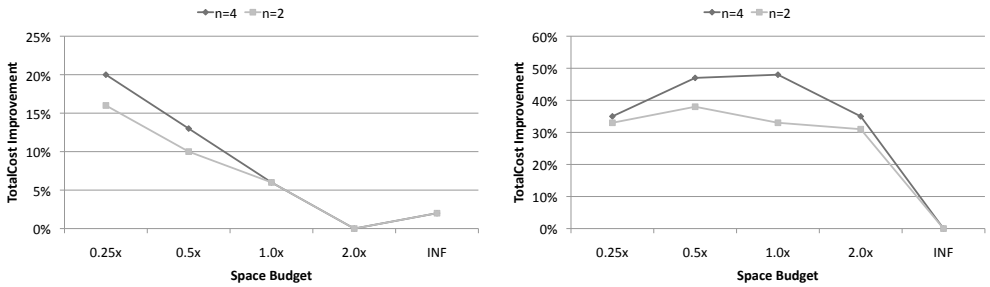

Figure 3.4: Performance improvement of divergent design with $m = 1$ for (a) TPC-H query-only workload and (b) TPC-DS query-only workload.

and updates appear in the following sub-section.

Figure 3.4(a) shows the performance of p_{divg} as we vary the space budget and the number of replicas, for $m = 1$ and the TPC-H workload. Figure 3.4(b) shows the same results for TPC-DS. In both cases, we report the improvement of $TotalCost(p_{divg}, m)$ over $TotalCost(p_{unif}, m)$, where p_{unif} is again the baseline uniform design for the same number of replicas. The results demonstrate a wide range of gains with several interesting trends. The largest performance improvements for TPC-H are

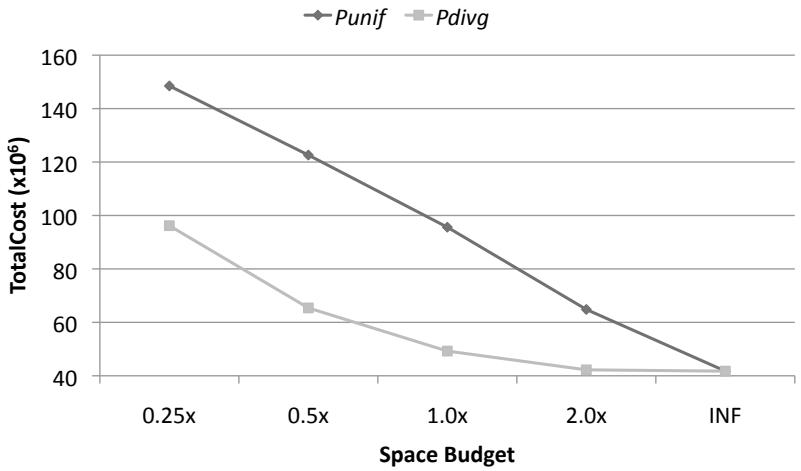

Figure 3.5: Performance of divergent design and uniform design for TPC-DS query-only workload, $m = 1$, $n = 4$.

16% and 20% for $n = 2$ and $n = 4$ respectively at the 0.25× space budget. TPC-DS shows a 38% improvement for $n = 2$ at 0.5× and 48% improvement for $n = 4$ at 1.0× space budgets.

The general trend is that improvements increase at lower space budgets and also with a higher number of replicas. A divergent design makes better usage of the aggregate disk space for configurations, and can thus install more indexes than p_{unif} even at low space budgets (see also Table 3.3). Accordingly, as n increases, a divergent design can specialize each replica to a smaller subset of the workload, thus yielding better performance. It is interesting to note that the gains are consistently higher for TPC-DS than for TPC-H. This is because the TPC-DS query workload is much more diverse and hence has a far greater number of beneficial indexes (see again Table 3.3); thus the divergent design is able to partition the workload and distribute the indexes more effectively.

The performance gains become smaller as the space budget grows, because at some point there is enough space to materialize all the beneficial indexes. We observed this to be near 2.0× for TPC-H and 7.5× for TPC-DS. An exception to this observation is the slight gain for the INF budget in Figure 3.4(a), which however can be attributed to the imperfect heuristics employed by the advisor. Overall, the improvements shown at each space budget appear to match well with the proportion of additional distinct index counts shown in Table 3.3.

Figure 3.5 shows in detail the performance of p_{divg} and p_{unif} for the TPC-DS workload and $n = 4$. Here we chart the *TotalCost*() metric of each design. We observe that the cost of the divergent design decreases rather smoothly with the larger space budgets as expected, exhibiting the same stable behavior as the uniform design. More importantly, we observe the following interesting pattern: the divergent design with a space budget of 0.25× performs nearly the same as the uniform design with a 1.0× space budget, and the same holds for the 0.5× divergent design versus the 2.0× uniform design, and for the 1.0× divergent design versus the 5.0× uniform design. In other words, given four replicas, the divergent design for TPC-DS can perform as well as the uniform design with only $\frac{1}{4}$ of the space budget per replica. (We obtained similar results with TPC-H.) This intuitive property validates the effectiveness of DivgDesign in computing a good divergent design, even though the algorithm does not have any visibility in the structure of the workload (which is quite complicated in this experiment) or in the inner workings of the DBMS configuration advisor. Moreover, these results indicate that a divergent design may be particularly beneficial if the configuration space

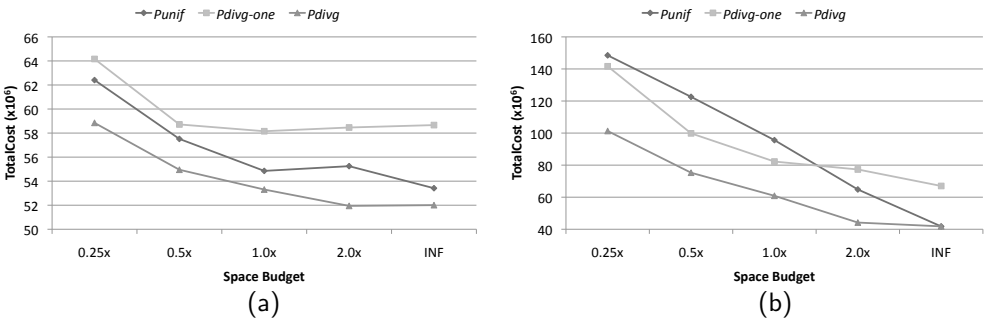

Figure 3.6: Performance of divergent designs for query-only workloads (a) TPC-H with $n = 4$, $m = 2$, (b) TPC-DS with $n = 4$, $m = 2$.

budget is restricted.

Our last set of experiments evaluates the effect of parameter m on the performance of the divergent designs computed by DivgDesign. To illustrate the trade-off between performance and load-balancing, we consider one additional design in these experiments that we denote as p^{one}_{divg} and which is the design computed by DivgDesign for $m = 1$. Clearly, p^{one}_{divg} has higher specialization than p_{divg} and can thus yield better performance if every query q can be routed to its best replica. On other hand, q may have a much higher cost on its 2nd-best replica in p^{one}_{divg}, which implies less flexibility in terms of load balancing.

Figure 3.6(a) shows the *TotalCost* metric for designs p_{divg}, p_{unif} and p^{one}_{divg} for the TPC-H workload, $n = 4$, $m = 2$. Figure 3.6(b) shows the same results for TPC-DS. Overall, the results confirm our intuition: p^{one}_{divg} has worse performance than p_{divg}, and in most cases it performs even worse that the uniform design p_{unif}. These results indicate that the specialization in p^{one}_{divg} causes great imbalance for the cost of the same query across different replicas, thus leading to a high $TotalCost()$ metric for $m = 2$. On

the other hand, p_{divg} creates a specialization of replicas that directly takes into account factor m, which in turn yields consistent improvements over the uniform design. The gains are small for TPC-H but very significant for TPC-DS, ranging from 32% to 39% over p_{unif}.

To summarize, our results demonstrate that the divergent designs computed by DivgDesign outperform the baseline uniform design, often by a significant margin. The improvement is more pronounced with the TPC-DS workload, which is somewhat expected given that this workload is more complex than TPC-H and also because IBM DB2 is heavily optimized for the execution of TPC-H queries (as are many commercial systems).

3.4.4 Results with Mixed Workloads

Next we evaluate the performance of divergent designs with mixed workloads of queries and updates. In this case, any design has to balance the benefits of the per-replica configurations against the maintenance cost due to updates. The experiments that follow employ the TPC-H-u and TPC-H-u110 workloads, to evaluate the effect of the query-to-update ratio on the performance of divergent designs. We first present experiments with $m = 1$ (highest specialization) and then consider the effect of $m > 1$.

Figure 3.7(a) shows the percent improvement of p_{divg} over p_{unif}, for the two TPC-H workloads and $m = 1$. The gains are small for TPC-H-u, since the cost of updates leads the DB2 Design Advisor to recommend relatively few indexes for both designs. Conversely, the gains increase with the TPC-H-u110 workload, ranging from

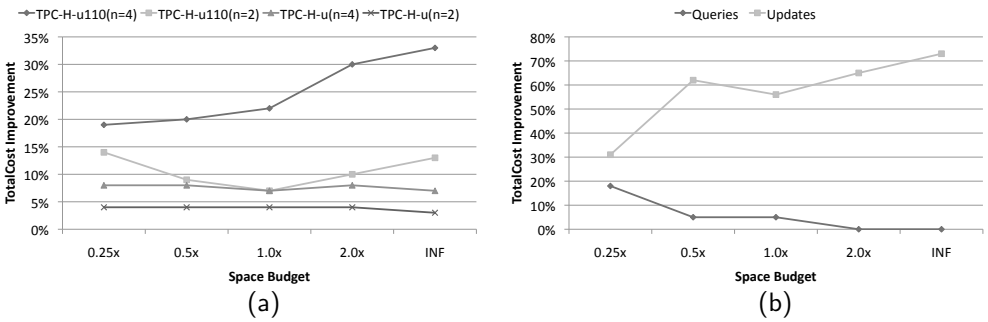

Figure 3.7: (a) Performance of divergent designs for TPC-H with updates and $m = 1$. (b) Percent improvement breakdown for queries and updates in TPC-H-u110, $n = 4$, $m = 1$.

19% to 33% for $n = 4$, since the higher percentage of queries justifies the creation of more indexes in the divergent design. We also observe an increase in gains as n grows, which is in accordance to the trend for query-only workloads.

Figure 3.7(b) shows a breakdown of the improvement for both queries and updates for TPC-H-u110 and $n = 4$. This breakdown provides further insights into the performance of divergent designs when the workload includes updates. We observe that at $b = 0.25$ query performance is improved by 18% over the uniform design, but for all space budgets larger than 0.25×, divergent design offers virtually no performance improvement over the uniform design. However, Figure 3.7(b) clearly shows that the big difference is in the cost of updates, where divergent design's improvement ranges from 30% to over 70%. The reason can be traced again to the asymmetry between queries and updates — a query can be executed by any single replica, whereas an update must be executed by all replicas. Intuitively, the cost to apply the update is multiplied by n. The uniform design ignores this magnified effect and results in indexes whose maintenance cost is much higher than their benefits. In contrast, the divergent design p_{divg} takes

Design	Table Name	#Indexes per Replica			
p_{unif}	ORDERS	5	5	5	5
	LINEITEM	7	7	7	7
p_{divg}	ORDERS	2	2	1	1
	LINEITEM	5	2	2	4

Table 3.4: Index counts of the ORDERS and LINEITEM tables per replica for p_{unif} and p_{divg} for TPC-H-updates-110, $n = 4$, $m = 1$, $b = 1.0$.

this asymmetry directly into account, and results in indexes that strike a better balance between benefit and maintenance cost.

Table 3.4 shows the number of indexes in p_{divg} and p_{unif} for $b = 1.0$ in the previous experiment, for the ORDERS and LINEITEM tables that receive all the updates. The reported counts support our previous observation. The divergent design places far fewer indexes across the replicas, which in turn leads to reduced maintenance cost (over 50% reduction, as shown in Figure 3.7(b)). Moreover, in this experiment we verified that p_{divg} comprises *all* the indexes of p_{unif}, except that they are distributed across different replicas. It is interesting to note that we observed similar results when we examined p_{divg} and p_{unif} for $b = INF$: the divergent design installed a total of 20 indexes, compared to 72 indexes for the uniform design. As shown in Figure 3.7(b), this allocation results in 70% improvement in index maintenance cost while retaining the same benefits for query evaluation. Thus, p_{divg} results in a more judicious allocation of indexes even when the per-replica configurations are not constrained by the space budget.

Additional experiments considered the effect of m on the performance of the divergent design computed by DIVGDESIGN. We experimented with $m = 1$ and $m = 2$,

using the same methodology as for query-only workloads. We observed that the results remained qualitatively the same and hence are omitted.

Overall, our experimental results demonstrate the divergent designs computed by DivgDesign continue to outperform the baseline uniform design. The improvement becomes more significant when the ratio of queries increases — most of the gains come from a judicious allocation of indexes to replicas, which substantially reduces the total maintenance cost.

3.4.5 Performance with Concurrent Execution

We conclude with a set of experiments that evaluate the performance of divergent designs in terms of query throughput. Our experimental setup is a replicated TPC-H database with $n = 3$ replicas, using IBM DB2 as the DBMS. We create several concurrent client processes, each executing a random permutation of a subset of the TPC-H workload (to be defined shortly). We tune the system according to a specific design, and then measure performance in terms of the total wall-clock time to evaluate the query streams of the concurrent clients. Hence, the execution time of the slowest client determines the overall performance.

We invoke DivgDesign to compute a divergent design p_{divg} with $b = 1.0$ and $m = 2$, so that there is some flexibility to load balance across the $n = 3$ replicas while maintaining some specialization. As in the previous experiments, we will compare p_{divg} against the uniform design p_{unif} for the same values of b and n.

We select the workload in order to favor the execution of queries under p_{unif}.

	15 clients	30 clients	45 clients
TPC-H	11.5%	20.1%	19.5%
TPC-DS	15.8%	23.3%	31.5%

Table 3.5: Total execution time improvement of divergent design over uniform design for 17 TPC-H and 20 TPC-DS Queries, $n = 3, m = 2$, $b = 1$, for 15, 30, and 45 concurrent clients.

Specifically, out of the 22 TPC-H queries, we select a subset of 17 queries that have similar[1] execution times under p_{unif}. This property allows us to use a simple query routing policy that is likely to work well for p_{unif}: When some client wishes to execute some query q, it routes q to the replica which currently has the least number of concurrently executing queries. It is not difficult to see that this policy is likely to evenly load each replica under p_{unif}, since each query has the same execution cost on all replicas and all queries have similar execution costs. We employ a slightly modified routing policy for p_{divg}: the client routes q to the replica which currently has the least number of concurrently executing queries and is among the best two replicas for q. The routing policy here follows the intuition of the divergent design (queries are routed to their specialized replicas), but it is not optimal for load balancing because the same query may have different execution costs across different replicas and queries do not necessarily have similar execution costs overall. Overall, we create a best-case scenario for the uniform design and a potentially problematic scenario for the divergent design.

Table 3.5 shows the total improvement of p_{divg} over p_{unif} in terms of wall-clock time as we vary the number of concurrent TPC-H clients. (We discuss the TPC-DS numbers later.) The divergent design consistently outperforms the uniform design, with

[1]Within one standard deviation of the median.

an overall improvement of up to 20% in terms of query throughput. This magnitude is non-trivial for several reasons. First, the divergent design employs a suboptimal query routing policy that does not take into account the actual load of each replica. Second, the predicted improvement according to *TotalCost*() is only 3%, which means that the non-linear effects of concurrent query execution magnify the benefits of divergent designs. Third, $m = 2$ implies that each query must have low execution cost on at least two out of the $n = 3$ replicas, limiting the specialization a divergent design can achieve. Finally, IBM DB2 (as many commercial systems) is heavily optimized to execute these benchmark workloads. Is it very significant that we are able to improve performance by 20% by simply tuning the configuration of each replica.

In addition to improved execution time, the divergent design brings clear benefits to the time required to tune the system. Specifically, the divergent design required 50% less time to build the corresponding indexes on all replicas, since fewer indexes are allocated per replica while maintaining similar query benefits. (This is analogous to the results we saw in Table 3.4.) Accordingly, the time to refresh the optimizer's statistics[2] after the indexes are built is reduced by 14% . All in all, if we count the end-to-end time to build the indexes, update the optimizer's statistics, and execute the workload through the concurrent clients, the divergent design improves upon the uniform by 67% for 15 concurrent clients, and by 52% for 45 concurrent clients.

We performed the same experiment with TPC-DS by first using a similar method to select a 20-query subset, and then executing the full experiment as previously

[2]This corresponds to the RUNSTATS command in DB2.

described. The results in Table 3.5 show a 15.8%, 23.3% and 31.5% improvement for 15, 30, and 45 clients respectively, indicating the same performance trends as the TPC-H experiment.

3.5 Implementation Considerations

The experimental results of the previous section show clearly the performance benefits of divergent design tuning. In what follows, we consider the implications of implementing divergent designs in a DBMS that employs replication. Our assumption is that the DBMS already has the infrastructure to employ the baseline uniform design p_{unif}.

An immediate observation is that the DBMS can readily replace p_{unif} with the fully-balanced design p_{FB}, i.e., the output of the DivgDesign algorithm for $m = n$. As in p_{unif}, p_{FB} prescribes the same configuration for each replica and hence the system can handle query routing and load balancing in exactly the same way. However, our analysis shows that p_{FB} can never perform worse than p_{unif}, and hence there is an immediate benefit to switching to divergent designs.

For $m < n$, the divergent design may assign a different configuration to each replica and hence the query routing policy is affected. Specifically, the DBMS must route an incoming query q to the currently-best replica, taking into account the load in the system and also the cost terms $cost(q, I_i)$ for each replica $i \in [1, n]$. These terms are already known for a query $q \in W$. For an unseen query $q \notin W$, it is possible to perform

what-if optimization or to estimate these cost terms by mapping q to a similar query q' that has already been analyzed (e.g., using the similarity metric of [24] or the general method of [82]).

The use of divergent designs also affects how the system responds to changes in node availability or the intensity of the workload. For instance, the failure of node i causes the queries in W_i to be (temporarily) left with $m-1$ "good" replicas and hence fewer load-balancing choices (until a node brings I_i up again). Similarly, scaling out the system implies that n increases, which likely causes the current divergent design to become suboptimal. In general, examining the interplay between divergent designs and elasticity is an interesting direction for future work.

3.6 Related Work

We next provide a description of related work, which we group by topic area.

Physical design tuning. There has been a long line of research studies on the problem of tuning the physical design of a single, centralized RDBMS [15, 25, 34, 57, 102]. The proposed methods typically take a representative workload W as input, and after some analysis they recommend a physical desgin that optimizes the evaluation of the workload according to the optimizer's estimates. However, the proposed methods work for a centralized system and do not take replication into account. Extending them to the setting of a replicated database is not obvious. Still, we show it is possible to leverage existing tools to compute good divergent designs, provided that we can compute an

effective partition of the workload.

Chaudhuri et al. [24] propose a method to cluster the queries of a workload into disjoint subsets for the purpose of workload compression. Clustering is based on (an approximation of) a pairwise-query distance function that takes into account the similarity of the optimal indexes for the two queries. Unfortunately, the partitioning generated by [24] produces an unspecified (and likely large) number of disjoint subsets, and hence cannot be directly applied to solve the divergent index tuning problem where the number of clusters must be equal to the (likely small) number of replicas. Additionally, the methods in [24] do not consider key factors for divergent design tuning such as the space budget constraint, the load-balancing factor m, or the *TotalCost* metric.

Shinobi [97] is a system that utilizes workload information to partition the data and selectively index data within each partition. This results in less expensive index maintenance and reorganization costs, by creating and dropping indexes on subsets of the data (the workload-based partitions) as the access patterns change. However, Shinobi does not address replication of partitions or different index configurations (i.e., physical design) on replicas of the same partition, which is the problem that we examine in our work. In fact, our techniques could be used to determine which indexes to install on each replica, and then Shinobi can be responsible for maintaining only the fragments of these indexes that are important for the current workload patterns.

Physical data organization on replicas. Previous works also considered the idea of diverging the physical organization of replicated data. The technique of Fractured

Mirrors [80] builds a mirrored database that stores its base data in a different physical organizations on disk (specifically, in a row-based and a column-based organization). To take advantage of this storage model, the query processor is modified to run query execution plans that can work on both formats of the data. While the physical data layout differs on both replicas, they do not consider different indexing. Similarly, Distorted Mirrors [90] presents logically but not physically identical mirror disks for replicated data. However, none of the above methods explore different physical organizations of indexes and materialized views for each replica.

Cloud database services. Numerous new designs and services for cloud-based databases have recently become available. We describe the relevant characteristics of a sample of such systems that employ replication and can benefit directly from our techniques.

Microsoft Cloud SQL Server [12] (MS Azure) partitions data by row groups, and partitions are the unit of replication. It uses a modified version of primary copy replication, and the primary copy processes all transactions for its row group, then asynchronously ships updates to replicas. To address load balancing, each server hosts 1 primary partition for each $x - 1$ secondary partitions; however, secondary partitions are not involved in active query processing and hence do not include the ability for their index configurations to diverge.

Amazon Relational Database Service (RDS) [7] has become increasingly available and popular. Amazon RDS currently offers the ability to launch many database

instances using Amazon Machine Images (AMIs) preconfigured with standard database systems such as MySQL, Oracle, and IBM DB2. The Amazon Read Replicas service allows up to 5 full (read-only) replicas of a MySQL database to be launched on-demand which may be used as fail-over databases, or to provide scalability as workloads change. Replicas may also be located in different data centers; an approach used to mitigate regional outages. While many RDBMSs provide database replication, these represent identical replicas and do not include the ability for their index configurations to diverge.

Chapter 4

Opportunistic Physical Design for Big Data Analytics

Data analysts have the crucial task of analyzing the ever increasing volume of data that modern organizations collect in order to produce actionable insights. As expected, this type of analysis on big data is highly exploratory in nature and involves an iterative process: the data analyst starts with an initial query over the data, examines the results, then reformulates the query and may even bring in additional data sources, and so on. Typically, these queries involve sophisticated, domain-specific operations that are linked to the type of data and the purpose of the analysis, e.g., performing sentiment analysis over tweets or computing network influence. Because a query is often revised multiple times in this scenario, there can be significant overlap between queries. There is an opportunity to speed up these explorations by reusing previous query results either from the same analyst or from different analysts performing a related task.

MapReduce (MR) has become a primary tool for this type of analysis. It offers scalability to large datasets, easy incorporation of new data sources, the ability to query right away without defining a schema up front, and extensibility through user-defined functions (UDFs). Data analysts often write their queries in a declarative query language, e.g., HiveQL [93] or PigLatin [73], which are automatically translated to a set of MR jobs. Each MR job involves the materialization of intermediate results (the output of mappers, the input of reducers and the output of reducers) for the purpose of failure recovery. A typical Hive or Pig query will spawn a multi-stage job that will involve several such materializations. We refer to these execution artifacts as *opportunistic materialized views*.

We propose to treat these views as an *opportunistic physical design* and to use them to rewrite queries. The opportunistic nature of our technique has several nice properties: the materialized views are generated as a by-product of query execution, i.e., without additional overhead; the set of views is naturally tailored to the current workload; and, given that large-scale analysis systems typically execute a large number of queries, it follows that there will be an equally large number of materialized views and hence a good chance of finding a good rewrite for a new query. Our results indicate the savings in query execution time can be dramatic: a rewrite can reduce execution time by up to an order of magnitude.

Rewriting a query using views in the context of MapReduce with opportunistic views involves a unique combination of technical challenges that distinguish it from the traditional problem of query rewriting. First, the queries and views almost certainly

contain UDFs; thus query rewriting requires some *semantic* understanding of MapReduce UDFs, which are composed of arbitrary user-code and may involve a sequence of MR jobs. Second, any query rewriting algorithm that can utilize UDFs now has to contend with a potentially large number of operators since any UDF can be included in the rewriting process. Third, due to our opportunistic design policy of retaining all materializations (storage permitting), there can be a large number of views in the system and hence the search space of views to consider for query rewriting can grow very large.

Recent methods to reuse MR computations such as ReStore [37], [65], and MRShare [72] lack any semantic understanding of execution artifacts and can only reuse/share cached results when execution plans are syntactically identical. We strongly believe that any truly effective solution will have to a incorporate a deeper semantic understanding of cached results and "look into" the UDFs as well.

In this work, we present a novel query-rewrite algorithm that targets the scenario of opportunistic materialized views in an MR system with queries that contain UDFs. To facilitate this, we first propose a gray-box UDF model that has a limited semantic understanding of UDFs, yet enables effective reuse of previous results. Our model captures a large class of MR UDFs that includes many common analysis tasks. The UDF model further provides a quick way to compute a lower-bound on the cost of a potential rewrite given just the query and view definitions. We next propose a rewrite algorithm which employs techniques inspired by spatial databases (specifically, nearest-neighbor searches in metric spaces [49]) in order to provide a cost-based incremental

enumeration of the large space of candidate rewrites, generating the optimal rewrite in an efficient manner. The algorithm uses the lower-bound cost to incrementally explore the space of rewrites by (a) gradually growing the space of rewrites as needed, and (b) only attempting a rewrite (an expensive process) for those views with a good potential to produce a low-cost rewrite. We show that our algorithm produces the optimal rewrite as well as finds this rewrite in a work-efficient manner, under certain assumptions.

The outline of this chapter is as follows. Section 4.1 describes our system architecture and introduces our notation. Sections 4.2–4.3 provide the UDF model, the types of UDFs it admits, and our cost model. Sections 4.5–4.6 describe our rewrite algorithm BfRewrite and its use of the lower-bound to incrementally explore the space of rewrites. Section 4.7 provides our experimental evaluation using a workload of big data queries that contain UDFs. Our experiments evaluate the quality of the rewrites in terms of query execution time improvement, the scalability of our rewrite algorithm, the effectiveness of our cost model, and the sensitivity of our algorithm to the degree of similarity among queries in the workload.

4.1 Preliminaries

Here we present the architecture of our system. We first briefly describe its components and how they interact with one another. We then provide the notations and problem definition.

4.1.1 System Architecture

Figure 4.1 provides a high level overview of our system and its components. Our system is built on top of Hive, and queries are written in HiveQL. Queries are posed directly over log data stored in HDFS. In Hive, MapReduce UDFs are given by the user as a series of Map or Reduce jobs containing arbitrary user code expressed in a supported language such as Java, Perl, Python, etc. To reduce execution cost, our system automatically rewrites queries based on the existing views. A query execution plan in Hive consists of a series of MR jobs, and each MR job materializes its output to HDFS. As Hive lacks a mature query optimizer and cannot cost UDFs, we implemented an optimizer based on the cost model from [72] and extended it to cost UDFs, as described later in Section 4.3.2.

During query execution, all by-products of query processing (i.e., the intermediate materializations) are retained as opportunistic materialized views. These views are stored in the system (space permitting) as the opportunistic physical design.

The materialized view metadata store contains information about the materialized views currently in the system such as the view definitions and standard data statistics used in query optimization. For each view stored, we collect statistics by running a lightweight Map job that samples the view's data. This constitutes a small overhead, but as we show experimentally in Section 4.7, this time is a small fraction of query execution time.

The rewriter, presented in Section 4.5, uses the materialize view metadata store

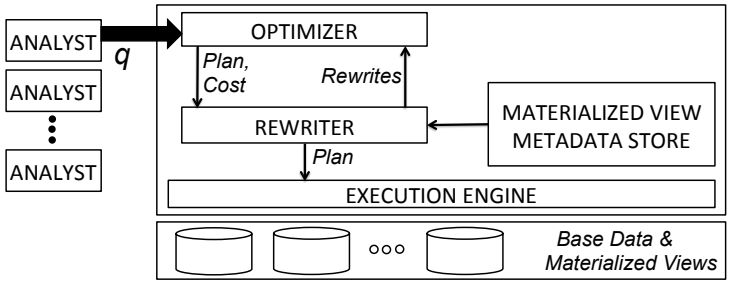

Figure 4.1: System diagram showing control flows.

to rewrite queries based on the existing views. To facilitate this, our optimizer generates plans with two types of annotations on each plan node: (1) the logical expression of its computation (Section 4.2.2) and (2) the estimated execution cost (Section 4.3.2).

The rewriter uses the logical expression in the annotation when searching for rewrites for each node in the plan. The expression consists of relational operators or UDFs. For each rewrite found during the search, the rewriter utilizes the optimizer to obtain an estimated cost for the rewritten plan.

4.1.2 Notations

P denotes a plan generated by the query optimizer for an input query q. P is represented as a DAG containing n nodes, ordered topologically. Each node represents an MR job, and we denote the i^{th} node of P as NODE_i, $i \in [1, n]$, under the topological ordering. An outgoing edge from NODE_k to NODE_i represents data flow from k to i. The plan has a single sink that computes the result of the query. Under the topological order assumption the sink is NODE_n. P_i is a sub-graph of P containing NODE_i and all of its ancestor nodes. We refer to P_i as one of the rewritable *targets* of plan P. Following

standard MapReduce semantics, we assume that the output of each job is materialized to disk. Hence, a property of P_i is that it represents a materialization point in P. In this way, materializations are free except for the cost of statistics collection and their overhead in terms of the additional disk storage consumed. V is the set of all opportunistic materialized views in the system.

We use $\text{COST}(\text{NODE}_i)$ to denote the cost of executing the MR job at NODE_i, as estimated by the query optimizer. Similarly, $\text{COST}(P_i)$ denotes the estimated cost of running the sub-plan rooted at P_i, which is computed as $\text{COST}(P_i) = \sum_{\forall \text{NODE}_k \in P_i} \text{COST}(\text{NODE}_k)$.

We use r_i to denote an equivalent rewrite of target P_i *iff* r_i uses only views in V as input and produces an identical output to P_i for a given database instance D. A rewrite r^* represents a rewrite among the class of rewrites whose cost is equivalent to the minimum cost rewrite of P (i.e., target P_n).

4.1.3 Problem Definition

Given these basic definitions, we introduce the problem we solve in this work.

Problem Statement. *Given a plan P for an input query q, and a set of materialized views V, find the rewrite r^* of P.*

Our rewrite algorithm considers views in V during the search for r^*. In order for the rewriter to utilize all views in V during its search, some semantic understanding of UDFs is required as some views may contain results generated by UDFs. Next we will describe our semantic UDF model and then present our rewrite algorithm that solves

this problem.

4.2 UDF Model

Since big data queries frequently include UDFs, in order to reuse previous computation in our system effectively we desire a way to model MR UDFs semantically. If the system has no semantic understanding of the UDFs, then the opportunities for reuse will be limited — essentially the system will only be able to exploit cached results when one query applies the exact same UDF to the exact same input as a previous query. However, to the extent that we are able to "look into" the UDFs and understand their semantics, there will be more possibilities for reusing previous results. In this section we propose a UDF model that allows a deeper semantic understanding of MR UDFs. Our model is general enough to capture a large class of UDFs that includes classifiers, NLP operations (e.g., taggers, sentiment), text processors, social network (e.g., network influence, centrality) and spatial (e.g., nearest restaurant) operators. As an example, we performed an empirical analysis of two real-world UDF libraries, Piggybank [78] and DataFu [67]. Our model captures about 90% of the UDFs examined: 16 out of 16 Piggybank UDFs, and 30 out of 35 DataFu UDFs as we detail in [61]. Of course, we do not require the developer to restrict herself to this model; rather, to the extent a query uses UDFs that follow this model, the opportunities for reuse will be increased.

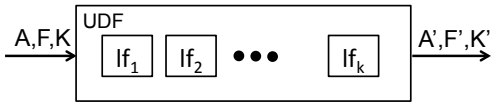

Figure 4.2: A UDF composed of local functions (lf_1, lf_2, $\cdots$, lf_k), showing the end-to-end transformation of input to output.

4.2.1 Modeling a UDF

We propose a model for UDFs that allows the system to capture a UDF as a *composition* of *local functions* as shown in Figure 4.2, where each local function represents a map or reduce task. The nature of the MR framework is that map-reduce functions are stateless and only operate on subsets of the input, i.e., a single tuple or a single group of tuples. Hence, we refer to these map-reduce functions as *local functions*. A local function can only perform a combination of the following three types of operations performed by *map* and *reduce* tasks.

1. Discard or add attributes, where an added attribute and its values may be determined by arbitrary user code

2. Discard tuples by applying filters, where the filter predicates may be performed by arbitrary user code

3. Perform grouping of tuples on a common key, where the grouping operation may be performed by arbitrary user code

The end-to-end transformation of a UDF is obtained by composing the operations performed by each local function *lf* in the UDF. Our model captures the fine-grain dependencies between the input and output tuples in the following way.

The UDF input is modeled as (A, F, K) where A is the set of attributes, F is set of filters previously applied to the input, and K is the current grouping of the input, which captures the keys of the data. The output is modeled as (A', F', K') with the same semantics. Our model describes a UDF as the transformation from (A, F, K) to (A', F', K') as performed by a composition of local functions using operation types (1) (2) (3) above. Figure 4.2 shows how to semantically model a UDF that takes any arbitrary input represented as A, F, K and applies local functions to produce an output that is represented as A', F', K'. Additionally, for any new attribute produced by a UDF (in the output schema A'), its dependencies on the input (in terms of A, F, K) are recorded as a signature along with the unique UDF-name. Note that since the model only captures end-to-end transformations, for a UDF containing multiple internal jobs (e.g., the local functions in Figure 4.2), the system only retains the final output but not the intermediate results of local functions.

The model also captures UDFs that take multiple inputs, which is similar to the single input case shown in Figure 4.2. For example, a UDF that combines 2 inputs on a common key (similar to an equi-join) can be described in the following way. The inputs $\{A_1, F_1, K_1\}$ and $\{A_2, F_2, K_2\}$ produce an output $\{A_J, F_J, K_J\}$ such that: A_J is the union of A_1 and A_2, F_J is the conjunction of the filters F_1, F_2 and the join condition, and K_J is K_1 union K_2 intersected with the join attributes. Note that the model only concerns itself with the end-to-end transformation of the inputs to the outputs, the actual implementation of an operator is not captured by the model.

```
// T1 is input table, T2 is output table
// user_id, tweet_text are input attributes, threshold is a UDF parameter
// sent_sum is an output attribute whose dependencies are recorded
UDF_FOODIES (T1, T2, user_id, tweet_text, threshold) {
    CREATE TABLE T2 (user_id, sent_sum) FROM T1                    (a)
      MAP user_id, text USING "hdfs://udf-foodies-lf1.pl"
                  AS user_id, sent_score  CLUSTER BY user_id
      REDUCE user_id, sent_score, threshold USING "hdfs://udf-foodies-lf2.pl"
                  AS (user_id, sent_sum)
}
UDF model for UDF_FOODIES:                                         (b)
   A={user_id, tweet_text, ...}, F= {f}, K = {k}
   A'={user_id, sent_sum}, F' = {f} ∪{sent_sum > threshold}, K' = {user_id}
   Sig. of new attribute sent_sum = {UDF_FOODIES, user_id, tweet_text,{f},{k}}
```

Figure 4.3: UDF_FOODIES a) implementation composed of two local functions, b) UDF model showing the end-to-end transformation of input to output.

As an example, consider UDF_FOODIES that applies a food sentiment classifier on tweets to identify users that tweet positively about food. An abbreviated HiveQL definition of the UDF is given in Figure 4.3(a) that invokes the following two local functions lf_1 and lf_2 written in a high-level language (Perl in this example). lf_1: For each (`user_id`, `tweet_text`), apply the food sentiment classifier function that computes a sentiment value for each tweet about food. lf_2: For each `user_id`, compute the sum of the `sent_score` values to produce `sent_sum`, then filter out users with a total score greater than `threshold`. Although the filter in this example is a simple comparison operator, as noted above in (2) the filter expression may be a function containing arbitrary user code (i.e., a UDF) and may even contain a nesting of UDFs.

The two local functions correspond to arbitrary user code that perform complex text processing tasks such as parsing, word-stemming, entity tagging, and word sentiment scoring. Yet, the UDF model succinctly captures the end-to-end transfor-

mation of this complex UDF as shown in Figure 4.3(b). In the figure, the end-to-end transformation of `UDF_FOODIES` is captured by recording the changes made to the input A, F and K by the UDF functions that produces A', F' and K' using a simple notation. Furthermore, for the new attribute `sent_sum` in A', its dependencies on the subset of the inputs are recorded. We provide a more concrete example of the application of the UDF model in a HIVEQL query in Section 4.2.2. In this way, the model encodes arbitrary user-code representing a sequence of MR jobs, by only capturing its end-to-end transformations.

Our approach represents a gray-box model for UDFs, giving the system a limited view of the UDF's functionality yet allowing the system to understand the UDF's transformations in a useful way. In contrast, a white-box approach requires a complete understanding of *how* the transformations are performed, imposing significant overhead on the system. While with a black-box model, there is very little overhead but no semantic understanding of the transformations, limiting the opportunity to reuse any previous results.

4.2.2 Applying the UDF Model and Annotations

Having presented our model for UDFs, we now show how to use it to annotate a query plan that contains both UDFs and relational operators. In Figure 4.4(a), we show a query that uses Twitter data to identify prolific users who talk positively about food (i.e., "*foodies*"). The query is expressed in a simplified representation of HiveQL and applies `UDF_FOODIES` from Figure 4.3(a) that computes a food sentiment score (`sent_-`

```
// Extract tweet_id, user_id and tweet_text from twitter log
   CREATE TABLE T1
        SELECT tweet_id, user_id, tweet_text                    (a)
     FROM large_twitter_log;

// Create table T2 containing sent_sum for each user_id
   UDF_FOODIES(T1, T2, user_id, tweet_text, 0.5)

// Join T1 and T2 to produce Result
CREATE TABLE Result
    SELECT T2.user_id, Foo.count, T2.sent_sum FROM T2,
         (SELECT user_id, COUNT(*) AS count FROM T1
                GROUP BY user_id) AS Foo
     WHERE Foo.user_id = T2.user_id AND Foo.count > 100;
```

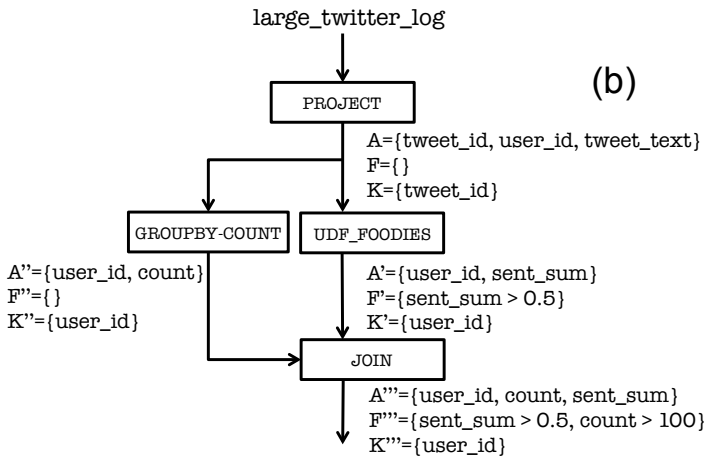

Figure 4.4: (a) Example query to obtain prolific foodies, and (b) corresponding annotated query plan.

`sum`) per user based on each user's tweets.

The HiveQL query is converted to an annotated plan as shown in Figure 4.4(b) by utilizing the UDF model of `UDF_FOODIES` as given in Figure 4.3(b). In addition to modeling UDFs, the three operations types (denoted as 1, 2, 3 above) can also be used to characterize standard relational operators such as select (2), project (1), join (2,3), group-by (3), and aggregation (3,1). Joins in MR can be performed as a grouping of multiple relations on a common key (e.g., co-group in Pig) and applying a filter. Aggregations are a re-keying of the input (reflected in K') producing a new output

attribute (reflected in A'). These A, F, K annotations can be applied to both UDFs and relational operations, enabling the system to automatically annotate every edge in the query plan. Since the queries in our system are posed over base data which is stored as log files and the schema of the logs may not be known, A, F, and K are each initialized to $\emptyset$.

Figure 4.4(b) shows the input to the UDF is modeled as $\langle A$={`user_id`, `tweet_id`, `tweet_text`}, $F=\emptyset$, K=`tweet_id`$\rangle$. The output is $\langle A'$={`user_id`, `sent_sum`}, F'=`sent_sum` > 0.5, K'=`user_id`$\rangle$. `UDF_FOODIES` produces the *new* attribute `sent_sum` whose dependencies are recorded (i.e., signature) as: $\langle A$={`user_id`, `tweet_text`}, $F=\emptyset$, K=`tweet_id`, udf_name=`UDF_FOODIES`$\rangle$. Lastly, as shown in Figure 4.4(b), the output of the UDF (A', F', K') forms one input to the subsequent join operator, which in turn transforms its inputs to the final result. The join operation in Figure 4.4(b) is an equi-join on `user_id`.

This example shows how a query containing a UDF with arbitrary user code can be semantically modeled. The A, F, K properties are straightforward and can be provided as annotations by the UDF creator with minimal overhead, or alternatively they may be automatically deduced using a static code analysis method such as [50], which is an emerging area of research. In this work, we rely on the UDF creator to provide annotations, which is a one-time effort. From our experience evaluating the two UDF libraries [67, 78] it took less than 10 minutes per UDF to examine the code and determine the annotations. We report our empirical evaluation of these UDFs showing the annotations we derived for each UDF in [61]. The annotations for each UDF are

only provided once, i.e, the first time the UDF is added to the system.

As noted earlier, our model is expressive enough to capture a large class of common UDFs. Two classes of UDFs not captured by our model are: (a) non-deterministic UDFs such as those that rely on runtime properties (e.g., current time, random, and stateful UDFs) and (b) UDFs where the output schema itself is dependent upon the input data values (e.g., pivot UDFs, contextual UDFs).

4.3 Using the UDF Model to Perform Rewrites

Our goal is to leverage previously computed results when answering a new query. The UDF model aids us in achieving this goal in three ways, as described in the following three sections. First, it provides a way to check for equivalence between a query and a view. Second, it aids in the costing of UDFs. Third, it provides a lower-bound on the cost of a potential rewrite.

4.3.1 Equivalence Testing

The system searches for rewrites using existing views and can test for semantic equivalence in terms of our model. We consider a query q and a view v to be equivalent if they have identical A, F and K properties; that is, given the logical expressions of q and v as $q = \{A, F, K\}$ and $v = \{A', F', K'\}$ then q and v are said to be equivalent if $A = A'$, $F = F'$, and $K = K'$. If a query and a view are not equivalent, our system considers applying transformations (sometimes referred to as *compensations*) to make the existing view equivalent to the query.

Here we develop the mechanics to test if a query q (i.e., a target in the annotated plan) can be rewritten using an existing view v. Query q can be rewritten using view v if v *contains* q. The query containment problem is known to be NP-Complete [19] even for the class of conjunctive queries, hence we make a first guess that only serves as a quick conservative approximation of containment. This conservative guess allows us to focus computational efforts toward checking containment on the most promising previous results and avoid wasting computational effort on less promising ones.

We provide a function GuessComplete(q, v) that performs this heuristic check. GuessComplete(q, v) takes an optimistic approach, representing a *guess* that v can produce a complete rewrite of q. This guess requires the following necessary conditions as described in [46] (SPJ) and [41] (SPJGA) that a view must satisfy to participate in a complete rewrite of q.

(i) v contains all attributes required by q; or contains all necessary attributes to produce those attributes in q that are not in v

(ii) v contains weaker selection predicates than q

(iii) v is less aggregated than q

The function GuessComplete(q, v) performs these checks and returns true if v satisfies the properties i–iii with respect to q. As noted in Section 4.2.1, selection predicates may include simple filters such as comparison operators or more complex filter expressions such as functions (i.e., UDFs). Note that these conditions under-specify the requirements for determining that a valid rewrite exists, as they are necessary but not

sufficient conditions. Thus the guess may result in a false positive, but will never result in a false negative. The purpose of GuessComplete(q, v) is to provide a quick way to distinguish between views that can possibly produce a rewrite from views that cannot. Because searching for rewrites is an expensive process, this helps to avoid examining views that cannot produce valid rewrites.

4.3.2 Costing a UDF

Given that our goal is to find a low cost rewrite for queries containing UDFs, we require a method of costing an MR UDF. We define the cost of a UDF as the sum of the cost of its local functions. Estimating the cost of a local function that performs any of the three operation types is complicated by two factors:

(a) Each operation type is performed by arbitrary user code, and thus can have varying complexity. For instance, although an NLP sentence tagger and a simple word-counter function perform the same operation type (discard or add attributes), they can have significantly different computational costs.

(b) There could be multiple operation types performed in the same local function, making it unrealistic to develop a cost model for every possible local function.

Due to these factors, we desire a conservative way to estimate the cost of a local function of varying complexity that may apply a sequence of operation types without knowing specifically how these operations interact with each other inside the local function.

Developing an accurate cost model is a general problem for any database system. In our framework, the importance of the cost model is only in guiding the exploration of the space of rewrites. For this reason, we appeal to an existing cost model from the literature [72], but slightly modify it to be able to cost UDFs. To this end, we extend the "data only" cost model in [72] in a limited way so that we are able to produce cost estimates for UDFs. Although this results in a rough cost estimate, experimentally we show that our cost model is effective in producing low cost rewrites (Section 4.7). The cost model we develop here is simple but works well in practice; however, an improved cost model may be plugged-in as it becomes available.

Recall that UDFs are composed of local functions, where each local function must be performed by a map task or a reduce task. The cost model in [72] accounts for the "data" costs (read/write/shuffle), and we augment it in a limited way to account for the "computational" cost of local functions. Since a UDF can encompass multiple jobs, we express the cost of each job as the sum of: the cost to read the data and apply a map task (C_m), the cost of sorting and copying (C_s), the cost to transfer data (C_t), the cost to aggregate data and apply a reduce task (C_r), and finally the cost to materialize the output (C_w). Using this as a *generic* cost model, we first describe our approach toward solving (a) above by assuming that each local function only performs one instance of a single operation type. Then we describe our approach for (b), above.

For (a) we model the cost of the three operation types rather than each local function, which provides the baseline cost value for each operation type. Since there may be a high variation in the cost of a UDF's local functions, we apply a scalar multiplier

to the baseline cost of C_m, C_r. To calibrate C_m, C_r we take an empirical approach to estimate the scalar values. The first time the UDF is added to the system, we execute the UDF on a 1% uniform random sample of the input data to determine the scalar values. Due to data skew and one-time calibration, this may result in imprecise cost estimates. However, we do not preclude (a) recalibrating C_m, C_r when the UDF is applied to new data, (b) a better sampling method if more is known about the data, and (c) periodically updating C_m, C_r after executing the UDF on the full dataset.

For (b), since a local function performs an arbitrary sequence of operations of any type, it is difficult to estimate its cost. This would require knowing how the different operations actually interact with one another, which requires a white-box approach. For this reason we desire a conservative way to estimate the cost of a local function, which we do by appealing to the following property of any cost model performing a *set* S of operations.

Definition 1. *Non-subsumable cost property: Let* $\textsc{Cost}(S, D)$ *be defined as the total cost of performing all operations in* S *on a database instance* D*. The cost of performing* S *on* D *is at least as much as performing the cheapest operation in* S *on* D*.*

$$\textsc{Cost}(S, D) \geq \min(\textsc{Cost}(x, D), \forall x \in S)$$

The gray-box model of the UDFs only captures enough information about the local functions to provide a cost corresponding to the least expensive operation performed on the input. We cannot use the most expensive operation in S (i.e.,

$\max(\text{COST}(x, D), \forall x \in S))$, since this requires $\text{COST}(S', D) \leq \text{COST}(S, D)$, where $S' \subseteq S$. The "max" requirement is difficult to meet in practice, which we can show using a simple example. Suppose S contains a filter with high selectivity, and a group-by with higher cost than the filter when considering these operations independently on database D. Let S' contain only group-by. Suppose that applying the filter before group-by results in few or no tuples streamed to group-by. Then applying group-by can have nearly zero cost and it is plausible that $\text{COST}(S', D) > \text{COST}(S, D)$.

The cost model utilizes the non-subsumable cost property in the following way. A local function that performs multiple operation types t is given an initial cost corresponding to the generic cost of applying the cheapest operation type in t on its input data. This initial value can then be scaled-up as described previously in our solution for (a).

4.3.3 Lower-bound on Cost of a Potential Rewrite

Now that we have a quick way to determine if a view v can potentially produce a rewrite for query q, and a method for costing UDFs, we would like to compute a quick lower bound on the cost of any *potential* rewrite – without having to actually find a valid rewrite, which is computationally hard. To do this, we will utilize our UDF model and the non-subsumable cost property when computing the lower-bound. The ability to quickly compute a lower-bound is a key feature of our approach.

Figure 4.5 provides an example showing a view v and a query q annotated as per the model, and a hypothetical local function lf_1, which we can use to compute a

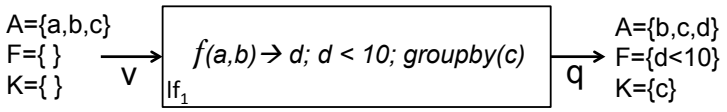

Figure 4.5: Synthesized UDF to perform the fix between a view v and a query q.

lower-bound as described next. View v is given by attributes $\{a, b, c\}$ with no applied filters or grouping keys. Query q is given by $\{b, c, d\}$, has a filter $d < 10$, and has key c, where attribute d is computed using a and b. It is clear that v is guessed to be complete with respect to q because v has all of the required attributes to produce those in q, and v has weaker filters and grouping keys (i.e., is less aggregated) than q. Note that even though v is guessed to be complete, grouping on c may remove a and b, which may render the creation of d not possible; hence, it only a guess. However, since it does pass the GuessComplete(q, v) test, we can then compute what we term as the *fix* for v with respect to q. To determine the fix, we take the set difference between the attributes, filters, and keys (A, F, K) of q and v, which is straightforward and simple to compute. In Figure 4.5, the fix for v with respect to q is given by: a new attribute d; a filter $d < 10$; and re-keying on c, as indicated in lf_1.

To produce a valid rewrite we need to find a sequence of local functions that "perform" the fix; these are the operations that when applied to v will produce q. As this a known hard problem, we *synthesize* a hypothetical UDF comprised of a single local function that applies all operations in the fix (e.g., lf_1 in Figure 4.5). The cost of this synthesized UDF, which serves as an initial stand-in for a potential rewrite should one exist, is obtained using our UDF cost model. This cost corresponds to a lower-bound

for any valid rewrite r. By the non-subsumable cost property, the computational cost of this single local function is the cost of the cheapest operation in the fix. The benefit of the lower-bound is that it lets us cost views by their potential ability to produce a low-cost rewrite, without having to expend the computational effort to actually find one. Later we show how this allows us to consider views that are "more promising" to produce a low-cost rewrite before the "less promising" views are considered.

We define an optimistic cost function $\textsc{OptCost}(q, v)$ that computes this lower-bound on any rewrite r of query q using view v only if $\textsc{GuessComplete}(q, v)$ is true. Otherwise v is given $\textsc{OptCost}$ of ∞, since in this case it cannot produce a complete rewrite, and hence the $\textsc{Cost}$ is also ∞. The properties of $\textsc{OptCost}(q, v)$ are that it is very quick to compute and

$$\textsc{OptCost}(q, v) \leq \textsc{Cost}(r).$$

When searching for the optimal rewrite r^* of P, we use $\textsc{OptCost}$ to enumerate the space of the candidate views based on their cost potential, as we describe in the next section. This is inspired by nearest neighbor finding problems in metric spaces where computing distances between objects can be computationally expensive, thus preferring an alternate distance function (e.g., $\textsc{OptCost}$) that is easy to compute with the desirable property that it is always less than or equal to the actual distance.

4.4 Problem Overview for Rewriting Queries Containing UDFs

Our UDF model enables reuse of views to improve query performance even when queries contain complex functions. However, reusing an existing view when rewriting a query with any arbitrary UDF requires the rewrite process to consider all UDFs in the system. The rewrite problem is known to be hard even when both the queries and the views are expressed in a language that only includes conjunctive queries [1, 46, 64].

In our scenario, users are likely to include many UDFs in their queries. If the rewrite process were to consider every UDF as an operator in the rewrite language, searching for the optimal rewrite would quickly become impractical for any realistic workload and number of views. This is because the search space for finding a rewrite is exponential in both 1) the number of views in V and 2) the number of operators (e.g., Relational and UDFs) considered by the rewrite process, which may include multiple applications of the same operator.

For our rewrite algorithm, the worst case complexity is $O(n \cdot J^{|V|} \cdot k^{|L_R|})$ where n is the number of nodes in the plan P, J is the maximum number of views that can participate in a rewrite, $|V|$ represents the number of views in the system, k is maximum number of times that a particular operator can appear in a rewrite, and $|L_R|$ is the number of operators considered by the rewrite algorithm. The values of J and k should be carefully chosen by a DBA. In the experimental evaluation of our rewrite algorithm presented later in in Section 4.7, we set $J = 4$ and $k = 2$ for practical reasons.

In our system, both the queries and the views can contain any arbitrary UDF, creating a potentially large number of UDFs in the system. Due to the complexity of the rewrite search process, in practice it is a good idea to limit the rewrite process to consider only a small subset of all UDFs in the system. For this reason, in our system the rewriter considers relational operators — select, project, join, group-by, aggregations (SPJGA), and a few of the most frequently used UDFs, which increases the possibility of reusing previous results. Selecting the right subset of UDFs to include in the rewrite process is an interesting open problem that must consider the tradeoff between the added expressiveness of the rewrite process versus the additional exponential cost incurred to search for rewrites.

A naive solution is to search for the optimal rewrite only for target P_n. However, (a) even if a rewrite is found for P_n, there may be a *cheaper* rewrite of P using a rewrite found for a different target P_i, and (b) if one cannot find a rewrite for P_n, one *may* be able to find a rewrite at a different target P_i. The source of this problem is that P_n may contain a UDF that is not included in the set of rewrite operators, and hence search process cannot be restricted only to P_n. For example, a rewrite for P_n can be expressed by composing a rewrite r_i for a target P_i with the remaining nodes in P indicated by $\text{NODE}_{i+1} \cdots \text{NODE}_n$. The composition of this rewrite could be cheaper than the rewrite found at P_n, thus the search process for the optimal rewrite must happen at all n targets in P.

A better solution is to independently search for the best rewrite at each of the n targets of P, and then use a dynamic programming approach to choose a subset among

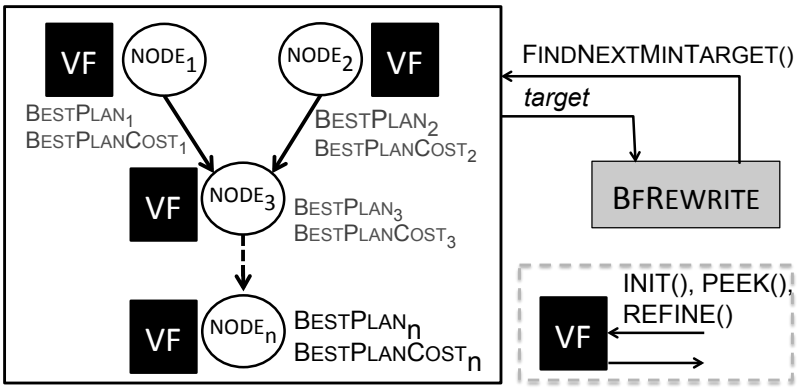

Figure 4.6: High level overview of the BfRewrite algorithm.

these to obtain the optimal rewrite r^*. One drawback of this approach is that there is no way of early terminating the search at a particular target since being independent, the search at one target does not inform the search at another. For instance, the algorithm may have searched for a long time at a target P_i only to find an expensive rewrite, when it could have found a better (lower-cost) rewrite at an upstream target P_{i-1} more quickly had it known to look there first.

The approach we take in this work, called BfRewrite, remedies the two shortcomings of the dynamic programming approach by (1) using the lower bound function OptCost introduced in Section 4.3.3 to guide the search process at each target, and (2) using results from the search process at one target to guide the search at the other targets. More specifically, first, after finding a rewrite r with cost c at a target P_i, there is no need to continue searching for rewrites at P_i if the OptCost of the the remaining unexplored space at P_i is greater than c. Second, r and c can be used to prune the search space at other targets in P by composing a rewrite of P_n using r and the remaining nodes (e.g., $\text{NODE}_{i+1} \cdots \text{NODE}_n$) in P.

Figure 4.6 provides a high-level overview of our BfRewrite algorithm with plan P represented as a DAG. Each node is associated with an instance of the ViewFinder module (VF), which is represented as a black box alongside that is described below. Additionally, each node stores its best rewrite found so far along with its cost. BfRewrite interacts with this DAG using a function that identifies the next target to continue the rewrite search. On the right side of the figure, the interface to the black box ViewFinder is shown, which implements 3 simple primitives. Using this setup, the BfRewrite algorithm performs a search for the globally optimal rewrite of P. There are 3 main components to the BfRewrite algorithm.

1. ViewFinder at each target implements three operations — Init sets up the initial search space of candidate views, ordering the available views by their OptCost; Peek provides the OptCost of the next potential rewrite at the target; and Refine which incrementally grows the space and attempts to find a rewrite of the target. This constitutes the local search at each target.

2. BfRewrite's FindNextMinTarget interface queries the DAG to identify the next target to explore. This constitutes the global search among the targets in P.

3. When a low-cost rewrite is found at a target, it is propagated to the remaining targets in P by updating their best plan and its cost (BestPlan and BestPlanCost). This constitutes the update mechanism that coordinates the search of all targets in P.

These three components represent the global logic of BfRewrite that explores

the rewrite search space, at each step deciding the next target to explore. For each local search, the termination condition is that the remaining views to be examined (PEEK) have a lower bound cost that is greater than the best rewrite found so far. For the global search, the termination condition is that none of the targets has a potential of producing a lower cost rewrite of P_n than the best one found so far. Note that due to the propagation, a node's best plan and cost do not necessarily correspond to the best rewrite found by the VIEWFINDER at that particular node, but could be a composition of rewrites found at other nodes.

In the next two sections, we provide the details of these components and the process outlined above. We first describe BFREWRITE in Section 4.5.1, which is the main driver of the rewrite search process, and is shown in Algorithm 2 and Algorithm 3. The mechanism to propagate the best rewrite is given in Algorithm 4. Then in Section 4.6 we describe the details of the VIEWFINDER component that is utilized as black box in the figure above.

4.5 Best-First Rewrite

The BFREWRITE algorithm produces a rewrite of P that can be composed of rewrites found at multiple targets in P. The computed rewrite r^* has provably the *minimum cost* among all possible rewrites in the same class. Moreover, the algorithm is *work-efficient*: even though COST(r^*) is not known a-priori, it will never examine any candidate view with OPTCOST higher than the optimal cost COST(r^*). To be work

efficient, the algorithm must choose wisely the next candidate view to examine. As we will show below, the OptCost functionality plays an essential role in choosing the next target to refine. Intuitively, the algorithm explores only the part of the search space that is needed to provably find the optimal rewrite. We prove that BfRewrite finds r^* while being work-efficient in Section 4.5.2.

4.5.1 The BfRewrite Algorithm

Algorithm 2 Optimal rewrite of P using ViewFinder

```
1:  function BfRewrite(P, V)
2:    for each P_i ∈ P do                                  ▷ Init Step per target
3:      ViewFinder.Init(P_i, V)
4:      BestPlan_i ←P_i                                    ▷ original plan to produce P_i
5:      BestPlanCost_i ←Cost(P_i)                          ▷ plan cost
6:    end for

7:    repeat
8:      (P_i, d) ← FindNextMinTarget(P_n)
9:      RefineTarget(P_i) if P_i ≠ NULL
10:   until P_i = NULL                                     ▷ i.e., d > BestPlanCost_n
11:   Return BestPlan_n as the best rewrite of P
12: end function
```

Algorithm 2 presents the main BfRewrite function. In lines 2–6, BfRewrite initializes a ViewFinder at each target P_i and sets BestPlan_i and BestPlanCost_i to be the original plan and its cost. In lines 7–10, it repeats the following procedure: Invoke FindNextMinTarget (described in Algorithm 3) to choose the next best target to continue the search, which returns (P_i, d), indicating that target P_i can potentially produce a rewrite with a lower bound cost of d. Next, invoke RefineTarget (described in Algorithm 3) which asks the ViewFinder to search for

the next rewrite at target P_i. This continues until there is no target that can possibly improve BestPlan_n, at which point BestPlan_n (i.e., r^*) is returned.

Algorithm 3 Identify next best target to refine

```
1: function FindNextMinTarget(P_i)
2:   d' ← 0; P_MIN ← NULL; d_MIN ← ∞
3:   for each incoming vertex node_j of node_i do
4:     (P_k, d) ←FindNextMinTarget(P_j)
5:     d' ← d' + d
6:     if d_MIN > d and P_k ≠ NULL then
7:       P_MIN ← P_k
8:       d_MIN ← d
9:     end if
10:  end for
11:  d' ←d' + Cost(node_i)
12:  d_i ← ViewFinder.Peek()
13:  if min(d', d_i) ≥ BestPlanCost_i then
14:    return (NULL, BestPlanCost_i)
15:  else if d' < d_i then
16:    return (P_MIN, d')
17:  else
18:    return (P_i, d_i)
19:  end if
20: end function
```

```
1: function RefineTarget(P_i)
2:   r_i ←ViewFinder.Refine(P_i)
3:   if r_i ≠ NULL and Cost(r_i) < BestPlanCost_i then
4:     BestPlan_i ←r_i
5:     BestPlanCost_i ←Cost(r_i)
6:     for each edge (node_i, node_k) do
7:       PropBestRewrite(node_k)
8:     end for
9:   end if
10: end function
```

FindNextMinTarget in Algorithm 3 identifies the next best target P_i to be refined in P, as well as the minimum cost (OptCost) of a potential rewrite for P_i. There can be three outcomes of a search at a target P_i. Case 1: P_i and all its ancestors

Algorithm 4 Update Mechanism

```
1: function PropBestRewrite(node_i)
2:   r_i ←plan initialized to node_i
3:   for each edge (node_j, node_i) do
4:     Add BestPlan_j to r_i
5:   end for
6:   if Cost(r_i) ¡ BestPlanCost_i then
7:     BestPlanCost_i ←Cost(r_i)
8:     BestPlan_i ←r_i
9:     for each edge (node_i, node_k) do
10:      PropBestRewrite(node_k)
11:    end for
12:  end if
13: end function
```

cannot provide a better rewrite. Case 2: An ancestor target of P_i can provide a better rewrite. Case 3: P_i can provide a better rewrite. By recursively making the above determination at each target P_i in P, the algorithm identifies the best target to refine next.

For a target P_i, the cost d' of the cheapest potential rewrite that can be produced by the ancestors of NODE_i is obtained by summing the ViewFinder.Peek values at NODE_i's ancestors nodes and the cost of NODE_i (lines 3–11). Note that we also record the target P_{MIN} representing the ancestor target with the minimum OptCost candidate view (lines 6–9). Then d_i is assigned to the next candidate view at P_i using ViewFinder.Peek (line 12).

Next the algorithm deals with the three cases outlined above. If both d' and d_i are greater than or equal to BestPlanCost_i (case 1), there is no need to search any further at P_i (line 13). If d' is less than d_i (line 15), then P_{MIN} is the next target to refine (case 2). Else (line 18), P_i is the next target to refine (case 3).

Finally, RefineTarget in Algorithm 3 describes the process of refining a target P_i. Refinement is a two-step process. In the first step it obtains a rewrite r_i of P_i from ViewFinder if one exists (line 2). The cost of the rewrite r_i obtained by RefineTarget is compared against the best rewrite found so far at P_i. If r_i is found to be cheaper, the algorithm suitably updates BestPlan_i and BestPlanCost_i (lines 3–9). In the second step (line 7), the algorithm tries to compose a new rewrite of P_n using r_i, through the recursive function given by PropBestRewrite in Algorithm 4. After this two-step refinement process, BestPlan_n contains the best rewrite of P found so far.

PropBestRewrite in Algorithm 4 describes the recursive update mechanism that pushes the new BestPlan_i downward along the outgoing nodes and towards node_n. At each step it composes a rewrite r_i using the immediate ancestor nodes of node_i (lines 2–5). It compares r_i with BestPlan_i and updates BestPlan_i if r_i is found to be cheaper (lines 6–12).

4.5.2 Proof of Correctness and Work-Efficiency

The following theorem provides the proof of correctness and the work-efficiency property of our BfRewrite algorithm.

Theorem 1. BfRewrite *finds the optimal rewrite* r^* *of* P *and is work-efficient.*

Proof. To ensure correctness, BfRewrite must not terminate before finding r^*. Correctness requires that we show the algorithm examines every candidate view with OptCost less than or equal to $\text{Cost}(r^*)$. To ensure work-efficiency, BfRewrite

should not examine any extra views that cannot be included in r^*. Work-efficiency requires that we show the algorithm must not examine any candidate view with OptCost greater than Cost(r^*). We first prove these two properties of BfRewrite for a query P containing a single target, then extend these results to the case when P contains n targets.

For P with a single target (i.e., $n = 1$), proof by contradiction proceeds as follows. Consider two different rewrites r and r^* such that r^* is the optimal rewrite and Cost(r^*) < Cost(r). Assume a candidate view v^* produces the optimal rewrite r^*. Assume another candidate view v produces the rewrite r. Suppose that BfRewrite examines v, sets BestPlanCost$_n$ to Cost(r), and then terminates before examining v^*. Hence BfRewrite incorrectly reports r as the optimal rewrite even though Cost(r^*) < Cost(r). Because BfRewrite examines candidate views by increasing OptCost, then since v was examined before v^*, it must have been the case that OptCost(v) < OptCost(v^*). By design, BfRewrite will continue until all candidate views whose OptCost is less than or equal to BestPlanCost$_n$ have been found. Given the lower bound property of OptCost with respect to Cost, we have that:

$$\text{OptCost}(v) < \text{OptCost}(v^*) \leq \text{Cost}(r^*) < \text{Cost}(r) = \text{BestPlanCost}_n.$$

From the above inequality, it is clear that the algorithm must have examined v^* (and consequently found r^*) before terminating. This results in a contradiction since we assumed earlier that BfRewrite terminated before examining v^*.

We can similarly prove work-efficiency by contradiction as follows. Assume that $\textsc{OptCost}(v) > \textsc{Cost}(r^*)$ and as above the algorithm examines v before v^*. This results in the following inequality.

$$\textsc{Cost}(r^*) < \textsc{OptCost}(v) \leq \textsc{Cost}(r) = \textsc{BestPlanCost}_n < \textsc{OptCost}(v^*).$$

The results in a contradiction since $\textsc{Cost}(r^*)$ cannot be less than $\textsc{OptCost}(v^*)$, based on the lower-bound property of OptCost. Hence, work-efficiency is shown since BfRewrite never examines any candidate view with OptCost greater than $\textsc{Cost}(r^*)$.

Now we extend this result to P with multiple targets, $n > 1$. It is sufficient to show that the BfRewrite algorithm works by reducing n individual search problems into a single global search problem that finds the optimal rewrite for the target P_n. Recall that BfRewrite instantiates a priority queue PQ at each of the n targets in the plan, where the candidate views at each PQ are ordered by increasing OptCost. Next we show that the search process degenerates these n PQs into a single virtual global PQ whose elements are potential rewrites of P_n, ordered by their increasing OptCost. Recall that every invocation of FindNextMinTarget identifies the next best target to refine out of all targets in P. It composes the lowest cost potential rewrite for P_n by recursively visiting each target P_i and selecting the cheaper among either the candidate view at the front of the PQ for P_i or the current best known rewrite for P_i. Thus this recursive procedure identifies the current lowest cost potential rewrite of P_n, in effect gradually exploring the space of potential rewrites of P_n by their increasing OptCost.

This creates a virtual global PQ whose elements are potential rewrites of P_n and is ordered by their OptCost.

We have now reduced n priority queues of candidate views ordered by OptCost to a single global priority queue of potential rewrites of P_n ordered by OptCost. This completes the proof since we already showed above that BfRewrite with a single PQ ordered by OptCost will find the optimal rewrite and is work-efficient. □

4.6 ViewFinder

The key feature of ViewFinder is its OptCost functionality that enables it to incrementally explore the the space of rewrites using the views in V. As noted earlier in Section 4.3.1, rewriting queries using views is known to be a hard problem. Traditionally, methods for rewriting queries using views for SPJG queries use a two stage approach [6, 46]. The pruning stage determines which views are *relevant* to the query, and among the relevant views, those that contain all the required join predicates are termed as *complete* otherwise they are called *partial* solutions. This is typically followed by a merge stage that joins the partial solutions using all possible equijoin methods to form additional relevant views. The algorithm repeats until only those views that are useful for answering the query remain.

We take a similar approach in that we identify partial and complete solutions, then follow with a merge phase. The ViewFinder considers candidate views C when

searching for rewrite of a target. C includes views in V as well as views formed by "merging" views in V using a MERGE function, which is an implementation of a standard view-merging procedure (e.g., [6, 46]). Traditional approaches begin merging partial solutions to create complete solutions, until no partial solutions remain. This "explodes" the space of candidate views exponentially up-front. In contrast, our approach gradually explodes the space, resulting in far fewer candidates views from being considered.

Additionally, with no early termination condition, existing approaches would need to explore the space exhaustively at all targets. The VIEWFINDER incrementally grows and explores only as much of the space as needed, frequently stopping and resuming the search as requested by BFREWRITE.

4.6.1 The ViewFinder Algorithm

The VIEWFINDER is presented in Algorithm 5. An instance of VIEWFINDER instantiated at each target, which is *stateful*; enabling it to start, stop, and resume the incremental searches at each target. The VIEWFINDER maintains state using a priority queue (PQ) of candidate views, ordered by OPTCOST. VIEWFINDER implements the INIT, PEEK, and REFINE functions.

The INIT function instantiates a VIEWFINDER with a query q representing a target P_i, and all views in V are added to PQ, which orders them by increasing OPTCOST. The PEEK function returns the head item in PQ. The REFINE function is invoked when BFREWRITE asks the VIEWFINDER to examine the next candidate view.

REFINE pops the head item v out of PQ and generates a set of new candidate

views M by merging v with those views previously popped from PQ which were stored in *Seen*. Note that *Seen* only contains candidate views that have an OptCost less than or equal to that of v. Critically, this results in an "on-demand" incremental growth of the candidate space as required by BfRewrite, rather than performing a pre-explosion of the entire search space. All newly created views in M are inserted into PQ and v is then added to *Seen*. A property of the new candidate views in M, which is required for the correctness of the algorithm, is that they have an OptCost greater than v, hence none of these views could have been examined before v. This property is provided below by Theorem 2.

Theorem 2. *The* OptCost *of every candidate view in M that is not in Seen is greater than or equal to the* OptCost *of v.*

Proof. The proof sketch is as follows. The theorem is trivially true for $v \in V$ as all candidate views in M cannot be in *Seen* and have OptCost greater than v. If $v \notin V$, it is sufficient to point out that all constituent views of v are already in *Seen* since they must have had OptCost lesser or equal to v. Hence all candidate views in M with OptCost smaller than v are already in *Seen*, and those with OptCost greater than v will be added to PQ if they are not already in PQ. □

The Refine function next attempts to find a rewrite using view v by invoking RewriteEnum, described next. Given the computational complexity of finding valid rewrites, ViewFinder limits the invocation of the RewriteEnum algorithm using two strategies. First, the expensive RewriteEnum operation is only applied to the

view at the head of PQ when requested by BfRewrite. Second, it avoids applying RewriteEnum on every candidate view unless it passes the GuessComplete test as described in Section 4.3.3.

Algorithm 5 ViewFinder

```
1: function Init(query, V)
2:   Priority Queue PQ ←∅; Seen ←∅; Query q
3:   q ←query
4:   for each v ∈ V do
5:     PQ.add(v, OptCost(q, v))
6:   end for
7: end function
```

```
1: function Peek
2:   if PQ is not empty return PQ.peek().OptCost else ∞
3: end function
```

```
1: function Refine
2:   if not PQ.empty() then
3:     v ←PQ.pop()
4:     M ←Merge(v, Seen)                    ▷ Discard from M those in Seen ∩ M
5:     for each v′ ∈ M do
6:       PQ.add(v′, OptCost(q, v′))
7:     end for
8:     Seen.add(v)
9:     if GuessComplete(q, v) then
10:      return RewriteEnum(q, v)
11:    end if
12:  end if
13: return NULL
14: end function
```

4.6.2 Rewrite Enumeration

The RewriteEnum function searches for a valid rewrite of query q using view v that has passed the GuessComplete test. Since GuessComplete can result in false positives, there is no guarantee that v will produce a valid rewrite for q. However, if

a rewrite exists, RewriteEnum returns the rewrite and its cost as computed by the Cost function.

In searching for a rewrite, recall from Section 4.4 that the rewrite process considers relational operators SPJGA and a subset of the UDFs in the system. These are the only rewrite operators considered by RewriteEnum. The rewrite process searches for equivalent rewrites of q by applying *compensations* [101] to v and then testing for equivalence against q. In our implementation of RewriteEnum this is done by generating all permutations of the rewrite operators and testing for equivalence, amounting to a brute force enumeration of all possible rewrites that can be produced with compensations. This makes the case for the system to keep the set of rewrite operators small since this search process is exponential in the size of this set. However, when the rewrite operators are restricted to a fixed known set, it may suffice to examine a polynomial number of rewrite attempts as in [44] for the specific case of simple aggregations involving group-bys. Such approaches are not applicable to our case as the system has the flexibility to add any UDF to the set of rewrite operators.

4.7 Experimental Evaluation

In this section, we present an experimental study showing the effectiveness of BfRewrite in finding low-cost rewrites of complex queries. First, we evaluate our methods in two scenarios. The *query evolution* scenario (Section 4.7.3.1) represents a user iteratively refining a query within a single session. This scenario evaluates the

benefit that each new query version can receive from the opportunistic views created by previous versions of the query. The *user evolution* scenario (Section 4.7.3.2) represents a new user entering the system presenting a new query. This scenario evaluates the benefit a new query can receive from the opportunistic views previously created by queries of other "similar" users. Query evolution and user evolution are described in more detail later in Chapter 6 which motivates these scenarios that are becoming commonplace in big data analytics. Next, we evaluate the scalability (Section 4.7.3.3) of our rewrite algorithm in comparison to a competing approach. We then compare our method to cache-based methods (Section 4.7.3.4) that can only reuse identical previous results. We then show the performance of our method (Section 4.7.3.5) under a storage reclamation policy that drops opportunistic views.

We follow these experiments with several experiments that provide a more qualitative evaluation of our approach. We evaluate the "goodness" of our cost model (Section 4.7.4) by comparing actual execution time to the estimates computed by our cost model. To further evaluate the quality of rewrites produced by BfRewrite, we perform a sanity check experiment (Section 4.7.5) with relational queries (i.e., no UDFs) utilizing data from the standard TPC-H [95] benchmark which compares the quality of rewrites produced by our algorithm with those produced by a state-of-the-art DBMS. Lastly, we provide a Microbenchmark experiment (Section 4.7.6) as a sensitivity analysis showing how BfRewrite performs as we vary the amount of "overlap" between the queries in the experimental workload.

4.7.1 Query Workload

We first provide some insights into the characteristics of exploratory processing on big data and then briefly describe and motivate the workload we use in this work; a more detailed description of the workload is provided later in Chapter 6. Recent work [28, 29, 81] examines MR queries "in the wild" on production and research clusters that were utilized by data scientists or other advanced users for up to a year, while additional work [47, 56, 89, 98] provides further insights into big data analytical queries. We discuss some of the relevant findings here. A key finding of [29] is that there is a need for better benchmarks to capture the use cases for MR queries that perform interactive analysis on big data. Below we summarize the main findings from our literature review.

1. Users spend time revising and improving exploratory queries [56,81], and thus queries near the end of an exploratory session tend to represent higher-quality and more complex versions of earlier queries [56].

2. Many studies note that complex analysis on big data frequently include UDFs [47, 89, 98].

3. Queries frequently incorporate multiple datasets [81] with a majority of queries (65% in [81]) accessing three or more.

Both [81] and [28] find that users frequently re-access their data and there can be significant benefits from caching. A majority of jobs involve data re-accesses with many occurring within 1 hour (50% [28] and up to 90% [81]). These frequent re-access patterns make a strong case for a method such as BfRewrite.

The experimental workload from [61] contains 32 queries on three datasets that simulate 8 analysts A_1–A_8 who write complex exploratory analytical queries for business marketing scenarios. The workload captures many of the findings discussed above, and is described in more detail later in Chapter 6 but we provide a high-level overview here. Each query uses at least one of 10 unique UDFs and the workload uses three real-world datasets: A Twitter log (TWTR) of user tweets, a Foursquare log (4SQ) of user check-ins, and a Landmarks log (LAND) containing locations of interest. Many queries in the workload begin by accessing only one or two datasets, but subsequent revisions use all three datasets. Each of the 8 analysts poses 4 versions of a query, representing the initial query followed by three subsequent revisions made during data exploration and hypothesis testing. Hence, there is some overlap expected between subsequent versions of a query. The queries are long-running with many operations, and executing the queries with Hive created 17 opportunistic materialized views per query on average.

Since each query in the workload has multiple versions, we use $A_i v_j$ to denote Analyst i executing version j of her query. Since there are 4 versions of each query, $A_i v_{j+1}$ represents a revision of $A_i v_j$. Below is a high-level description of query $A_1 v_1$ and $A_1 v_2$, from [61].

Example 1. *Analyst1 (A_1) wants to identify a number of "wine lovers" to send them a coupon for a new wine being introduced in a local region.*

Query $A_1 v_1$: (a) From TWTR, apply UDF-CLASSIFY-WINE-SCORE on each user's tweets and group-by user to produce a wine-sentiment-score for each user and then threshold on wine-sentiment-score. (b) From TWTR, compute all pairs $\langle u_1, u_2 \rangle$ of

users that communicate with each other, assigning each pair a friendship-strength-score based on the number of times they communicate and then threshold on the friendship-strength-score. (c) From TWTR, apply UDAF-CLASSIFY-AFFLUENT on users and their tweets. Join results from (a), (b), (c) on user_id.

Query A_1v_2: Revise the previous version by reducing the wine-sentiment-score threshold, adding new data sources (4SQ and LAND) to find the check-in counts for users that check-in to places of type wine-bar, then threshold on count, joining this result with the users found in the previous version. Queries A_1v_3 and A_1v_4 are similarly revised by changing the threshold parameters and requiring that a user's friends also have a high check-in count to wine-bars.

The performance of any method that reuses results from previous queries will obviously depend on the degree of "similarity" between queries. However, choosing a meaningful metric to compute the similarity between queries in the workload from [61] was not clear. While methods such as [56] characterize query similarity in terms of query text (FROM clause, WHERE clause, etc.), we found this did not directly correspond with result reusability. We observed this effect in a microbenchmark we performed based on revising queries, and report those results later in Section 4.7.6 at the end of the experimental evaluation.

4.7.2 Experimental Methodology

Our experimental system consists of 20 machines running Hive version 0.7.1 and Hadoop version 0.20.2. Each node has the same hardware: 2 Xeon 2.4GHz CPUs

(8 cores), 16 GB of RAM, and exclusive access to its own disk (2TB SATA 2012 model). We use HiveQL as the declarative query language, and Oozie as a job coordinator. The MR UDFs are implemented in Java, Perl, and Python and executed using the HiveCLI. UDFs implemented in our system include a log parser/extractor, text sentiment classifier, sentence tokenizer, lat/lon extractor, word count, restaurant menu similarity, and geographical tiling, among others. All UDFs are annotated using the model as per the example annotations given in Section 4.2.2. For each UDF in the workload we calibrate its cost model using the procedure described in Section 4.3.2. Although we calibrate our cost model only the first time the UDF is added, our experimental results show that it is able to discriminate between good plans and really bad plans for the purpose of query rewriting. The quality of the cost model after calibration is evaluated later in Section 4.7.4.

Our experiments use over 1TB of data that includes three logs, consisting of 800GB of Twitter (TWTR) tweets, 250GB of Foursquare (4SQ) check-ins, and 7GB of Landmarks data (LAND) containing 5 million landmarks. The identity of a user (`user_id`) is common across the TWTR and 4SQ logs, while the identity of a landmark (`location_id`) is common across 4SQ and LAND. Log rows that contain dirty, malformed, or incomplete data are discarded (i.e., ignored) by the queries.

We report the following metrics for all experiments. Experiments on query execution time report both the original execution time of the query in Hive, labelled as ORIG, and the execution time of the rewritten query, labelled as REWR. The reported time for REWR includes the time to run the BFREWRITE algorithm, the time to execute

the rewritten query, and any time spent on statistics collection. Experiments on the runtime of rewrite algorithms report the total time used by the algorithm to find a rewrite of the original query using the views in the system. For these experiments, BFR denotes our BfRewrite rewriting algorithm, and DP represents a competing approach based on dynamic programming. DP does not use OptCost, and searches exhaustively for rewrites at every target. DP then rewrites a query by applying a dynamic programming solution to choose the best subset of rewrites found at each target. Critically, we note that both algorithms produce identical rewrites (i.e., r^*). The primary comparison metric for BFR and DP is algorithm runtime. In addition, we report results for two secondary metrics: the number of candidate views examined during the search for rewrites, and the number of valid rewrites attempted and produced during the search process. These correspond to the candidate space explored and rewrites attempted before identifying r^*.

4.7.3 Experimental Results

The following sections evaluate the quality of the rewrites from BfRewrite (in terms of query execution time improvement) and the performance of the BfRewrite algorithm.

4.7.3.1 Query Evolution

In this experiment, for each analyst A_i, query A_iv_1 is executed followed by query A_iv_2, A_iv_3, and A_iv_4, applying BfRewrite each time to rewrite the new query

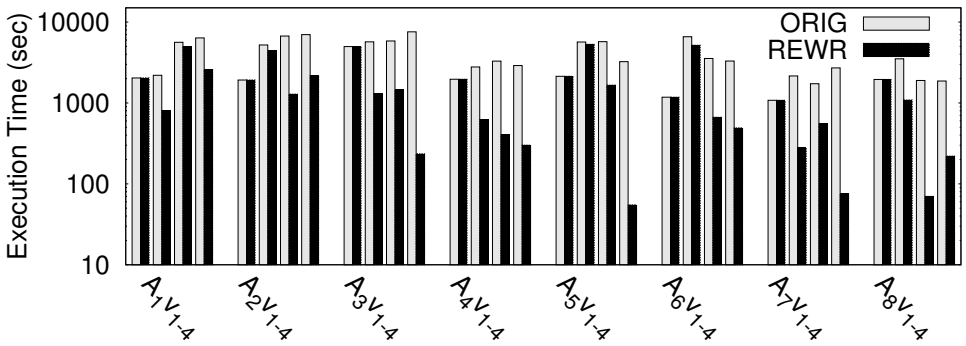

Figure 4.7: Query Evolution comparisons for query execution time (log-scale)

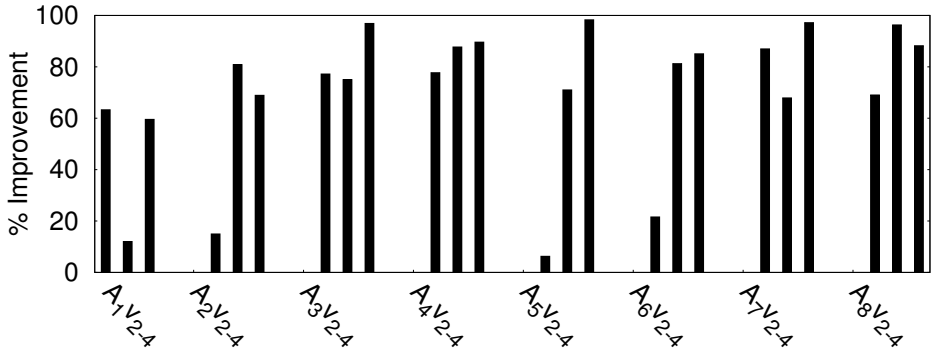

Figure 4.8: Query Evolution comparisons for execution time improvement.

using the opportunistic views generated by the previous query versions. Before each Analyst A_i begins, all views are dropped from the system. In this scenario, an analyst may benefit by reusing results from previous versions of their own query.

Figure 4.7 shows the total execution time for each original query (ORIG) in the workload followed by the rewritten query (REWR) produced by BfRewrite. As there are no other views in the system when each analyst's first query arrives (i.e., A_iv1), the original query and the rewritten query always have the same execution time. For each

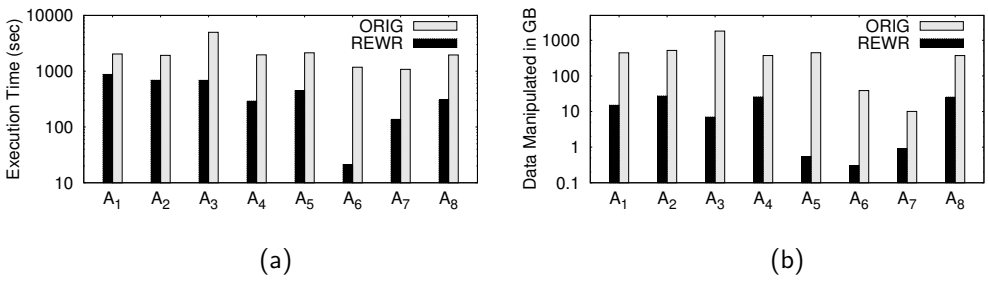

Figure 4.9: User Evolution comparisons (log-scale) for (a) execution time , and (b) data moved.

subsequent version of the query (i.e., $A_i v_2 - A_i v_4$), these results show that the rewritten query always resulted in a lower execution time. This is because BFR was able to take advantage of the opportunistic views generated by all of the prior versions of each query.

Figure 4.8 reports the corresponding percent improvement in execution time for each query in Figure 4.7. Note that since the execution time for ORIG and REWR are always the same for $A_i v_1$, the figure only reports the improvements for query versions $A_i v_2$, $A_i v_3$, and $A_i v_4$. Figure 4.8 shows REWR provides an overall improvement of 10% to 90% for each query; with an average improvement of 61% and up to an order of magnitude. As a concrete data point, $A_5 v_4$ requires 54 minutes to execute ORIG, but only 55 seconds to execute the rewritten query (REWR). REWR has much lower execution time because it is able to take advantage of the overlapping nature of the queries, e.g., version 2 has some overlap with version 1. REWR is able to exploit and reuse previous results effectively, providing significant savings in query execution time. The savings is due in part to reduced data access and movement costs (read/write/shuffle), which we evaluate explicitly in the following section.

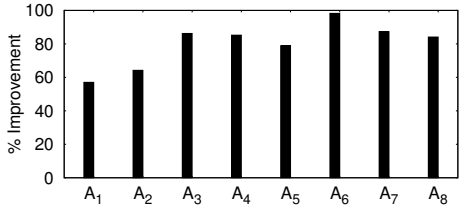

Figure 4.10: Execution time improvement for REWR corresponding to Figure 4.9(a).

4.7.3.2 User Evolution

In this experiment, each analyst (except one, a holdout analyst) executes the first version of their query. Then, we execute the first version of the holdout analyst's query (e.g., $A_i v_1$) after applying BFREWRITE to rewrite the holdout query using the opportunistic views generated by the previous queries. We then drop (i.e., discard) all views from the system and repeat using a different holdout analyst each time. In this scenario, an analyst may benefit by reusing results from previous versions of other analysts' queries.

Figure 4.9(a) shows the execution time for REWR and ORIG for each different holdout analyst along the x-axis, while Figure 4.9(b) shows the corresponding data manipulated (read/write/shuffle) in GB. These data statistics are automatically collected and reported by Hadoop and include the amount of data read from HDFS, moved across the network, and written to HDFS. These results demonstrate that the execution time of REWR is always lower than ORIG, and the data manipulated shows similar trends.

The corresponding percentage improvement in execution time is given in Figure 4.10 which shows REWR results in an overall improvement of about 50%–90%. Of course, these results are workload dependent but they show that even when several an-

Analysts added	1	2	3	4	5	6	7
Improvement	0%	73%	73%	75%	89%	89%	89%

Table 4.1: Improvement in execution time of A_5v_3 as more analysts become present in the system.

alysts query the same data sets while testing different hypothesis, our approach is able to find some overlap and hence benefit by exploiting previous results.

As an additional experiment for user evolution, we first execute a single analyst's query (A_5v_3) with no opportunistic views in the system, to create a baseline execution time. Then we "add" another analyst by executing all four versions of that analyst's query, which creates new opportunistic views. Then we re-execute A_5v_3 and report the execution time improvement over the baseline, and repeat this process for the other remaining analysts. We chose A_5v_3 as it is a complex query that uses all three logs. Table 4.1 reports the execution time improvement after each analyst is added, showing that the improvement increases when more analysts are present in the system. That is, each additional analyst's queries generates more opportunistic views that can then be exploited by our approach.

4.7.3.3 Algorithm Comparisons

To evaluate the performance of our BfRewrite algorithm (BFR), we first compare BFR to DP in terms of the number of candidate views considered, the number of times the algorithm attempts a rewrite, and the algorithm runtime in seconds. We use the user evolution scenario from the previous experiment, where there were approximately 100 views in the system when each holdout analyst's query was executed.

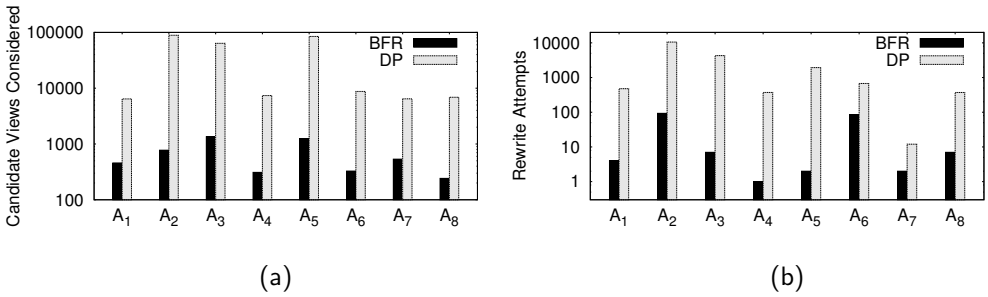

Figure 4.11: Algorithm comparisons (log-scale) for (a) candidate views considered, (b) rewrite attempts.

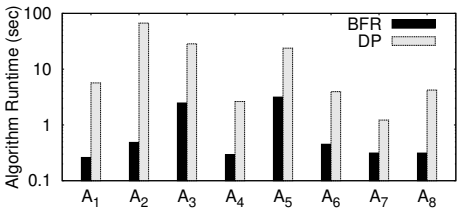

Figure 4.12: Comparison of algorithm runtime (log-scale).

Figure 4.11(a) shows that even though both algorithms find identical rewrites, BFR searches much less of the space than DP since it considers far fewer candidate views when searching for rewrites. This result highlights the effectiveness of our lower-bound OptCost property and our BfRewrite approach to incrementally grow and search the space of candidate views in an on-demand fashion as required.

Similarly, Figure 4.11(b) shows that BFR attempts far fewer rewrites compared to DP. This improvement can be attributed to GuessComplete identifying the promising candidate views, and OptCost enabling BFR to incrementally examine the candidate views, thus applying RewriteEnum far fewer times. Together, these contribute to BFR doing far less work than DP, which is reflected in the algorithm runtime shown in Figure 4.12. In summary, by growing the space of candidate views incremen-

tally as needed on-demand (Figure 4.11(a)) rather than an up-front explosion of the space of candidate views and by controlling the number of times a rewrite is attempted (Figure 4.11(b)), BfRewrite results in runtime significant savings (Figure 4.12) since both of these operations increase the search space exponentially.

We next test the scalability of BFR and DP by scaling up the number of views in the system from 1–1000 and report the algorithm runtime for both algorithms as they search for rewrites. We use one query here, query A_3v_1, as it incorporated multiple datasets in its first version, increasing the difficulty of the rewrite search due to the need for views from several different datasets. During the course of design and development of our system, we created and retained about 9,600 views; from these we discarded duplicate views as well as those views that are an exact match to the query (simply to prevent the algorithms from terminating trivially). In Figure 4.13, the x-axis reports the the number of views (views are randomly drawn from among these candidates), while the y-axis reports the algorithm run time (log-scale). DP becomes prohibitively expensive even when 250 views are present in the system, due to the exponential search space. BFR on the other hand scales much better than DP and has a runtime under 1000 seconds even when the system has 1000 views relevant to the given query. This is due to the ability of BFR to control the exponential space explosion and incrementally search the rewrite space in a principled manner.

While this runtime is not trivial, we note that these are complex queries involving UDFs that run for thousands of seconds. The amount of time spent to rewrite a query plus the execution time of the rewritten query is far less than the execution time

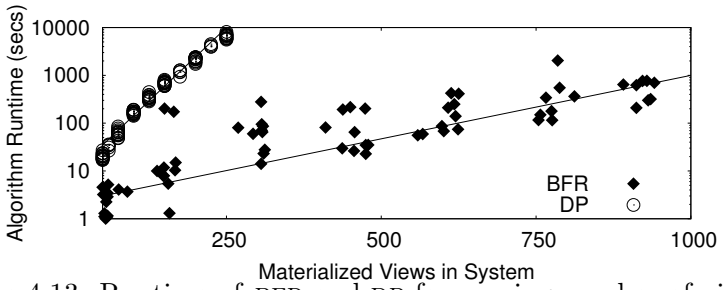

Figure 4.13: Runtime of BFR and DP for varying number of views.

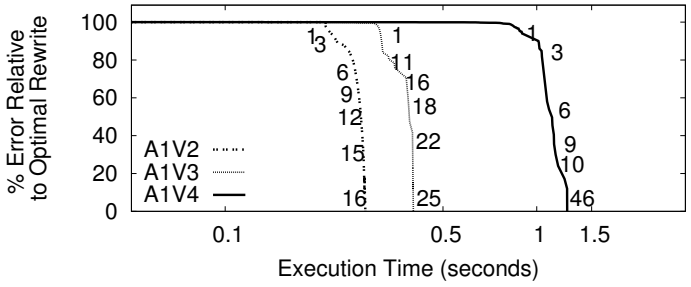

Figure 4.14: Execution time analysis of the quality of BFR's rewrite solutions found during its search for the optimal rewrite.

of the original query. For instance, Figure 4.9(a) reports a query execution time of 451 seconds for A_5 optimized versus 2134 seconds for unoptimized. Even if the rewrite time for A_5 were 1000 seconds (it is actually 3.1 seconds here as seen in Figure 4.11(c)), the total execution time would still be 32% faster than the original query.

Finally, we show the effectiveness of OptCost in pruning the search space for BFR. Figure 4.14 shows the runtime behavior of BfRewrite as it explores the space of rewrites in its search to identify the optimal rewrite. In this experiment, query A_1v_1 is first executed, producing a number of views. For each subsequent query (A_1v_2, A_1v_3, A_1v_4), BfRewrite searches for a rewrite given the views produced by the previous queries. The x-axis reports the BfRewrite's elapsed run time, and the y-axis reports

the percent error relative to the optimal rewrite, in terms of cost. For each query the error begins at 100% (i.e., no rewrite has been found yet) and as BfRewrite finds rewrites, the error is reduced until it reaches zero percent. At this point BfRewrite has identified the optimal rewrite and can terminate the search (note that this is the same rewrite identified by the exhaustive DP algorithm). During the initial "flat" period for each query, BfRewrite is growing the space of candidate views by examining views with the lowest OptCost. Since those views fail to produce a rewrite, BfRewrite begins merging them with views that have the next lowest OptCost. This phase represents BfRewrite incrementally growing the space of candidate views, which is in contrast to an exhaustive approach (DP) that first grows the entire space of all possible candidate views before beginning to search for rewrites. The execution time for BfRewrite increases slightly for the subsequent queries A_1v_3 and A_1v_4 since the execution of each subsequent query adds more opportunistic views to the system and hence the search space increases exponentially.

In Figure 4.14, BfRewrite finds the first three valid rewrites for query A_1v_4 at about 0.9 seconds (indicated by the numbers 1 and 3) which reduces the error to 96% and 90% respectively. At shortly after 1 second, BfRewrite finds valid rewrite number 46 which is the optimal rewrite and hence reduces the relative error to 0% and terminates. Two notable take-aways from Figure 4.14 are: (a) once BfRewrite finds the first rewrite, it quickly converges to the optimal rewrite, and (b) when BfRewrite finds the optimal rewrite and terminates for A_1v_4, it only had to find 46 rewrites before terminating, while the DP algorithm (not shown in figure) found 4656 rewrites. Sim-

ilarly, BfRewrite only had find 16 and 25 rewrites for A_1v_2 and A_1v_3 respectively, whereas DP found 66 and 323. This result illustrates a case when BfRewrite can terminate early without examining all possible rewrites. These observations suggest that the OptCost is effective at pruning the search space for BfRewrite.

4.7.3.4 Comparison with Caching-based methods

Next we provide a brief comparison of our approach with caching-based methods (such as [37]) that perform only syntactic matching when reusing previous results. With this class of solutions, results can only be reused to answer a new query when their respective execution plans are identical, i.e., the new query's plan and the plan that produced the previous results must be syntactically identical. This means that if the respective plans differ in any way (e.g., different join orders or predicate push-downs), then reuse is not possible. For instance, with syntactic matching, a query that applies two filters in sequence a, b will not match a view (i.e., a previous result) that has applied the same two filters in a different sequence b, a. In contrast, our BFR approach performs semantic matching and query rewriting. In this case, not only will BFR match a, b with b, a, but it would also match the query to a view that only has filter b, by applying an additional filter a during the rewrite process.

To represent the class of syntactic caching methods, we present a conservative variant of our approach that performs a rewrite only if a view and a query have identical A, F, K properties as well as have identical plans. We term this variant BFR-SYNTACTIC.

Figure 4.15 highlights the limitations of caching-based methods by repeating

the query evolution experiment for Analyst 1 (A_1v_1–A_1v_4). We first execute query A_1v_1 to produce opportunistic views, and then we apply both BFR and BFR-SYNTACTIC to queries A_1v_2, A_1v_3 and A_1v_4 and report the results in terms of query execution time improvement of the solutions produced by BFR and BFR-SYNTACTIC. Figure 4.15 shows that both BFR and BFR-SYNTACTIC result in the same execution time improvement for A_1v_2. This is because both methods were able to reuse some of the (syntactically identical) views from the previous query. However, BFR-SYNTACTIC performs worse than BFR for query A_1v_3 and A_1v_4. This is because BFR-SYNTACTIC was unable to find many views that were exact syntactic matches, whereas BFR was able to exploit additional views due to BFR's ability to reuse and re-purpose previous results through semantic query rewriting. Although this result is workload dependent, this example highlights the fact that while reusing identical results is clearly beneficial, our approach completely subsumes those that only reuse syntactically identical results: even when there are no identical views our method may still find a low-cost rewrite. To further illustrate this, we next perform an additional experiment after removing *all* identical views from the system before applying our BfRewrite algorithm.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
BFR	57%	64%	83%	85%	51%	96%	88%	84%
BFR-SYNTACTIC	0%	0%	0%	0%	0%	0%	0%	0%

Table 4.2: Execution time improvement for each analyst's first query (e.g. A_iv_1) after (1) executing all other analysts' queries and (2) discarding from the system all views identical to those required by the first query (i.e., the holdout query).

We repeat the user evolution experiment after discarding from the system all

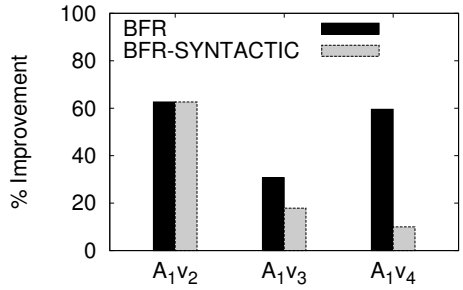

Figure 4.15: Execution time improvement over ORIG.

views that are identical to a target in each of the holdout queries ($A_{1-8}v_1$). Without these specific views, syntactic caching-based methods will not be able to find *any* rewrites, offering no improvement. This scenario represents a worst-case environment for syntactic caching methods. Table 4.2 reports the percentage improvement of the BFR and BFR-SYNTACTIC algorithms for each analyst A_1–A_8 after discarding all identical views.

Table 4.2 shows 0% improvement for BFR-SYNTACTIC, while BFR is able to reduce query execution time dramatically even when there are no views in the system that are an exact match to the new queries. This result highlights the benefits of our semantic query rewriting approach that can effectively reuse and repurpose previous results. While reusing cached results is clearly effective, the performance of syntactic methods is highly dependent upon the presence of identical answers previously generated in the system, as such methods lack a robust capability to reuse previous results and cannot perform semantic query rewriting.

The performance improvements reported in Table 4.2 are comparable to the

View Storage Space	2.0×	1.8×	1.5×	1.0×	0.6×	0.2×
Improvement	89%	89%	86%	82%	58%	30%

Table 4.3: Execution Time Improvement after reclaiming 10% to 90% of the view storage space.

result in Figure 4.10 which represents the same experiment without discarding the identical views. Notably there is a drop in performance improvement for A_5 compared to the results reported in Figure 4.10 for A_5. This is because previously in Figure 4.10, A_5 had benefited from an identical view corresponding to a restaurant-similarity computation that must now be recomputed.

Interestingly, the identical views discarded constituted only 7% of the view storage space in this experiment and hence the vast majority of views in the system were not identical. This indicates there are *many other* useful views that are not necessarily identical but can still provide significant opportunities for reuse. This result highlights the fact that our semantic-based method can exploit these other views, whereas syntactic methods cannot. Given that analysts pose different but *related* queries in an evolutionary analytical scenario on big data, any method that relies solely on identical matching of previous results may have limited benefit.

4.7.3.5 Storage Reclamation

Since storage is not infinite, a reclamation policy is necessary. Although choosing a beneficial set of views to retain within a finite storage budget is an interesting problem for future work, here we apply two simple policies for storage reclamation. Our aim in this experiment is to show the robustness of our approach even when the

reclamation policy makes unsophisticated choices such as dropping views randomly or dropping all of the views identical to a given query.

First, we repeat the experiment in Section 4.7.3.2 using the A_5v_3 query, but reduce the view storage budget each time. Retaining all views resulted in a storage space of approximately 2.0× the base data size (≈2TB). The relatively small total size of the views with respect to the log base data is due to several reasons. First, the logs are very wide, as they record a large number of attributes. However, a typical query only consumes a small fraction of these log attributes, which consistent with observations in big data systems. Second, it is not uncommon for the log attributes to have missing values, since the data may be dirty or incomplete. For instance, in the Twitter log, a tweet may have missing location values. Thus a query may discard those tweets without location values. Third, in this experiment, views are not duplicated since the rewriter makes use of existing views whenever possible. For these reasons, the total size of the views is relatively small compared to the size of the base data.

Table 4.3 reports the execution time improvement for the rewritten query compared to the original query for each storage budget. We repeat each experiment twice, each time randomly dropping views from the full set of views, and report the average improvement. The results show that our method is able to find good rewrites using the remaining views available, until the view storage budget is very small. Second, by the results shown earlier in Table 4.2, our method is able to find good rewrites even when there are no identical views in the system; which could be the case if a poor reclamation policy were used. Designing a good storage reclamation policy is equivalent

to the view selection problem [53, 68] with a storage constraint, and certainly better methods could be applied. We do not address this problem here, but in the following chapter (Chapter 5) we consider different policies for reclaiming storage consumed by opportunistic views.

4.7.4 Calibration of Cost Model

In this section, we provide an experimental validation of our cost model presented in Section 4.3.2. The cost model that we use is static, requiring a one-time calibration for each UDF added to the system. However, we do not preclude a dynamic cost model that (a) recalibrates C_m, C_r when the UDF is applied to new data, or (b) periodically updates C_m, C_r after executing the UDF on the full dataset in the same way an RDBMS optimizer periodically updates its statistics. Both of these approaches simply provide a way to fine-tune the values for C_m, C_r, and we note that either (a) or (b) could be applied without affecting our techniques. Here we provide an additional experiment showing that once calibrated, our static model works well enough to distinguish between good and bad rewrites.

An initial calibration step is a common practice with big data since the query operator cost and specific data statistics are not immediately available. The static calibration approach that we take is similar to [89, 91] which also calibrate cost estimates one time, up front, per system configuration. A dynamic approach would be similar to [38, 50, 70] that run a micro-job per UDF to obtain initial cost estimates, and then update cost estimates during runtime to optimize iterative UDFs (e.g., k-means func-

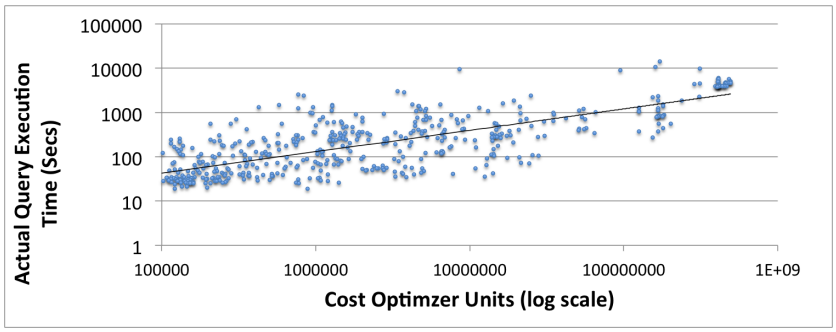

Figure 4.16: MR job execution time vs.estimated cost using our cost model (log-scale). Each point represents an MR job in the 32 query workload.

tions). However, the focus of those works is not query rewriting but fine-grained tuning of iterative MR jobs to handle data skew at runtime. As noted in the previous paragraph, adapting our techniques to be dynamic is possible but care must be taken to control the amount of overhead required for continuous fine-grained tuning. After performing our calibrations, next we show the effectiveness of our cost model by reporting actual execution time versus the expected cost as computed by our cost model.

Figure 4.16 plots actual execution time in seconds along the y-axis and estimated cost as determined by our cost model along the x-axis. Each point in Figure 4.16 represents an MR job executed by a query in the workload. After calibration of each UDF we executed all 32 queries of the workload in a realistic setting on our 20-node cluster described in our Experimental Methodology (Section 4.7.2), where multiple MR jobs may execute concurrently. A trend line is added to the figure as a visual aid.

The trend shown in Figure 4.16 shows the cost estimates generally serve their purpose to differentiate between good plans and really bad plans. Estimating cost is significantly complicated in this space due to several factors: "big data" has not

been ETL'd and thus lacks good statistics and may be skewed; UDFs are comprised of arbitrary user code; and MR jobs are scheduled on shared resources.

These challenges notwithstanding, the results in Figure 4.16 along with our experimental results in Figures 4.7–4.12 show that estimates produced by our cost model are good enough to (a) guide the search such that BfRewrite examines far-fewer candidates than DP (Figure 4.11(a)) and (b) significantly reduce query execution time since the rewritten query always executed much faster than the original query, up to an order of magnitude (Figures 4.7–4.10).

Taken together, these results show that although we calibrate our cost model only the first time the UDF is added, it is able to discriminate between good plans and really bad plans for the purpose of query rewriting.

4.7.5 Comparison with DBMS-X

In this section we evaluate the quality of the rewrites produced by BfRewrite by comparing with the rewrites produced by a commercial DBMS. The ability of BfRewrite to improve query execution time is in part due to the amount of overlap between the queries in the workload, the nature of the data, and the presence of UDFs in the workload. Here we show that improved query execution time is also due in part to the quality of the rewrites found by BfRewrite. This experiment provides a sanity check for our methods, validating experimentally that BfRewrite is indeed producing good-quality rewrites.

In this experiment, we use base data from the TPC-H [95] benchmark and

we create 3 simple relational queries that do not contain UDFs in order to provide a more direct comparison of methods that is less dependent on the nature of the data and the workload. While not exhaustive, the following experiment provides some evidence to verify that our method indeed produces good quality rewrites that are comparable to (in some cases better than) a widely-used commercial database system denoted by DBMS-X.

The experimental setup contains two materialized views (MV_1 and MV_2) and three simple queries for the comparison. We first created two materialized views on the `LINEITEM` table: MV_1 is a GROUPBY–COUNT on `L_ORDERKEY, L_SUPPKEY`, and MV_2 is a GROUPBY–COUNT on `L_ORDERKEY, L_PARTKEY`. We also refreshed the optimizer's statistics on both the base table and the materialized views. We then created the following three queries that reference the large `LINEITEM` table (this is the largest table in TPC-H), and used the procedure described below to compare the rewrites produced by BfRewrite and DBMS-X.

```
(Q1) SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY
     FROM LINEITEM
     GROUP BY L_ORDERKEY, L_PARTKEY,L_SUPPKEY;

(Q2) SELECT L_SUPPKEY, COUNT(*) AS C1
     FROM LINEITEM
     GROUP BY L_SUPPKEY;

(Q3) SELECT L_SUPPKEY, COUNT(*) AS C2, MAX(L_SUPPKEY) AS C5
     FROM LINEITEM
     GROUP BY L_SUPPKEY;
```

First, DBMS-X was tuned to its highest optimization level and was configured to consider all views for query rewriting. Next, we obtained rewrites for each query

($Q1, Q2, Q3$) from both BfRewrite and DBMS-X. Lastly, to obtain common cost units for a direct comparison, we use DBMS-X's `EXPLAIN QUERY` facility to obtain a cost estimate for the rewritten query produced by BfRewrite and the rewritten query produced by DBMS-X. The `EXPLAIN QUERY` facility uses DBMS-X's query optimizer to provide a cost estimate for both rewrites (without further optimizations) in DBMS-X units.

- Q1 results: BfRewrite's solution joined views MV_1 and MV_2 to rewrite the query, while DBMS-X did not make use of either materialized view to rewrite the query. DBMS-X's query cost was 37,352 units while BfRewrite's rewritten query cost was 12,106 units; representing a 3× improvement over DBMS-X.
- Q2 results: Both BfRewrite and DBMS-X produced identical rewrites (making use of MV_2) with a rewritten query cost of 6,792 units.
- Q3 results: BfRewrite's solution applied a GROUPBY on MV_2, recomputing the sum and max; while DBMS-X did not make use of either materialized view to rewrite the query. DBMS-X's query cost was 35,955 units, while BfRewrite's rewritten query cost was 6,806 units; representing a 5× improvement over DBMS-X.

This experimental result suggests that BfRewrite produces rewrites that are at least competitive with a state-of-the-art commercial DBMS.

4.7.6 Microbenchmark

Here we develop a microbenchmark to evaluate the sensitivity of BfRewrite as we vary the "amount" of change between query revisions. As noted earlier in Sec-

tion 4.7.1, quantifying the amount of change is not at all straightforward, and indeed our microbenchmark results show that even minor query revisions can affect the reusability of previous results. First in Section 4.7.6.1 we describe our findings from a literature review to identify the ways in which queries commonly evolve. This helps us to determine in what ways we can vary and control the amount of change (e.g., overlap) between a sequence of query revisions. We use the "types" changes defined in our previous work [61], and provide supporting evidence for these changes here. Second, we provide our microbenchmark results in Section 4.7.6.2 showing how well BfRewrite can improve query execution time by utilizing results from the previous versions of a "similar" query. This study is imperfect, but provides some insight into the benefits of reusing previous results as queries are modified from one version to the next.

4.7.6.1 Designing the Microbenchmark

To design the microbenchmark, we started out by performing a review of literature on the kinds of revisions users perform within a query session. A query session is defined as the query trace as the user is trying to find the information he or she seeks. Below we summarize our findings that guided the design of our microbenchmark, and then present the results.

1. The work in [18] defines query sessions as a sequence of queries where each query forms the jumping-off point for the next. The authors define a "Steering Algebra" of the common type of action performed during query exploration. They provide four common patterns of revisions across subsequent versions of queries: Narrow,

DrillDown, Move, and Relate, Narrow. The Narrow restricts the result set to produce less answers, DrillDown adjusts group-by and changing aggregate functions, Move replacing values in predicate with different values, and Relate applies a join operator as a way of finding related information from another table (or log). Incidentally, the authors extol the potential benefits from caching, prefetching and "check pointing" answers for future reuse, which is in part the approach we take with our opportunistic design approach.

2. The recently updated UC Berkeley AMPLab benchmark (`https://amplab.cs.berkeley.edu/benchmark/`) provides 3 queries each with 3 revisions. Query 1 changes the selection predicate, Query 2 changes group-by, while Query 3 changes Join Predicates. They provide a fourth query containing a simple UDF but provide no revisions for this query. Each of Q1, Q2, Q3 has 3 versions, which vary the cardinality of tuples in the result set from small/med/large sizes, to mimic a small/med BI-query and a larger result ETL-style query.

3. [52] discusses a use case for query relaxation where predicates are automatically adjusted to obtain to obtain "enough answers" (i.e., more answers) in contrast with "top k" that is meant to provide fewer answers.

4. Our previous work [61] examines interactive and exploratory queries in several domains by analyzing query revisions in the TPC-DS benchmark (business data) and real-world queries from Taverna (scientific data) and Yahoo Pipes (web data). Four common types of changes are defined: *P, L, U, G*, outlined below and show in

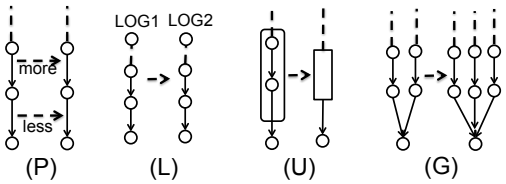

Figure 4.17: Dimensions of change (PLUG) between query versions as defined in [61]

Figure 4.17. We provide a more detailed description later in Chapter 6.

- (P) Parameters: An analyst may modify the query's operations to allow for more or less tuples in the results (Figure 4.17P). For example, the analyst may alter a select predicate or a top-k value to allow more or less data in the output.
- (L) Logs: An analyst may augment a query with a new log (i.e., another data source) to obtain richer results (Figure 4.17L).
- (U) UDFs: An analyst may replace a set of operations in the query with a specialized UDF (Figure 4.17U).
- (G) Sub-Goal: An analyst writes several sub-queries such that each achieves a single goal and then joins these to obtain the final output (Figure 4.17G).

Our review of the existing literature informs us that the PLUG dimensions capture most of the commonly observed changes between subsequent versions of an analyst's data exploration. Varying parameters (i.e., P dimension) also captures the Narrow and DrillDown operations from [18], Queries 1–3 from the AMP Lab workload and query expansion from [52]. Adding a new log to the analysis (i.e., L dimension) captures the Relate operation from [18]. Replacing operations with a specialized UDF (i.e., U dimension) is similar to an evolution on Query 4 of the AMPlab. Adding a

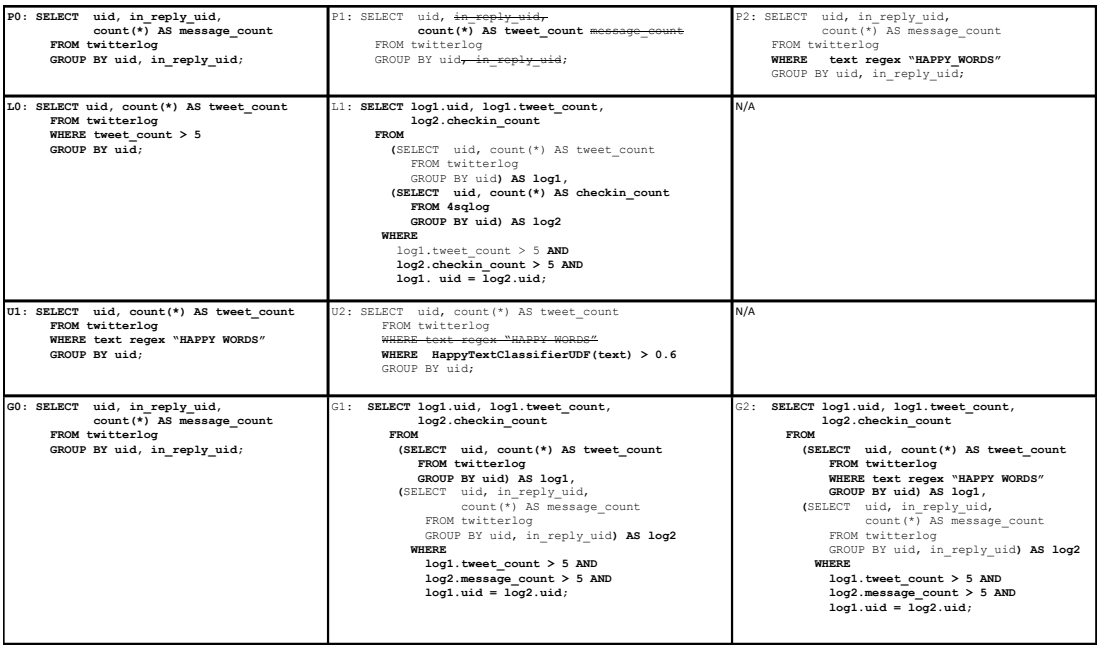

Figure 4.18: Illustration of the microbenchmark queries showing the changes from one version to another. The left-most column shows the original version of each query, the next columns show the revised versions with additions in gray and removals with strike-through.

sub-goal (i.e., G dimension) is similar to adding a RELATE from [18]. Since the PLUG dimensions seem to be the most comprehensive among the literature we surveyed, we design a *microbenchmark* based on $\{P, L, U, G\}$.

Using $\{P, L, U, G\}$ as building blocks, we devise a base query for each of the PLUG dimensions denoted as $P0$, $L0$, $U0$, $G0$ where we can vary and control the specific type of change. Each base query is then revised according to its assigned dimension. For example, $P0$ is revised several times along the P dimension only, while similarly $G0$ is revised several times along the G dimension only. The base queries and their revisions are shown in Figure 4.18. From top to bottom, the first column shows the base query for each of the PLUG dimensions (P0, L0, U0, G0). From left to right,

each base query is revised according to the dimension. For instance, base query P0 is revised into P1 by modifying the GROUPBY clause, and then P0 is revised into P2 by modifying the WHERE clause. Each revision is highlighted by strike-throughs for query text removals and bold for query text additions with respect to the base query. We evaluate the performance of the rewrite algorithm for each revision (e.g. P1), given only the views generated by the base query (e.g. P0).

4.7.6.2 Microbenchmark Results

Our microbenchmark experimental setup consists of Hive on a small cluster of 3 machines. Each query accesses data from a Twitter log containing user tweets stored as text in JSON format, one tweet per line. Hence for each query there is an implicit extract of the necessary attributes from the log using the associated SerDe (for JSON text here) as is standard in Hive. This is performed by Hive as an MR job which then serves as input to the subsequent MR job in the query. The microbenchmark results are shown in Figure 4.19 where the PLUG dimensions are along the x-axis and the execution time is along the y-axis for both the original query (ORIG), and the query as rewritten by our algorithm (BFREWRITE).

Along the P dimension, BFREWRITE shows significant improvement in execution time for P1 as compared to ORIG. However, although P2 has a large "amount" of query text in common with P0, BFREWRITE provides no improvement over ORIG. Executing P0 produced two opportunistic views, one corresponding to the attributes extracted from the Twitter log and another corresponding to the final query result after

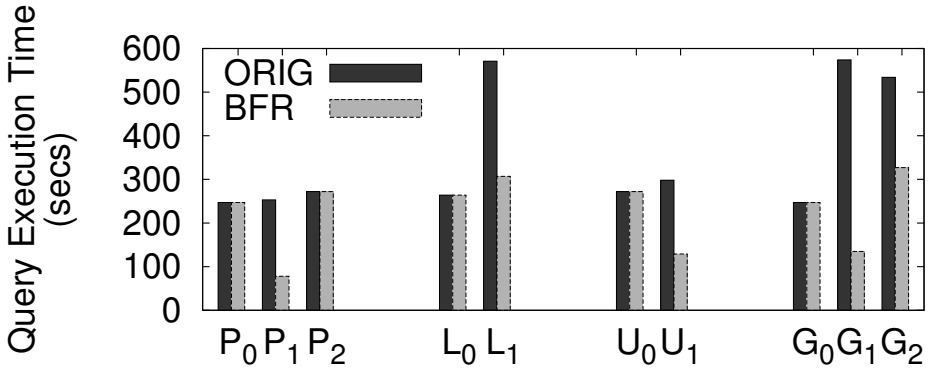

Figure 4.19: Microbenchmark results showing original execution time (ORIG) and rewritten query execution time of BfRewrite (BFR) for queries revised along each of the P, L, U, G dimensions.

the application of the GROUPBY operator. The reason for P1's significant improvement over ORIG is that P1 applied additional transformations to the final query result of P0, whereas P2 had to go back to the log to obtain the additional "text" attribute. In our experience in designing the microbenchmark, we found that it was not clear that methods such as [20, 55] that use query fragment similarity (e.g., WHERE clauses, FROM clauses) to quantify the amount of overlap between queries are useful to predict the benefit from reusing previous results. For instance, we observed that even a seemingly trivial change to a query could render it not conducive to rewriting, while a larger change could be "rewrite friendly". Queries P1 and P2 from Figures 4.19 and 4.18 are an example of this effect.

Along the L dimension, L1 augments L0 by including an additional log data source in the query. For L1, ORIG must execute the query using both logs, resulting in a high execution time. BfRewrite must execute the query on the new log, but is able

to re-use the results from the query on the first log, resulting in significant execution time savings. However, the savings is obviously dependent upon the relative sizes of the old and new logs.

Along the U dimension, U1 modfies U0 by replacing the REGEX that matches "happy" words in the tweet text with a specialized UDF that classifies tweet text with a "happy" score. For U1, the execution time of ORIG is a little longer than ORIG for U0 since the UDF is more expensive than REGEX. BFREWRITE provides significant savings for U1 by reusing the previously extracted attributes from the log. While this experiment only tests the effect of replacing a query operation with a specialized UDF, we found that in general, the savings due to rewrite is sensitive to the position of the UDF in the query.

Along the G dimension, G1 augments G0 with an additional subquery and a WHERE clause. For G1, the execution time of the ORIG query is much higher than G0 due to the subquery addition. However, BFREWRITE is able to reuse results and avoids repeating the first query as well as rewriting the new subquery with results from the first query, thus realizing a significant savings in execution time. G2 augments G0 in the same way as G1, but includes an additional WHERE clause looking for "happy" words. BFREWRITE provides significant savings over ORIG for G2 by reusing results from G0, although the savings is not as much as for G1. The savings for revisions along the G dimension depends on how much the newly added subgoal can benefit from the computed results of the original query.

In summary, our microbenchmark results again show the importance of reusing

previous results in a big data exploratory query scenario and it highlights that the expected savings is difficult to infer or directly correlate with the "type" of change. Clearly there is more work needed in this area, and our findings highlight that due to this problem, a semantic rewriting method such as BfRewrite is more robust and hence potentially more beneficial than simply caching previous results.

4.8 Related Work

Query Rewriting Using Views. There is a rich body of previous work on rewriting queries using views, but these only consider a restricted class of queries. Representative work includes the popular algorithm MiniCon [79], recent work [58] showing how rewriting can be scaled to a large number of views by applying a form of sub-graph matching among queries and views, rewriting methods implemented in commercial databases [41, 101], and [59] which extends [58] to consider multiple queries. However, in these works both the queries and views are restricted to the class of conjunctive queries (SPJ) or may additionally include groupby and aggregation (SPJGA).

Our work differs in the following two ways: (a) We show how UDFs can be included in the rewrite process using our UDF model, which results in a unique variant of the rewrite problem when there is a divergence between the expressivity of the queries and that of the rewrite process; (b) Our rewrite search process is cost-based — OptCost enables the enumeration of candidate views based on their ability to produce a low-cost rewrite. In contrast, traditional approaches (e.g., [41,79]) typically determine containment first (i.e., if a view can answer a query) and then apply cost-based pruning

in a heuristic way. This unique combination of features has not been addressed in the literature for the rewrite problem.

Online Physical Design Tuning. Methods such as [83] adapt the physical configuration to benefit a dynamically changing workload by actively creating or dropping indexes/views. Our work is opportunistic, and simply relies on the by-products of query execution that are almost free. However, online view selection methods could be applicable during storage reclamation to retain only those opportunistic views that provide maximum benefit. Since storage space is not infinite, this is an interesting problem and we address a unique variant of this problem next in Chapter 5.

Reusing Computations in MapReduce. Other methods for optimizing MapReduce jobs have been introduced such as those that support incremental computations [65] (in-flight data sharing), sharing computation or scans [72], and re-using previous results [37]. These approaches operate on lower-level constructs (e.g., query plans) than our approach. Syntactic-based sharing methods are obviously effective but provide limited benefit compared to our approach. As shown in Section 4.7.3.4, our semantic-based approach completely subsumes these methods.

Multi-Query Optimization (MQO). The goal of MQO [87] (and similar approaches [72]) is to maximize resource sharing, in particular common intermediate data, by producing a scheduling strategy for a set of queries. Thus the output of MQO is a scheduling strategy that minimizes the amount of duplicated work among the queries.

Our goal is to rewrite a query to take advantage of the available materialized views, thus the output of our work is a low-cost rewrite rather than a scheduling policy for multiple concurrent queries.

Chapter 5

MISO: Souping Up Big Data Query Processing with a Multistore System

Parallel relational database management systems (RDBMS) have long been the "work horse" storage and query processing engine of choice for data warehousing applications. Recently there has been a substantial move toward "Big Data" systems for massive data storage and analysis; HDFS with Hive is perhaps the most common example of such a system. The arrival of big data stores has set off a flurry of research and development, initially dealing with questions such as which store is best suited for which purpose; however, the growing consensus is that both stores have their place, and many organizations simultaneously deploy instances of each. Currently these stores have different roles – the big data store for exploratory queries to find business insights and the RDBMS for business reporting queries, as depicted in Figure 5.1(a).

Having both a parallel RDBMS and a big data store inevitably raises questions about combining the two into a "multistore" system, either for performance reasons or for analysis of data sets that span both stores. There have been a number of interesting proposals for this combination [3, 36, 45, 89]. In this work, we explore a simple yet

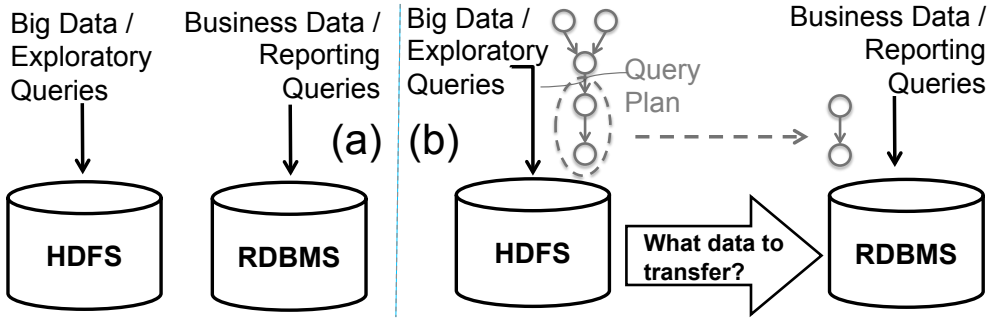

Figure 5.1: Typical setup in today's organizations showing (a) two independent stores and (b) multistore system utilizing both stores.

powerful and ubiquitous opportunity: using the parallel RDBMS as an accelerator for big data queries, while preserving their above-mentioned roles.

Multistore systems represent a natural evolution for big data analytics, where query processing may span both stores, transferring data and computation as depicted in Figure 5.1(b). One approach to multistore processing is to transfer and load all of the big data into the RDBMS (i.e., up-front data loading) in order to take advantage of its superior query processing performance [77] relative to the big data store. However, the large size of big data and the high cost of an ETL process (Extract-Transform-Load) may make this approach impractical [77, 88]. Another approach is to utilize both stores during query processing by enabling a query to transfer data on-the-fly (i.e., on-demand data loading). However, this results in redundant work if the big data workload has some overlap across queries, as the same data may be repeatedly transferred between the stores.

A more effective strategy for multistore processing is to make a tradeoff between up-front and on-demand data loading. This is challenging since exploratory

queries are ad-hoc in nature and the relevant data is changing over time. A crucial problem for a multistore system is determining what data to materialize in which store at what time. We refer to this problem as tuning the physical design of a multistore system.

In this work we present the first method to tune the physical design of a multistore system. While physical design tuning is a well-known problem for a conventional RDBMS, it acquires a unique flavor in the context of multistore systems. Specifically, tuning the physical design of a multistore system has two decision components: *what* data to materialize (i.e., which materialized views) and *where* to materialize the data (i.e., which store). Our primary focus in this work is on the *where* part since this is critical to good multistore query performance. Furthermore, since the different stores are coupled together by a multistore execution engine, the decision of where to place data creates a technically interesting problem that has not been addressed in the existing literature. Beyond its technical merits, we argue that the problem actually lies at the core of multistore system design — regardless of how multistore systems are setup (i.e., tightly versus loosely coupled) or the way they are used (i.e., enabling data analysis spanning stores versus query acceleration). Without a well-tuned design, a multistore system will be forced to split query processing between the big data store and the RDBMS in a way that does not leverage the full capabilities of each system.

We introduce *MISO*, a MultIStore Online tuning algorithm, to "soup up" big data query processing by tuning the set of materialized views in each store. MISO maximizes the utilization of the existing high-performance RDBMS resources, employing

some of its spare capacity for processing big data queries. Given that the RDBMS holds business data that is very important to an organization, DBAs are naturally protective of RDBMS resources. To be sensitive to this concern, it is important that our approach achieves this speedup with little impact on the RDBMS reporting queries. MISO is an adaptive and lightweight method to tune data placement in a multistore system, and it has the following unique combination of features:

- It works online. The assumed workload is ad-hoc since analysts continue posing and revising queries. The multistore design automatically adapts as the workload changes dynamically.

- It uses materialized views as the basic element of physical design. The benefits of utilizing materialized views are that views encapsulate prior computation, and the placement of views across the stores can obviate the need to do costly data movement on-the-fly during query processing. Having the right views in the right store at the right time creates beneficial physical designs for the stores.

- It relies solely on the by-products of query processing. Query processing using HDFS (e.g., Hadoop jobs) materializes intermediate results for fault-tolerance. Similarly, query processing in multistore systems moves intermediate results between the stores. In both cases we treat these intermediate results as *opportunistic* materialized views [63], which can be strategically placed in the underlying stores to optimize the evaluation of subsequent queries. These views represent the current relevant data and provide a way to tune the physical design almost for free, without

the need to issue separate costly requests to build materialized views.

Online physical design tuning has been well-studied, but previous works have not considered the multistore setting. Other previous work has studied query optimization for multistore systems in the context of a single query accessing data in different stores. In contrast, our proposed technique allows the multistore system to tailor its physical design to the current query workload in a low-overhead, "organic" fashion, moving data between the stores depending on common patterns in the workload. A key metric for evaluating big data queries is the time-to-insight (TTI) [60], since this metric represents the total time to get answers from data — which includes more than query execution time. TTI includes data-arrival-to-query time (the time until new data is queryable), the time to tune the physical design, *and* the query execution time. TTI represents a holistic time metric for ad-hoc workloads over big data, and our approach results in significant improvements for TTI compared to other multistore approaches.

In the following sections we describe the MISO approach to speeding up big data queries by utilizing the spare capacity available in the RDBMS, performing lightweight online tuning of the physical design of the stores. In Section 5.5 we provide an overview of the related work, and we describe our multistore system architecture in Section 5.1. We describe the problem of multistore design and its unique challenges in Section 5.2. In the remainder of the section we present MISO, which solves the multistore design problem by tuning both stores. Since we take an online approach, the design adapts to changing workloads. Furthermore, as the design is opportunistic, MISO imposes little additional overhead. In Section 5.3 we provide experimental results

showing the benefits of MISO in improving TTI, up to 3.1×, and further analyze where the improvements come from. Then in Section 5.3.4, we show that MISO has very little impact on an RDBMS that is already executing a workload of reporting queries. Finally, we summarize our findings in Section 5.4.

5.1 System Architecture and Multistore Tuning

In a multistore scenario there are two data stores, the big data store and the RDBMS. In our system, the big data store is Hive (HV) and the RDBMS is a commercial parallel data warehouse (DW) as depicted in Figure 5.2. HV contains the big data log files, and DW contains analytical business data. While some multistore systems enable a single query to access data that spans both stores, MISO accelerates big data exploratory queries in HV by utilizing some limited spare capacity in DW and tuning the physical design of the stores for a workload of queries. In this way, MISO makes use of existing resources to benefit big data queries. With this approach, DW acts as an *accelerator* in order to improve the performance of queries in HV; later we show that MISO achieves this speedup with very little impact on DW.

Figure 5.2 depicts our system architecture, containing the *MISO Tuner*, the *Query Optimizer*, the *Execution Layer*, and the two stores HV and DW. In our system, each store resides on an independent cluster. HV contains log data while DW contains analytical data, but in this work our focus is on accelerating big data queries posed only on the log data. As indicated in the figure, each store has a set of materialized *views* of the base data (i.e., logs) stored in HV, and together these views comprise the multistore

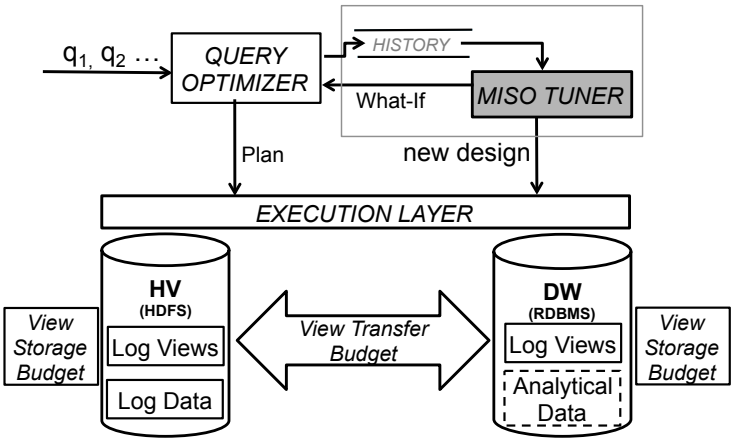

Figure 5.2: Multistore System Architecture.

physical design.

For fault-tolerance, Hadoop-based systems (e.g., HV) materialize intermediate results during query execution, and we retain these by-products as *opportunistic* materialized views. On the other hand, DW products generally do not provide access to their intermediate materialized results. However, if those results become available, then they can also be used for tuning. In our system, we consider the class of opportunistic materialized views when tuning the physical design. It is MISO Tuner's job to determine the placement of these views across the stores to improve workload performance. When placing these views, each store has a view storage budget as indicated in the figure, and there is also a transfer budget when moving views across the stores. These budgets limit the total size of views that can be stored and transferred.

5.1.1 System Components

Data sets. The primary data source in our system is large log files. Here we expect social media data drawn from sites such as Twitter, Foursquare, Instagram, Yelp, etc.

This type of data is largely text-based with little structure. Logs are stored as flat HDFS files in HV in a text-based format such as JSON or XML.

Queries. The input to the system is a stream of queries $\langle q_1, q_2, \cdots \rangle$ as indicated in Figure 5.2. The query language is HiveQL, which represents a subset of SQL implemented by Hive (HV). Queries are declarative and posed directly over the log data, such that the log schema of interest is specified within the query itself and is extracted during query execution. In HV, extracting flat data from text files is accomplished by a SerDe (serialize/deserialize) function that understands how to extract data fields from a particular flat file format (e.g., JSON). A query q may contain both relational operations and arbitrary code denoting user-defined functions (UDFs). UDFs are arbitrary user code, which may be provided in several languages (e.g., Perl, Python); the UDFs are executed as Hadoop jobs in HV. Queries directly reference the logs and are written in HiveQL, the system automatically translates query sub-expressions to a DW-compliant query when executing query sub-expressions in DW. The processing location (DW or HV) is hidden from the end user, who has the impression of querying a single store.

Execution Layer. This component is responsible for forwarding each component of the execution plan generated by the *Query Optimizer* to the appropriate store. A multistore execution plan may contain "split points", denoting a cut in the plan graph whereby data and computation is migrated from one store to the other.

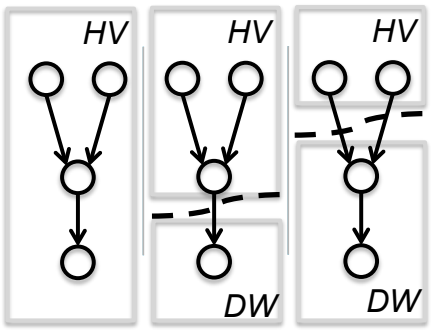

Figure 5.3: A plan with several possible split points.

Since DW is used as a accelerator for HV queries, the splits in a plan move data and computation in one direction: from HV to DW. It is the multistore query optimizer's job (described next) to select the split points for the plan. As an example, the figure alongside has three panels, showing an execution plan (represented as a DAG) and then two possible split points indicated by the cuts. At each split point, the execution layer migrates the intermediate data (i.e., the working set corresponding to the output of the operators above the cut) and resumes executing the plan on the new store. Note that in the second panel, one intermediate data set needs to be migrated by the execution layer, while in the third panel two intermediate data sets need to be migrated. When intermediate data sets are migrated during query execution, they are stored in temporary DW table space (i.e., not catalogued) and discarded at the end of the query. The execution layer is also responsible for orchestrating the view movements when changes to the physical design are requested by the tuner. When views are migrated during tuning, they are stored in permanent DW table space and become part of the physical design until the next tuning phase.

Multistore Query Optimizer. This component takes a query q as input, and computes a multistore execution plan for query q. The plan may span multiple stores, moving data and computation as needed, and utilizes the physical design of each store.

Multistore query optimizers are presented in [36, 89], and our implementation is similar to that in [89]. As noted in [89,91], the design of an optimizer that spans multiple stores must be based on common cost units (expected execution time) between stores, thus some unit normalization is required for each specific store. Our multistore cost function considers three components: the cost in HV, the cost to transfer the working set across the stores, and the cost in DW, expressed in normalized units. For HV we use the cost model given in [72] and for DW we use its own cost optimizer units obtained from its "what-if" interface. We performed experiments to calibrate each system's cost units to query execution time, similar to the empirical method used in [89]. As it turns out, when the necessary data for q was present in DW, it was always faster to execute q in DW by a very wide margin. Because the stores have very asymmetric performance, the HV units completely dominate the DW units, making a rough calibration/normalization of units sufficient for our purposes.

Because the choice of a split point affects query execution time, it is important to choose the right split points. The multistore query optimizer chooses the split points based on the logical execution plan and then delegates the resulting sub-plans to the store-specific optimizers. The store in which query sub-expressions are executed depends on the materialized views present in each store. Furthermore when determining split points the optimizer must also consider valid operations for each store, such as a UDF that can only be executed in HV. Moving a query sub-expression from one store to another is immediately beneficial only when the cost to transfer and load data from one store to another plus the cost of executing the sub-expression in the other store is *less*

than continuing execution in the current store. Furthermore, due to the asymmetric performance between the stores we observed that when the data (views) required by the sub-expression was already present in DW, the execution cost of the sub-expression was always lower in DW. The primary challenge for the multistore query optimizer is determining the point in an execution plan at which the data size of a query's working set is "small enough" to transfer and load it into DW rather than continue executing in HV.

In our implementation, we have added a "what-if" mode to the optimizer, which can evaluate the cost of a multistore plan given a hypothetical physical design for each store. The optimizer uses the rewriting algorithm from [61] in order to evaluate hypothetical physical designs being considered by the MISO Tuner.

MISO Tuner. This componentis invoked periodically to reorganize the materialized views in each store based on the "latest traits" of the workload. The tuner examines several candidate designs and analyzes their benefit on a sliding window of recent queries (*History* in Figure 5.2) using the what-if optimizer. The selected design is then forwarded to the execution layer, which then moves views from HV to DW and from DW to HV (indicated in Figure 5.2) as per the newly computed design. The invocation of the MISO tuner, which we term a *reorganization phase*, can be time-based (e.g., every 1h), query-based (e.g., every n queries), activity-based (e.g., when the system is idle), or a combination of the above. In our system, reorganizations are query-based.

The *View Transfer Budget* is indicated in Figure 5.2 by the arrow between

the stores. This represents the total size of views in GB transferred between the stores during a reorganization phase and is provided as a constraint to the tuner. The HV *View Storage Budget* and the DW *View Storage Budget* are also indicated in the figure and similarly provided as constraints to the tuner. These represent the total storage allotted in GB for the views in each store. While DW represents a tightly-managed store, HV deployments are typically less tightly-managed and may have more spare capacity than DW. For these reasons, new opportunistic views created in HV between reconfigurations are retained until the next time the MISO tuner is invoked, while the set of views in DW is not altered except during reorganization phases. In any Hadoop configuration, there must be enough temporary storage space to retain these materializations during normal operation, we only propose to keep them a little while longer – until the next reorganization phase. Since the HV view storage budget is only enforced at tuning time, a few simple policies can be considered if the system has limited temporary space: Reorganizations could be done more frequently, a simple LRU policy could be applied to the temporary space, or a small percentage of the HV view storage budget could be reserved as additional temporary space.

5.1.2 Overview of Multistore Tuning

Tuning the physical design of a multistore system is important to obtaining good query performance. In a multistore system, the query optimizer can choose many different split points at which to migrate computation from one store to the other during query evaluation. The overall execution time of a query is dependent on the existing

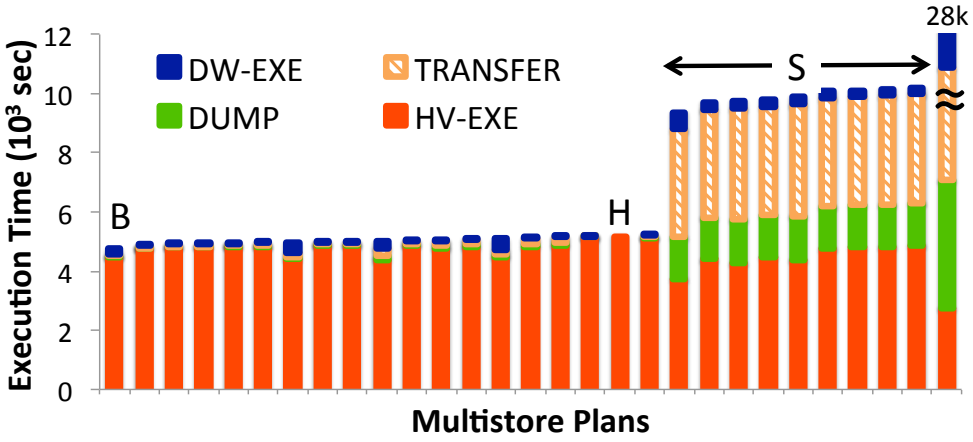

Figure 5.4: Execution time profile of all multistore plans for a single query. Plans are ordered by increasing execution time.

physical design of each store, and varies depending on which portion of the query is evaluated in which store. A well-tuned physical design enables query execution plans to migrate to the DW sooner rather than later, thus making more extensive use of the DW's superior query-processing capabilities. In contrast, without a good physical design, the high cost of data movement will cause a query to spend more time in the slower HV store, which in turn provides limited opportunities for optimization. We demonstrate this point with the following simple experiment.

Figure 5.4 provides the execution time profiles (ordered from lowest to highest) for all possible plans of a single query, where each plan represents a unique split. The query is a complex query from [61] (query A_1v_1), that accesses social media data (e.g., Twitter, Foursquare), has multiple joins, computes several aggregates, and contains UDFs. These queries relate to marketing scenarios for a new product, and they try to identify potential customers based on their interests as identified in social media

data. While the figure only shows one query, the trends observed were typical of all queries that we tested. Each plan begins execution in HV, and for each split point, the intermediate data is migrated to DW where the remainder of the plan is executed. For each plan, the stacked bar indicates the time spent executing in HV (red), the time spent to DUMP (green) and then TRANSFER/LOAD (yellow) the intermediate data over the network into DW, and finally the time spent executing in DW (blue).

The plan with the lowest execution time is on the far left of the figure, marked B. This plan is only 10% faster than the HV-only execution plan marked H. The worst plan in the figure is on the far right, at 28,000 seconds, and the bar is truncated here for display purposes. This plan represents the earliest possible split point, using HV only as an ETL engine to load all data into DW and then execute the query. From these observations it is clear that multistore execution offers little benefit for a single query (10% in the best case), and many multistore plans (those marked S) are *far more* expensive than the HV-only plan. This is due to the very high cost to transfer and load data from HV to DW, as can be seen in the yellow and green portions of the plans marked S. Additionally, there is a clear delineation between "good" plans (those to the left of S) and "bad" plans (those marked S).

We have found that the good plans all dump, transfer and then load much smaller data sets into DW than the bad plans, as shown in Figure 5.4. Good plans represent splits where the size of the intermediate data is small enough to offset the transfer and load costs. Unfortunately, we typically observed that the working set size does not shrink significantly until near the end of the query. This effect was also observed in

Bing MapReduce workflows [4] (i.e., the "little data" case). Another recent paper [89] examines a related experimental setup with two stores, and their optimizer considers ping-ponging a query (multiple sync/merges) back and forth across both stores. An examination of their experimental results reveals that the queries with the best performance are those that remain in Hadoop until the working set is small before spilling their remaining computation to the DW; effectively degenerating into the single-split multistore plans that we consider here. Hence, a query is only able to utilize DW for a small amount of processing, not benefiting much from DW's superior performance.

Tuning the physical design of HV and DW can facilitate earlier split points for multistore query plans. Better still, when the right views are present a query can bypass HV and execute directly in DW, thus taking advantage of DW's superior performance. The role of the tuner is to move the "right views" into the "right store". Again, we illustrate this point with another experiment on the same multistore system. In Figure 5.5 alongside, we consider a workload of two exploratory queries, q_1 and q_2, that correspond to A_1v_2 and A_1v_3 from the workload in [61]. These are subsequent queries issued by the same data analyst, and thus have some overlap. Query q_1 is executed first, followed by q_2, and each query produces some opportunistic views when executed in HV. The figure reports the total execution time for q_1 followed by q_2 using three different system variants. HV-ONLY represents an HV system that executes both queries in their entirety and does not utilize DW at all.

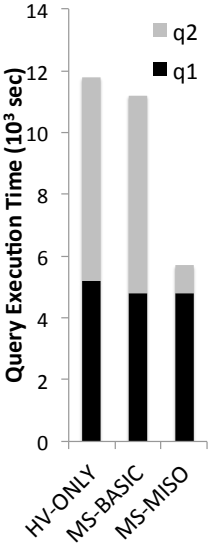

Figure 5.5: Two query workload on several system variants.

MS-BASIC represents a basic multistore system without tuning, and is the same as the setting in Figure 5.4. This system does not make use of opportunistic views to speed up query processing. MS-MISO represents our multistore system using the MISO Tuner. To produce the MS-MISO result in the figure, we triggered a reorganization phase after q_1 but before q_2 executed. During reorganization, the Tuner transfers views between the stores in order to create beneficial designs for q_2. In the figure, MS-BASIC is only about 8% faster than HV-ONLY; while MS-MISO is about 2× faster than both HV-ONLY and MS-BASIC. While this result is for 2 related queries, for a larger workload MS-MISO results in up to 3.1× improvement later in Section 5.3. MS-MISO's improvement over MS-BASIC is because the tuner created a new multistore physical design after q_1, which enabled q_2 to take advantage of DW performance since the "right" views were present in DW.

5.2 MISO Tuner

In this section we describe the details of the MISO Tuner component shown in Figure 5.2. Because we address the online case where new multistore designs will be computed periodically, we desire a solution that is lightweight with low-overhead. This is achieved in the following ways. Since we utilize opportunistic views, the elements of our design are almost free, thus the tuner does not incur an initial materialization cost

for a view. The reorganization time is controlled by the view transfer budget and view storage budgets for HV and DW. The values for these constraints can be kept small, in order to restrict the total tuning time. At the beginning of each reorganization phase, the tuner considers migrating views between the stores based on the recent workload history (indicated in Figure 5.2) and computes the new physical designs for HV and DW, subject to the view storage budget constraints and the view transfer budget.

Multistore systems notwithstanding, solving even a single physical design problem is computationally hard [23] because the elements of the design can interact with one another. Commercial tools (e.g., IBM DB2 Design Advisor [96]) take a heuristic approach to the physical design problem for a single store by treating it as a variant of the knapsack problem with a storage budget constraint. We similarly treat the multistore design problem as a variant of a knapsack problem. Our scenario is different in that we have two physical designs to solve, where each is dependent on the other, and each design has multiple dimensions — a view storage budget constraint and a view transfer budget constraint — that are consumed at different rates depending on the current design of each store. Due to the hardness of the problem, an optimal solution is impractical thus we take a heuristic approach. In order to keep our approach lightweight, we develop an efficient dynamic programming based knapsack solution.

5.2.1 Basic Definitions and Problem Statement

The elements of our design are materialized views and the universe of views is denoted as V. The physical design of each store is denoted as V_h for HV, and V_d for

DW, where $V_h, V_d \subseteq V$. The values B_h, B_d denote the view storage budget constraints for HV and DW respectively, and the view transfer budget for a reorganization phase is denoted as B_t. As mentioned previously, reorganization phases occur periodically (e.g., after j queries are observed by the system), at which point views may be transferred between the stores. The constraints B_h, B_d, and B_t are specified in GB.

Let $\mathcal{M} = \langle V_h, V_d \rangle$ be a *multistore design*. $\mathcal{M}$ is a pair where the first component represents the views in HV, and the second component represents the views in DW. In this work we restrict the definition to include only two stores, but the definition may be extended to include addition stores in the more general case.

We denote the most recent n queries of the input stream as workload W. A query q_i represents the i^{th} query in W. The cost of a query q given multistore design $\mathcal{M}$, denoted by $cost(q, \mathcal{M})$, is the sum of the cost in HV, the cost to transfer the working set of q to DW and the cost in DW under a hypothetical design $\mathcal{M}$. The evaluation metric we use for a multistore design $\mathcal{M}$ is the total workload cost, defined as:

$$TotalCost(W, \mathcal{M}) = \sum_{i=1}^{n} cost(q_i, \mathcal{M}). \tag{5.1}$$

Intuitively, this metric represents the sum of the total time to process all queries in the workload. This is a reasonable metric, although others are possible. Note that the metric does not include the reorganization constraint B_t since it is provided as an input to the problem.

The *benefit* of a view $v \in V$ for a q query is loosely defined as the change in cost of q evaluated with and without view v present in the multistore design. Formally,

$benefit(q, v) = cost(q, \mathcal{M} \cup v) - cost(q, \mathcal{M})$, where $\mathcal{M} \cup v$ means that v is added to both stores in $\mathcal{M}$.

For a query q, a pair of views (a, b) may interact with one another with respect to their benefit for q. The interaction occurs when the benefit of a for q changes when b is present. We employ the *degree of interaction* (*doi*) as defined in [85], which is a measure of the magnitude of the benefit change. Furthermore, the type of interaction for a view a with respect to b may be positive or negative. A positive interaction occurs when the benefit of a increases when b is present. Specifically, a and b are said to interact positively when the benefit of using both is higher than the sum of their individual benefits. In this case, we want to pack both a and b in the knapsack. A negative interaction occurs when the benefit of a decreases when b is present.

As an example, suppose that either a or b can be used to answer q. Hence both are individually beneficial for q, but suppose that a offers slightly greater benefit. When both views are present, the optimizer will always prefer a and thus never use b, in which case the benefit of b becomes zero. In this case packing both a and b in the knapsack results in an inefficient use of the view storage budget. We will utilize both the *doi* and the type of interaction between views when formulating our solution later.

Given our definitions of multistore design, the constraints, cost metric, and workload, we can define the multistore design problem.

Multistore Design Problem. Given an observed query stream, a multistore design $\mathcal{M} = \langle V_h, V_d \rangle$, and a set of design constraints B_h, B_d, B_t, compute a new multistore design $\mathcal{M}^{new} = \langle V_h^{new}, V_d^{new} \rangle$, where $V_h^{new}, V_d^{new} \subseteq V_h \cup V_d$, that satisfies the

V_h, V_d	Design of HV and DW before reorg
V_{cands}	Candidate views for M-KNAPSACK packing
V_h^{new}, V_d^{new}	Design of HV and DW after reorg
V_d^-	Views evicted from DW ($= V_d - V_d^{new}$)
B_h, B_d, B_t	View storage and transfer budgets
B_t^{rem}	Transfer budget remaining after DW design
$sz(v), bn(v)$	Size and benefit of view $v \in V_{cands}$

Table 5.1: MISO Tuner variables.

constraints and minimizes future workload cost.

5.2.2 MISO Tuner Algorithm

Because we desire a practical solution to the multistore design problem, we adopt a heuristic approach in the way we handle view interactions and solve the physical design of both stores. In this section we motivate and develop our heuristics and then later in the experimental section we show that our approach results in significant benefits compared to simpler tuning approaches.

The workload we address is online in nature, hence reorganization is done periodically in order to tune the design to reflect the recent workload. Our approach is to obtain a good multistore design by periodically solving a static optimization problem where the workload is given. At each reorganization phase, the tuner computes a new multistore design $\mathcal{M}^{new} = \langle V_h^{new}, V_d^{new} \rangle$ that minimizes total workload cost, as computed by our cost metric $TotalCost(W, \mathcal{M}^{new})$. Note that minimizing the total workload cost here is equivalent to maximizing the benefit of $\mathcal{M}^{new}$ for the representative workload W. The MISO Tuner algorithm is given in Algorithm 6, and its variables are defined in Table 5.1. We next give a high-level overview of the algorithm's steps

and then provide the details in the following sections.

Algorithm 6 MISO Tuner algorithm

1: **function** MISO_TUNE($\langle V_h, V_d \rangle$, W, B_h, B_d, B_t)
2: $V \leftarrow V_h \cup V_d$
3: $P \leftarrow$ COMPUTEINTERACTINGSETS(V)
4: $V_{cands} \leftarrow$ SPARSIFYSETS(P)
5: $V_d^{new} \leftarrow$ M-KNAPSACK(V_{cands}, B_d, B_t)
6: $B_t^{rem} \leftarrow B_t - \sum_{v \in V_h \cap V_d^{new}} sz(v)$
7: $V_h^{new} \leftarrow$ M-KNAPSACK($V_{cands} - V_d^{new}, B_h, B_t^{rem}$)
8: $\mathcal{M}^{new} \leftarrow \langle V_h^{new}, V_d^{new} \rangle$
9: **return** $\mathcal{M}^{new}$
10: **end function**

The first two steps given in lines 3 and 4 handle interactions between views as a pre-processing step before computing the new designs. The tuner algorithm begins by grouping all views in the current designs V_h and V_d into interacting sets (line 3). The goal of this step is to identify views that have strong positive or negative interactions with other views in V. At the end of this step, there are multiple sets of views, where views within a set strongly interact with each other, and views belonging to different sets do not. We sparsify each set by retaining some of the views and discarding the others (line 4). To sparsify a set, we consider if the nature of the interacting views within a set is positive or negative. If the nature of the interacting views is strongly positive, then as a heuristic, those views should always be considered together since they provide additional benefit when they are all present. Among those views, we treat them all as a *single* candidate. If the nature of the interacting views is strongly negative, those views should not be considered together for $\mathcal{M}^{new}$ since there is little additional benefit when all of them are present. Among those views, we choose a representative view as a

candidate and discard the rest. We describe these steps in detail in Section 5.2.3.

After the pre-processing step is complete, the resulting set of candidate views V_{cands} contains views that may be considered independently when computing the new multistore design, which is done by solving two multidimensional knapsack problems in sequence (lines 5 and 7). The dimensions of each knapsack are the storage budget and the transfer budget constraints. First on line 5, we solve an instance of a knapsack for DW, using view storage budget B_d and view transfer budget B_t. The output of this step is the new DW design, V_d^{new}. Then on line 7, we solve an instance of a knapsack for HV, using view storage budget B_h and any view transfer budget remaining (B_t^{rem}) after solving the DW design. The output of this step is the new HV design, V_h^{new}. The reason we solve the DW design first is because it can offer superior execution performance when the right views are present. With a good DW design, query processing can migrate to DW sooner thus taking advantage of its query processing power. For this reason, we focus on DW design as the primary goal and solve this in the first phase. After the DW design is chosen, the HV design is solved in the second phase. In this two-phase approach, the design of HV and DW can be viewed as complimentary, yet formulated to give DW greater importance than HV as a heuristic. We describe the two knapsack packings in detail in Section 5.2.4.

5.2.3 Handling View Interactions

Here we describe our heuristic solution to consider view interactions since they can affect knapsack packing. For example if a view v_a is already packed, and a view v_b

is added, then if an interaction exists between v_a and v_b, the benefit of v_a will change when v_b is added. Our heuristic approach only considers the strongest interactions among views in V, producing a set of candidate views whereby we can treat each view's benefit as independent when solving the knapsack problems. This is because a dynamic programming formulation requires that the benefit of items already present in the knapsack does not change when a new item is added. We use a two-step approach to produce the candidate views V_{cands} that will be used during knapsack packing, described below.

Before computing the interacting sets, we first compute the expected benefit of each view by utilizing the predicted future benefit function from [83]. The benefit function divides W into a series of non-overlapping epochs, each a fraction of the total length of W. This represents the recent query history. In this method, the predicted future benefit of each view is computed by applying a decay on the view's benefit per epoch — for each $q \in W$, the benefit of a view v for query q is weighted less as q appears farther in the past. The outcome of the computation is a smoothed averaging of v's benefit over multiple windows of the past. In this way, the benefit computation captures a longer workload history but prefers the recent history as more representative of the benefit for the immediate future workload.

Compute Interacting Sets. To find the interacting sets, we use the method from [84] to produce a *stable partition* $P = \{P_1, P_2, \ldots, P_m\}$ of the candidate views V, meaning views within a part P_i do not interact with views in another part P_j. In this method,

the magnitude of the interaction (i.e., change in benefit) between views within each part P_i is captured by *doi* [85]. When computing *doi*, we slightly modify the computation to retain the sign of the benefit change, where the sign indicates if the interaction is positive or negative. For a pair of views that have both positive and negative interactions, the magnitude is the sum of the interactions, and the sign of the sum indicates if it is a net positive or negative interaction. The partitioning procedure preserves only the most significant view interactions, those with magnitude above some chosen threshold. The threshold has the effect of ignoring weak interactions. The value of the threshold is system and workload dependent, but should be sufficiently large enough to result in parts with a small number (e.g., 4 in our case) of views thus only retaining the strongest interactions.

Since the parts are non-overlapping (i.e., views do not interact across parts), each part may be considered independently when packing M-KNAPSACK. This is because the benefit of a view in part P_i is not affected by the benefit of any view in part P_j, where $P_i \neq P_j$. At this point however, some parts may have a cardinality greater than one, so we next describe a method to choose a representative view among views within the same part.

Sparsify Sets. To sparsify the parts, we first take account of positively interacting views, then negatively interacting views. We exploit positive interactions by applying a heuristic to ensure that views that have a large positive interaction are packed together in the knapsack. Within each part, view pairs with positive interactions are "merged"

into a single knapsack item, since they offer significant additional benefit when used together. The new item has a weight equal to the sum of the size of both views, and the benefit is the total benefit when both views are used together. This merge procedure is applied recursively to all positively interacting pairs within each part, in decreasing order of edge weight. Merging continues until there are no more positively interacting pairs of views within the part.

After applying the merge procedure, all parts with cardinality greater than 1 contain only strongly negative interacting views. Our next heuristic is to not pack these views together in the knapsack, but select one of them as a representative of the part. This is because packing these views together is an inefficient use of knapsack capacity. To choose the representative, we order items in the part by decreasing benefit per unit of weight and only retain the top item as a representative of the part. This greedy heuristic is commonly used for knapsack approaches to physical design [25, 96] and we use it here for choosing among those views with strong negative interactions within a part. The procedure is applied to each part until all parts have a single representative view. After these steps, the resultant partition P contains only singleton parts. These parts form V_{cands} and are treated as independent by M-KNAPSACK.

5.2.4 MISO Knapsack Packing

During each reorganization window, the MISO Tuner solves two instances of a 0-1 multidimensional knapsack problem (M-KNAPSACK henceforward); the first instance for DW and the second instance for HV. Each instance is solved using a dynamic pro-

gramming formulation. At the beginning of each reorganization window, note that V_h includes all views added to the HV design by the MISO tuner during the previous reorganization window as well as any new opportunistic views in HV created since the last reorganization. During reorganization, the MISO tuner computes the new designs V_h^{new} and V_d^{new} for HV and DW respectively. Since the DW has better query performance, as a first heuristic MISO solves the DW instance first resulting in the best views being added to the DW design. Furthermore, we ensure $V_h \cap V_d = \emptyset$ to prevent duplicating views across the stores. Although this is also a heuristic, our rationale is it potentially results in a more "diverse" set of materialized views and hence better utilization of the limited storage budgets in preparation for an unknown future workload. If desired, this property could be relaxed by including all views in V_{cands} when packing both HV and DW.

As an additional heuristic we do not limit the fraction of B_t that can be consumed when solving the DW design in the first phase. Any remaining transfer budget B_t^{rem} can then be used to transfer views evicted from DW to HV. This means that all of the transfer budget could potentially be consumed during the first phase when transferring views to DW. Alternatively, we could reserve a fraction of B_t for transfers in each direction, although this too would be a heuristic.

5.2.4.1 Packing DW M-Knapsack

In this phase, the target design is V_d^{new}, and the M-KNAPSACK dimensions are B_d and B_t. There are two cases to consider when packing DW. Views in HV (V_h) will

consume the transfer budget B_t (Case 1), while views in DW (V_d) will not (Case 2). The candidate views are $V_{cands} = V_h \cup V_d$. The variable k denotes the k^{th} element in V_{cands} (i.e.,view v_k), the order of elements is irrelevant. The recurrence relation C is given by the following two cases.

Case 1: $v_k \in V_h$ applies when view v_k is present in HV.

$$C(k, B_d, B_t) = \begin{cases} C(k-1, B_d, B_t) & ; sz(v_k) > B_t \\ \max \begin{cases} C(k-1, B_d, B_t), \\ C(k-1, B_d - sz(v_k), B_t - sz(v_k)) + bn(v_k) \end{cases} & ; sz(v_k) \leqslant B_t, sz(v_k) \leqslant B_d \end{cases}$$

In this case, either element k does not fit in B_t (since $sz(v_k) > B_t$), and if so, skip element k and continue. Otherwise, k does fit in B_t and B_d (since $sz(v_k) \leqslant B_t$, and $sz(v_k) \leqslant B_d$), and if so take the max value of either (a) skip element k and continue, or (b) add element k to M-KNAPSACK, subtract its size from B_t and B_d, accumulate its benefit ($bn(v_k)$), and continue.

Case 2: $v_k \notin V_h$ applies when view v_k is not in HV.

$$C(k, B_d, B_t) = \begin{cases} C(k-1, B_d, B_t) & ; sz(v_k) > B_d \\ \max \begin{cases} C(k-1, B_d, B_t), \\ C(k-1, B_d - sz(v_k), B_t) + bn(v_k) \end{cases} & ; sz(v_k) \leqslant B_d \end{cases}$$

In this case, either element k does not fit in B_d (since $sz(v_k) > B_d$), and if so, skip element k and continue. Otherwise, k does fit in B_d (since $sz(v_k) \leqslant B_d$), and if so take the max value of either (a) skip element k and continue, or (b) add element k to M-KNAPSACK, subtract its size from B_d, accumulate its benefit ($bn(v_k)$), and continue. At the end of this first phase the design of DW, V_d^{new}, is complete.

5.2.4.2 Packing HV M-Knapsack

In this phase, the target design is V_h^{new}, and the M-KNAPSACK dimensions are B_d and B_t^{rem}, and V_d^- represents the views evicted from the DW which are now available for transfer back to HV. The solution is symmetric to Phase 1, with modified inputs. The value B_t is initialized to B_t^{rem}. The candidate views are those remaining in V_h and those evicted from DW (V_d^-); therefore $V_{cands} = (V_h \cup V_d) - V_d^{new}$. We similarly have 2 cases, defined in Phase 2 as:

- Case 1: $v_k \in V_d^-$ applies when view v_k was evicted from DW.
- Case 2: $v_k \notin V_d^-$ applies when view v_k was not evicted from DW.

As before, these cases indicate whether a view needs to be moved or not. The recurrence relations from Phase 1 can now be used with these modified cases.

The complexity of our knapsack approach is $O(|V| \cdot \frac{B_t}{d} \cdot \frac{B_d}{d} + |V| \cdot \frac{B_t}{d} \cdot \frac{B_h}{d})$, where d represents the discretization factor (e.g., 1 GB). By discretizing the storage and transfer budgets, this approach can be made efficient in practice and suitable for our scenario that requires periodically recomputing new designs.

5.3 Experimental Study

In this section we present an experimental evaluation of the effectiveness of MISO for speeding up multistore queries. First we evaluate MISO in a multistore system in the absence of any other workload on DW in order to better understand the performance of our tuning techniques. Later we perform a more realistic evaluation of MISO by using a DW with limited spare capacity by running a background workload on DW. We then measure the impact of the multistore queries on the DW background workload and vice-versa. In Section 5.3.1 we describe the experimental methodology. In Section 5.3.2 we compare MS-MISO to several other system variants to highlight its benefit for multistore query processing. In Section 5.3.3 we compare the behavior MS-MISO to several possible multistore tuning algorithms. In Section 5.3.4, to evaluate MS-MISO in a realistic scenario we model a DW with an existing business workload and show the impact of executing multistore queries on the DW. Finally, in Section 5.3.5 we briefly compare aspects of our approach that use HV (i.e., Hive, which is Hadoop-based) with two alternatives that are not Hadoop-based.

5.3.1 Methodology

Data sets and workloads. Our evaluation utilizes three data sets: a 1 TB Twitter data stream of user "tweets", a 1 TB Foursquare data stream of user "check-ins", and a 12 GB Landmarks data set represent static data containing geographical information about locations of popular interest. The identity of the user is common across Twitter and Foursquare logs, while the checkin location is common across the Foursquare and Landmarks logs. The data sets are stored in their native JSON format as logs in HDFS. Log rows that contain dirty, mal-formed, or incomplete data are discarded (i.e., ignored) by the queries.

Our workload consists of 32 complex analytical queries given in [61] that access social media data and static historical data, for restaurant marketing scenarios. The queries model eight data analysts, each posing and iteratively refining a query multiple times during their data exploration. Each analyst (referred to as A_i for Analyst i) evolves a query through four versions denoted as $A_iv_1, A_iv_2, A_iv_3, A_iv_4$. An evolved query version A_iv_2 represents a mutation of the previous version A_iv_1, as described in [61], thus there is some overlap between queries. The queries contain relational operators as well as UDFs, and the queries are written in HiveQL with UDFs included via the HiveCLI.

Stores. We utilize two distinct data stores: a Hive store (HV) and a parallel data warehouse (DW). Each store is a cluster of nodes, and the clusters are independent. The clusters are connected via 1GbE network and reside on adjacent racks. Each node

has the same hardware: 2 Xeon 2.4GHz CPUs (8 cores), 16 GB of RAM, and exclusive access to its own disk (2TB SATA 2012 model). Additionally, the head nodes each have another directly attached 2 TB disk for data staging (used for data transfers and loading between the stores). HV runs Hive version 0.7.1 and Hadoop version 0.20.2, while DW runs the latest version of a widely-used, mature commercial parallel database (row-store) with horizontal data partitioning across all nodes. The HV cluster has 15 machines and the DW cluster has 9 machines. The HV cluster is 1.5× the size of the DW cluster since HV is less expensive than DW and thus in a realistic setting would typically be larger.

Experimental parameters. For some experiments we vary the view storage budgets B_h, B_d and the view transfer budget B_t. Storage budgets B_h and B_d are varied as a fraction of the base data size in each store, i.e., with 10 GB of base data a storage budget of 1.0× equals 10 GB. Because the log data is stored in HV, we consider its base data size to be 2TB corresponding to the size of the logs. Because there is no log data stored in DW, we consider its "base data" size to be 200 GB corresponding to the relevant portion of the log data accessed by the queries. For example, $B_h = 2\times$ is 4 TB, whereas $B_d = 2\times$ is 400 GB. B_t is expressed in GB representing the size of the data transferred between the stores during each reorganization phase.

System variants. We evaluate the workload on several system variants described below. Note that all base data (logs) is stored in HV.

- HV-ONLY is the Hive store. Queries complete their entire execution in the 15 node HV store only.

- DW-ONLY is the data warehouse store. Queries complete their entire execution in the 9 node DW store only. Prior to query execution, we ETL the subset of the log data accessed by the queries using HV as an ETL engine. Note that using Hive or Pig for ETL is becoming a common industry practice (e.g., Netflix, Yahoo, Intel)[1]. Any UDFs that do not execute in DW are applied as part of the ETL process in HV, before loading into DW.

- HV-OP is an HV store that retains opportunistic views and rewrites queries using these views, with the method from [63]. When views exceed the view storage budget, they are evicted using an LRU replacement policy. Queries complete their entire execution in the 15 node HV store only.

- MS-BASIC is a basic multistore system, that does not make use of opportunistic views. Queries may execute in part on both the 15 node HV store and the 9 node DW store as determined by the optimizer.

- MS-MISO is our multistore system that uses the MISO Tuner, which determines the placement of views in each store. Queries may execute in part on both the 15 node HV store and 9 node DW store, as determined by the optimizer which takes into account the current placement of views in each store. Reorganization phases

[1] `http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html`, `http://developer.yahoo.com/blogs/hadoop/pig-hive-yahoo-464.html`, `http://hadoop.intel.com/pdfs/ETL-Using-Hadoop-on-Intel-Arch.pdf`.

(R) occur every 1/10 of the full workload, which in this case is after every 3 queries. When computing the expected future benefit (Section 5.2.3), we use a query history length of 6 and epoch length of 3 queries. During reorganizations, no queries are executed and MISO completes all view movements, after which time query execution begins again.

Metrics. Our primary metric is TTI, representing the total elapsed time from workload submission to workload completion. TTI is defined as the cumulative time of loading data, transferring data during query execution, tuning the systems, and executing the queries. Wherever relevant, we break down TTI into the following components.

1. HV-EXE is the cumulative time spent executing queries in HV.
2. DW-EXE is the cumulative time spent executing queries in DW.
3. TRANSFER is the cumulative time for transferring (and loading) data between the stores during multistore query execution.
4. TUNE is the cumulative time spent computing new designs by a tuning algorithm, moving views between stores, and creating any recommended indexes for the views in DW.
5. ETL is the time spent loading the relevant data into DW using HV as an ETL engine before executing the workload. ETL time is only applicable to the DW-ONLY system variant.

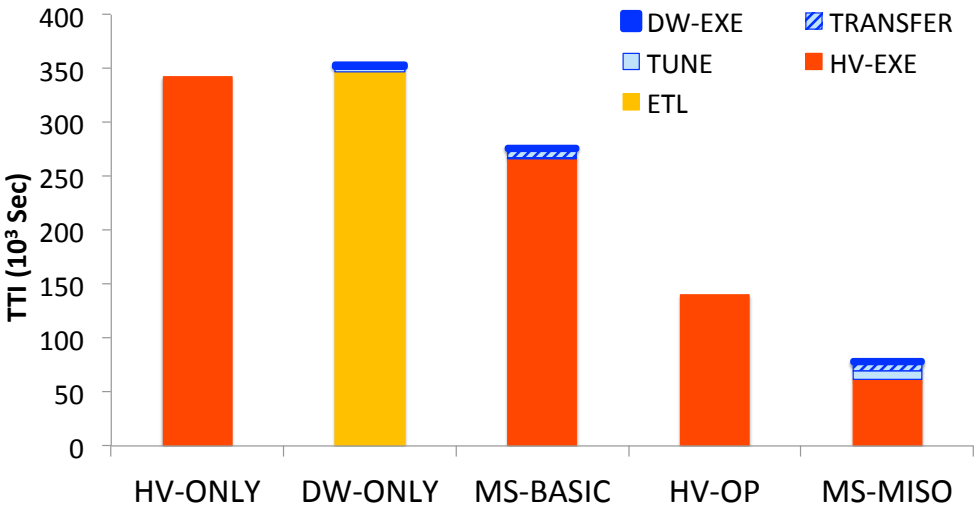

Figure 5.6: Performance results for 5 system variants.

5.3.2 Main Results

We first summarize the main results showing how MS-MISO outperforms other system variants. Figure 5.6 compares MS-MISO with the 4 other system variants – HV-ONLY, DW-ONLY, MS-BASIC, and HV-OP. In this experiment, HV-OP has a view storage budget of 2× on HV, and views are retained in HV within the storage budget using a simple LRU caching policy. MS-MISO has storage budgets B_h and B_d of 2× and B_t of 10GB. The systems variants are shown along the x-axis, and the y-axis reports TTI in seconds. For MS-MISO, TTI also includes time to compute new designs as well as the time spent moving views during reorganization phases. HV-ONLY represents the native system for our Hive queries, thus we use it as the baseline for comparing the other system variants.

Figure 5.6 shows MS-MISO has the best performance in terms of TTI, and

DW-ONLY has the worst performance. DW-ONLY is actually 3% slower than HV-ONLY. This is because the vast majority of TTI for DW-ONLY is spent in the ETL phase, while the query execution time (DW-EXE) only constitutes a small fraction of the TTI. Although ETL is very expensive, it is a one-time process in our system and the cost is amortized by the benefit DW provides for all the queries in the workload. Clearly DW-ONLY would provide greater benefit for a longer workload. However, making this high up-front time investment to ETL all of the data is difficult to justify due to the exploratory nature of the queries. MS-BASIC offers limited performance improvement over HV-ONLY, only 19%, or a 1.2× speedup. This is due to the high cost of repeatedly transferring data between the stores for each query. Since the data transferred and loaded is not retained after a query completes, subsequent queries do not benefit from this effort. HV-OP shows a 59% improvement over HV-ONLY, representing a 2.4× speedup. This improvement is attributable to opportunistic views retained during query processing in HV. MS-MISO has a 77% improvement over HV-ONLY, representing a speedup of 4.3×. MS-MISO also results in a 68% improvement over MS-BASIC (3.1×), and a 44% improvement over HV-OP (1.8×). These results show that MS-MISO is able to utilize both the opportunistic views and DW effectively to speed up query processing.

The improvement of MS-MISO over HV-OP and MS-BASIC can be attributed to the following two reasons. First, it is able to load the right views into the right store during its reorganization phases. The reorganization phases allow MS-MISO to tune both stores periodically as the workload changes dynamically. This is especially important for exploratory queries on big data, since the queries and the data they access

are not known up-front. Second, by performing lightweight periodic reorganization, MS-MISO is able to benefit from DW's superior query processing power. Furthermore, these two factors are relevant even with small storage budgets. For example, when we repeat this experiment using B_h and B_d equal to 0.125×, MS-MISO still results in 59% improvement over HV-ONLY (2.4× speedup). Next we provide a breakdown of the performance of MS-MISO to offer insights into how the combination of these two factors results in better performance.

5.3.2.1 Breakdown of TTI

To further breakdown the previous results from Figure 5.6, Figures 5.7 and 5.8 report the cumulative distribution plots (CDF) for TTI and query execution time (respectively) for the five system variants. Figure 5.7 shows TTI along the y-axis, and the number of queries completed on the x-axis. A point (x, y) in this figure indicates that x queries had a total TTI of no more than y seconds. DW-ONLY has the worst TTI due to its significant delay before any query can begin execution – the first query can only start after the ETL completes at 348,000 seconds. In contrast, HV-ONLY, HV-OP, MS-BASIC, and MS-MISO allow users to start querying right away, although with varying performance. MS-BASIC and HV-ONLY have similar performance, again indicating that MS-BASIC is not able to make effective use of DW because it only considers a single query at a time and does not perform any tuning. HV-OP has the second best TTI due to its effective use of opportunistic views. MS-MISO has the fastest TTI among all system variants while still offering the benefits of immediate querying.

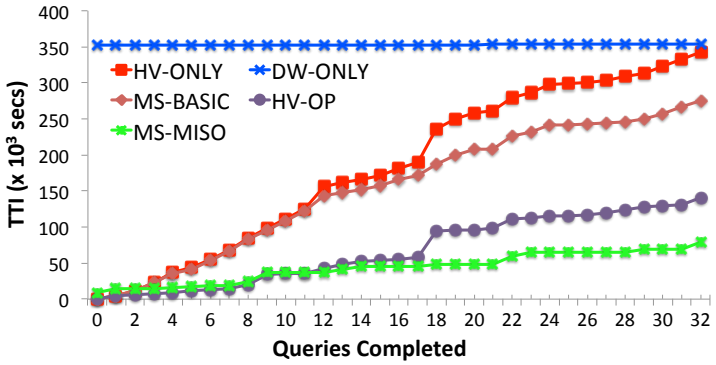

Figure 5.7: CDF plot showing the progressive TTI (y-axis) as each of the 32 queries in the workload is completed (x-axis).

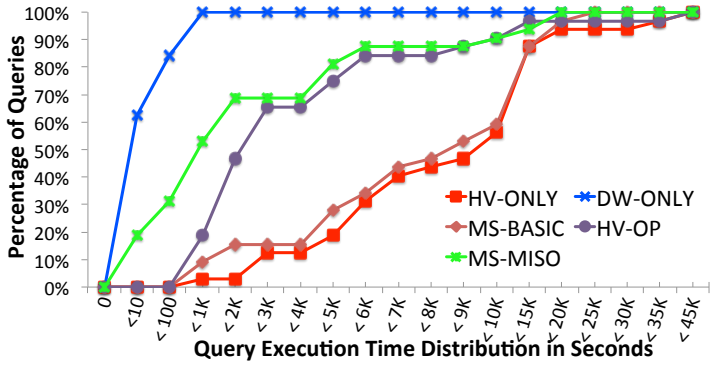

Figure 5.8: CDF plot showing the percentage of queries (y-axis) below a given execution time (x-axis).

Figure 5.8 shows the percent of queries with execution time less than 10, 100, 1,000, 2,000 and so on until 45,000 seconds. To gain insight on how the queries perform, we report query execution time only, which does not include any ETL time or tuning time. Clearly DW-ONLY has the best query performance (top-most curve) with 84% of queries completing in less than 100 seconds and all the queries finishing within 1,000 seconds. Furthermore, the DW-ONLY curve shows that nearly 65% of queries complete within 10 seconds, and 90% within 100 seconds. In contrast, HV-ONLY performs worst

(bottom-most curve) with less than 3% of queries completing within 1,000 seconds. The systems near the bottom of the graph, HV-ONLY, MS-BASIC, and HV-OP, have no queries that execute in less than 100 seconds. The systems near the top of the graph, DW-ONLY and MS-MISO, complete at least 30% of their queries in less than 100 seconds. While it is not surprising that DW-ONLY outperforms MS-MISO in terms of its *post-ETL* query performance, DW-ONLY is no longer competitive when ETL time is included. This shows that the lightweight adaptive tuning approach of MS-MISO is able to utilize the query processing power of DW without incurring the high up front cost of the ETL process.

5.3.2.2 Breakdown of Store Utilization

Next we highlight why MS-MISO performs better than MS-BASIC by examining the amount of time queries spend in each store. For each system in Figure 5.9, the y-axis indicates the fraction of time each query spends in HV, DW, or transferring data. For each system we rank the 32 queries by their DW utilization percent, and assign query rank 1 to the query that has the highest DW utilization, and rank 2 to the query with the next highest DW utilization; the x-axis indicates the query rank. Because the utilization trends stabilize after 15–20 queries, we truncate the x-axis in each figure.

Figure 5.9 shows this breakdown for three system variants – MS-BASIC in (a), MS-MISO with 0.125× view storage budget in (b), and MS-MISO with 2× view storage budget in (c). For MS-BASIC, only 2 queries spend the majority of their execution time in DW. In contrast, 9 of the queries in MS-MISO with the 0.125× storage budget spend

the majority of their total execution time in DW. This increases to 14 queries when the MS-MISO storage budget is increased to 2×.

Another way of quantifying DW utilization for each system is to consider the total execution time spent in HV versus DW for the first fastest queries in the workload, i.e., query ranks 1–16 in each figure. For queries ranked beyond 16, the HV-EXE component dominates most of a query's execution time, which does not illuminate DW utilization at all. With MS-BASIC in Figure 5.9(a), for every second spent in DW the queries spend 55 seconds in HV. With MS-MISO in Figure 5.9(b), for every 1 second spent in DW the queries only spend 1.6 seconds in HV. With MS-MISO in Figure 5.9(c), for every 1 second spent in DW the queries only spend 0.12 seconds in HV. This breakdown shows that MS-MISO is able to utilize DW more effectively than MS-BASIC, and increasing the storage budget for MS-MISO further increases DW utilization.

Finally, a different way to to examine store utilization is by reporting the ratio of plan operators executed by each store for each of the 32 queries. For MS-BASIC the best splits for all 32 queries were typically about 2/3 of the operators in HV and 1/3 in DW. For MS-MISO, an overview of the 32 splits is roughly as follows. For the fastest 9 queries, the split was 0/3 HV and 3/3 DW. For the next-fastest 20 queries, the average split was 1/3 HV and 2/3 DW. For the remaining 3 queries the average split was around 2/3 HV and 1/3 DW. Note that splits do not correlate exactly with plan execution times in Figure 5.9(b) due to the asymmetric performance of the stores.

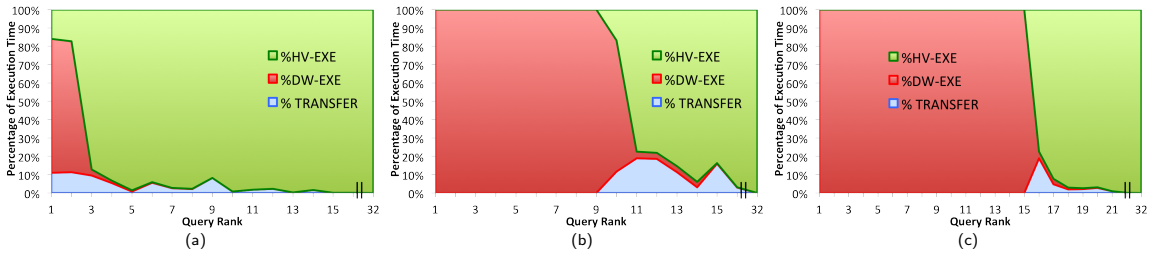

Figure 5.9: Breakdown of execution time into DW-EXE, HV-EXE and Transfer time components for queries from Figure 5.7(b); (a) MS-BASIC, (b) MS-MISO for 0.125× storage budget, and (c) MS-MISO for 2× storage budget.

5.3.3 Tuning Algorithm Comparisons

In this section we evaluate MS-MISO against four other possible tuning techniques for multistore physical design.

- MS-BASIC: Performs no tuning (i.e., does not use views).
- MS-OFF: Performs offline tuning, where it is given the entire workload up-front and tunes the stores one time by choosing the most useful set of views for the workload. This approach represents what is possible with current design tools and is provided only as a point of reference to show the performance trends.
- MS-LRU: Performs "passive" tuning by retaining working sets transferred between the stores during query execution as "views". LRU and its many variants are access-based approaches and provide a simple way of deciding which views to retain within a limited storage budget.
- MS-MISO: Performs online tuning using our MISO Tuner.
- MS-ORA: Performs online tuning using our MISO Tuner also, but the *actual* future

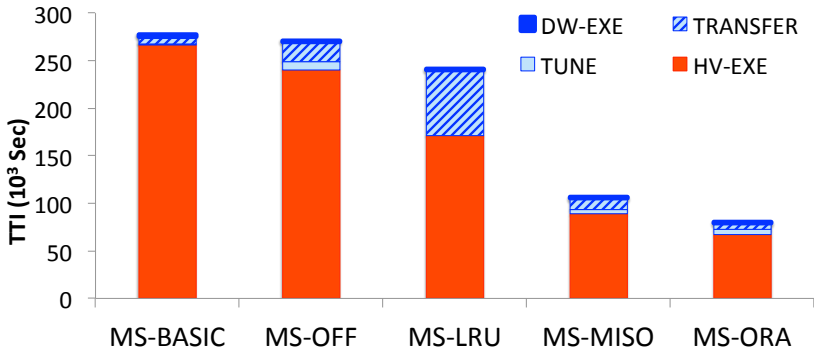

Figure 5.10: *TTI* Comparison of multistore tuning techniques.

workload is provided rather than predicted. This technique ("oracle") is just provided as a point of reference as the best performance possible for the MISO Tuner.

Tuning parameters for MS-OFF, MS-LRU, MS-MISO, and MS-ORA are set to $B_h = 0.125\times$, $B_d = 0.125\times$, and $B_t = 10$ GB since these budgets represent a more constrained environment.

Figure 5.10 compares the performance of the tuning techniques described above. Among all techniques, MS-BASIC performs the worst. This shows that multistore query processing without tuning does not perform as well as any of the other tuning techniques. Among the techniques that perform tuning, MS-OFF has the worst performance because the small storage budgets are insufficient to store all beneficial views for the entire workload. MS-MISO provides a 60% improvement over MS-OFF since it is able to adapt the physical design to the changing workload. Furthermore, MS-MISO results in a 56% improvement over MS-LRU because MS-LRU does not explicitly consider view benefits or interactions. This is because MS-LRU is an access-based approach whereas MS-MISO is a cost-benefit based approach that considers view benefits

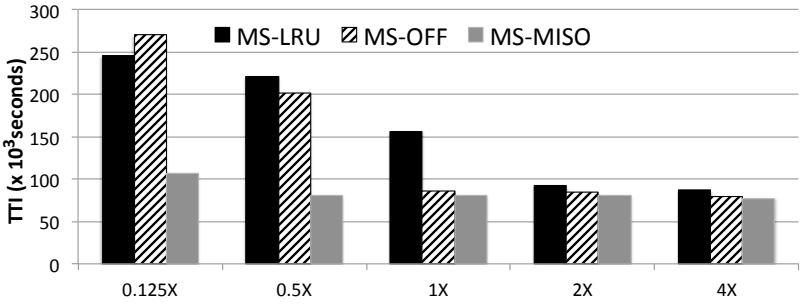

Figure 5.11: Varying the view storage budgets B_h and B_d, while transfer budget B_t is held constant at 10GB.

and interactions. Notice there is a large transfer time for MS-LRU, indicating that this method does not do a good job of retaining beneficial views. Finally, MS-MISO is 32% worse than MS-ORA, since the actual future workload is unknown to MS-MISO.

5.3.3.1 Varying the Tuning Parameters

Next, we analyze the performance of MS-MISO, MS-OFF, and MS-LRU under varying tuning constraints. Figure 5.11 shows the performance of MS-OFF, MS-LRU, and MS-MISO as we vary the storage budget parameters B_h and B_d from 0.125× to 4.0×. B_t was kept fixed at 10 GB. The y-axis reports TTI, and the x-axis shows the storage budgets on each store.

MS-MISO performs the best for all storage budgets, including the larger storage budgets. Among the smaller budgets of 0.125× and 0.5×, both MS-OFF and MS-LRU perform significantly worse than MS-MISO. However, the performance of MS-OFF and MS-LRU improves with increasing storage budgets. For example, at 1× budget, MS-MISO is 50% better than MS-LRU but only 7% better than MS-OFF, while with larger budgets the performance of all three methods begins to converge. Note that MS-

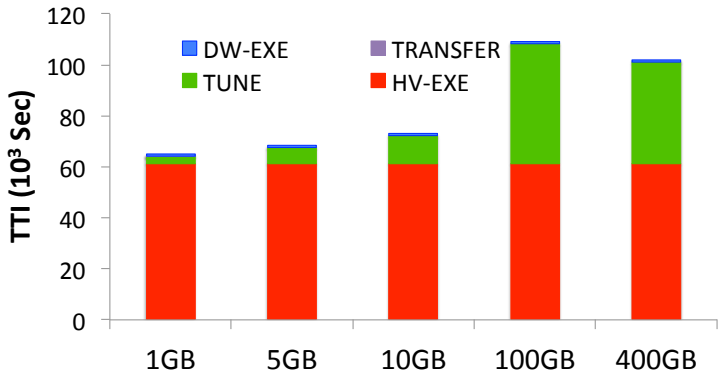

Figure 5.12: TTI for MS-MISO while varying the B_t parameter

OFF is not actually feasible in practice for exploratory queries since the workload is changing dynamically but offline techniques need to be provided with the entire workload up-front in order to create its design — hence MS-OFF is shown here only as an interesting point of reference for comparison. The tuning techniques begin to have similar performance with larger storage budgets (2–4×) since there is plenty of available storage space to retain many views in support of the recent workload.

We now study the effect of the view transfer budget B_t on the MS-MISO system. In Figure 5.12, the parameters B_h and B_d are held constant at 2×, and transfer budget B_t is varied from 1 GB, 5 GB, 10 GB, 100 GB, 400 GB. MS-MISO has good performance when the transfer budget is small (less than 10 GB), while the performance is significantly affected for larger values of the transfer budget because the tuning time becomes very large. Although the storage budgets B_h and B_d are 2× here, we observed this trend for all B_h and B_d storage budgets. As Figure 5.12 shows, choosing large transfer budgets has the effect of "over tuning" the system. This indicates that performing small changes over time to match workload shifts is sufficient. Hence, the

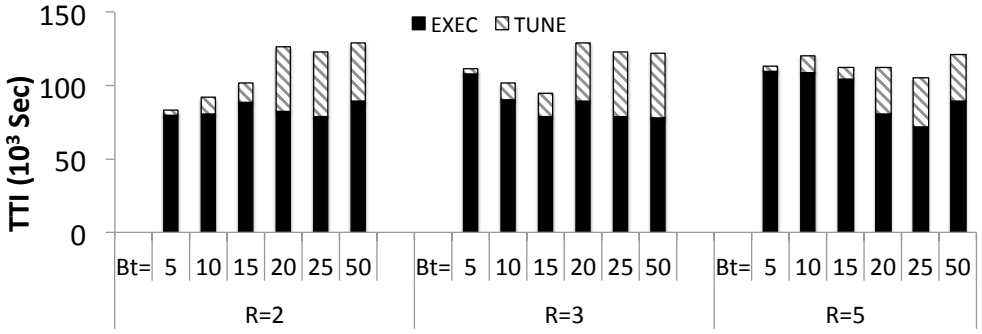

Figure 5.13: TTI for MS-MISO while varying the B_t parameter and reorganization frequency (R).

reconfiguration windows can be relatively short, and still provide the majority of benefit for MS-MISO.

Next we study the effect of varying the view transfer budget B_t on the MS-MISO system. In Figure 5.13, parameters B_h and B_d are held constant at 0.125× while the transfer budget B_t is varied from 5 GB to 50 GB. While each bar represents TTI, the solid portion is the cumulative query execution time, while the striped portion is the cumulative tuning time. To further illuminate the effect of varying B_t, we also consider different reorganization frequencies (R): every 2 queries ($R = 2$), every 3 queries ($R = 3$), and every 5 queries ($R = 5$). While we vary R and B_t here, we do not vary the query history length. Note that all other experiments throughout Section 5.3 use $R = 3$ and $B_t = 10$; as a point of reference $B_t = 10$ and $R = 3$ in Figure 5.13 corresponds to MS-MISO in Figure 5.10 and MS-MISO with 0.125× in Figure 5.11. Figure 5.13 shows that smaller values for both R and B_t generally result in better TTI. While these results are workload specific, they show that MS-MISO is sensitive to the B_t value.

Figure 5.13 shows the best TTI occurs for $R = 2$ when $B_t = 5$, for $R = 3$

when $B_t = 15$, and for $R = 5$ when $B_t = 25$. This indicates that when reorganizing less frequently, larger B_t values are beneficial. Intuitively this makes sense as more tuning effort is required if tuning is done less often. Additionally, the best TTI for $R = 2$ is 19% better than the best TTI for $R = 5$. These trends generally indicate that MS-MISO performs best when keeping B_t small and reorganizing fairly frequently. However, as we show in the next section there is a tradeoff to consider for the reorganization frequency. Loading views into the DW is a resource intensive operation (IO and CPU), thus reorganizing too frequently can have a negative impact on ongoing DW operations (e.g., reporting queries). Although specific to this workload, this supports MISO's lightweight adaptive approach to tuning, particularly when queries exhibit some temporal locality such as a series of exploratory queries. Choosing a good value for the B_t parameter is workload and system dependent, later in the discussion in Section 5.4 we provide some guidance for DBAs based on our experiences.

5.3.4 Utilizing DW with Limited Spare Capacity

Now that we have shown the basic performance tradeoffs with MS-MISO, we evaluate MS-MISO in a realistic scenario with a DW that has limited spare capacity – that is, a DW that is executing a workload of reporting queries. As DW is a tightly-controlled resource, it is important to understand how our method affects the DW. To do this, we measure the impact of the multistore query workload on the DW reporting queries as well as the impact of the DW queries on the multistore query workload. These experiments are important because they model a realistic deployment scenario

for a multistore System where an organization has two classes of stores and may want to utilize their existing DW resources to speed up the big data queries in HV.

To model a running DW, we consider varying amounts of spare IO and CPU capacity by executing a reporting query workload in DW. We examine four cases: A DW with 20% spare IO capacity, 20% spare CPU capacity, 40% spare IO capacity, and 40% spare CPU capacity. The spare capacities are measured as the amount of unused resources as reported by Linux IOstat; for example, 60% CPU consumption indicates a 40% spare CPU capacity. We measure these values for each of the 9 machines in the DW cluster every 10 seconds, and report the average value. In this scenario, we execute a background DW workload on the DW cluster that consumes a fixed proportion of IO or CPU resources. Below, we present the results for one case: a DW with 40% spare IO capacity. The remaining three cases showed very similar trends and hence are omitted. However, Table 5.2 below summarizes all of the results.

The multistore query workload is then executed as a single stream of 32 queries. As the multistore query workload is executed, we measure its impact on the background DW queries by the amount of slowdown of the background queries, representing the impact that the multistore workload has on the reporting queries running in DW. Conversely, we also measure the impact of the DW queries on the multistore workload.

To create a DW environment with limited IO or CPU resources, we use a single TPC-DS query and run multiple parameterized versions of the query in parallel in order to consume a fixed amount of each resource. The queries are used to control the amount of spare IO or CPU capacity. To do this, we first load a 1TB scale TPC-DS

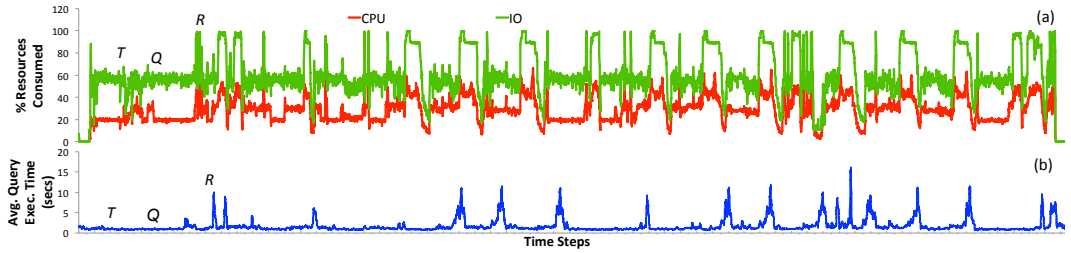

Figure 5.14: Impact of multistore workload on DW with 40% spare IO capacity showing (a) IO/CPU resources consumed and (b) average query execution time of the DW queries.

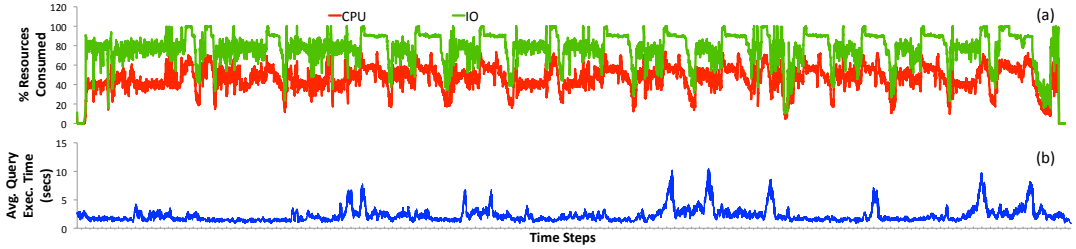

Figure 5.15: Impact of multistore workload on DW with 20% spare IO capacity showing (a) IO/CPU resources consumed and (b) average query execution time of the DW queries.

dataset into DW. We then selected 2 queries, query templates q3 and q83, since query q3 is IO intensive and query q83 is CPU intensive. To obtain 40% spare IO capacity in DW, we continuously execute one instance of q3 (which consumes 60% of the IO resources). To obtain 20% spare IO capacity, we continuously execute three instances of q3. Similarly, to obtain 40% spare CPU capacity in DW, we continuously execute two instances of q83 (which consumes 60% of the CPU resources). To obtain 20% spare CPU capacity, we continuously execute three instances of q83. All views transferred during multistore query execution are kept in a temporary table space, whereas views transferred by MS-MISO during reorganization are stored in permanent table space.

Figure 5.14(a) reports the CPU and IO load for the DW cluster with 40%

spare IO capacity. The x-axis corresponds to time steps during experiment, and the y-axis indicates CPU and IO resource consumption. Initially, the IO is stable at 60% and the CPU consumption is at about 20%, then the multistore queries begin executing. Multistore queries may spend a lot of their execution time in HV, during which time DW continues executing only its background workload.

The peaks in Figure 5.14(a) (e.g., marked R, T) correspond to periods of either view transfers during reorganization (R) by MS-MISO or working set transfers during query processing (T), while the flatter areas (e.g., marked Q) correspond to periods of multistore query execution. For both R and T cases, the data transfers put heavy demands on the IO resources, in some instances consuming 100% of the IO resources. In Figure 5.14(a), view transfers during (R) and (T) phases consume a lot of the available spare capacity of DW, while multistore query execution (Q) has little impact on the spare capacity. This results in brief instances of high resource impact, but long time periods with low impact. Next, we show how the resource consumption by the multistore queries impacts the execution of the running DW queries.

Figure 5.14(b) shows the average execution time of the DW background queries as Multistore queries are executed for the same experiment. The x-axis again reports the timesteps during the experiment and the y-axis reports the average execution time of q3 during the entire experiment. The average execution time of q3 when no Multistore queries are executing is 1.06 seconds. Over the course of the entire experiment, the average execution time of q3 increased to 1.09 seconds, representing an overall slowdown of 2.5% for q3. The peaks in this figure correspond similarly to working set

transfers during query execution (T), or view transfers during reorganization (R). This figure shows that these events briefly impacting the average execution time of q3, which increases to over 5 seconds several times. The regions Q again correspond to Multistore query execution in DW. This figure shows that the impact on the average execution time of DW background queries is very small in spite of brief periods of high impact.

Figure 5.15(a) reports the CPU and IO load for the DW cluster with 20% spare IO capacity, while Figure 5.15(b) shows the average execution time of the DW queries during the entire experiment. These figures show similar trends to the previous figures, with brief periods of high resource impact for the same reasons. For the 20% and 40% spare CPU capacity case, the figures also look similar and hence are not shown here, but we report the impact numbers for all CPU and IO spare capacities tested in Table 5.2.

DW Spare Capacity		Percent Slowdown of	
		DW Queries	Multistore Workload
IO	40%	1.1%	2.5%
	20%	1.7%	4.0%
CPU	40%	0.3%	4.2%
	20%	0.8%	5.0%

Table 5.2: Impact of multistore workload on DW queries and vice-versa.

In order to quantify the impact of an active DW versus an empty DW, Table 5.2 reports the impact of the multistore workload and the DW queries on each other. For the DW queries, we report the slowdown in their average execution time with versus without the multistore queries running. For the multistore queries, we report the slowdown in the TTI with and without the DW queries executing. The slowdown of the DW queries

due to the Multistore queries less than 2%. Similarly, the slowdown of the multistore workload is no more than 5% versus an empty DW. This shows that the impact of the workloads on each other is actually quite minimal, indicating MS-MISO is indeed beneficial when there is limited spare capacity in DW.

5.3.5 Comparison with non-Hadoop approaches

In our work, we assume a scenario where a business has both HV and DW, and can utilize the DW to speed up big data exploratory queries. The HV store is based upon Hadoop for data storage and query processing, but non-Hadoop solutions could be considered. Here we briefly evaluate two alternative approaches to speed up the big data queries.

First, one can consider other ETL tools rather than using HV as an ETL engine as we did in our experiments, and simply load all of the big data into DW before executing the big data queries. However, since we assume the base data is stored in an HDFS system, the data must be extracted from there and loaded into DW. We compare our approach with an ETL tool in Section 5.3.5.1. Second, one can consider a number of emerging systems that seek to improve big data processing, among the most popular of these is Cloudera's Impala data processing engine. Impala can directly process data stored in HDFS by first loading it into main memory and then using its own query execution engine that is not based on Hadoop. The Hadoop job execution engine is known to have significant overhead, so by using its own engine and only operating on in-memory data, Impala can provide significant performance improvements. We

compare our approach with Impala in Section 5.3.5.2.

The following two small-scale experiments evaluate the current feasibility of such approaches as compared to ours. Because these approaches are significantly different than ours, the experimental setup for each differs from our experimental setup as described in Section 5.3.1, and is described independently in each of the following subsections.

5.3.5.1 ETL Alternatives

Although we have used HV for our ETL process as noted in 5.3.1, alternative approaches exist and in this section we perform a small-scale experiment comparing our ETL method with a commercial open-source ETL tool. We first provide the details of our approach and then compare with an alternative approach.

For our approach, we have highly optimized our ETL process which uses Hive along with the DW's bulk loader. We found the two main bottlenecks to be the unpacking of JSON log files and the loading into DW. For JSON unpacking, we used a highly-optimized Perl unpacker, JSON::XS, which uses an external C-library. For DW loading, we optimized our commercial DW's bulk load utility by carefully configuring all of its available tuning parameters.

We surveyed the available ETL tools, and among the open-source tools, we found Pentaho/Kettle, Talend, and CloverETL to be the most popular. Other commercial ETL tools exist but we do not have access to these, so we chose an open-source tool. We found that the Pentaho/Kettle, Talend, and CloverETL tools all had very similar

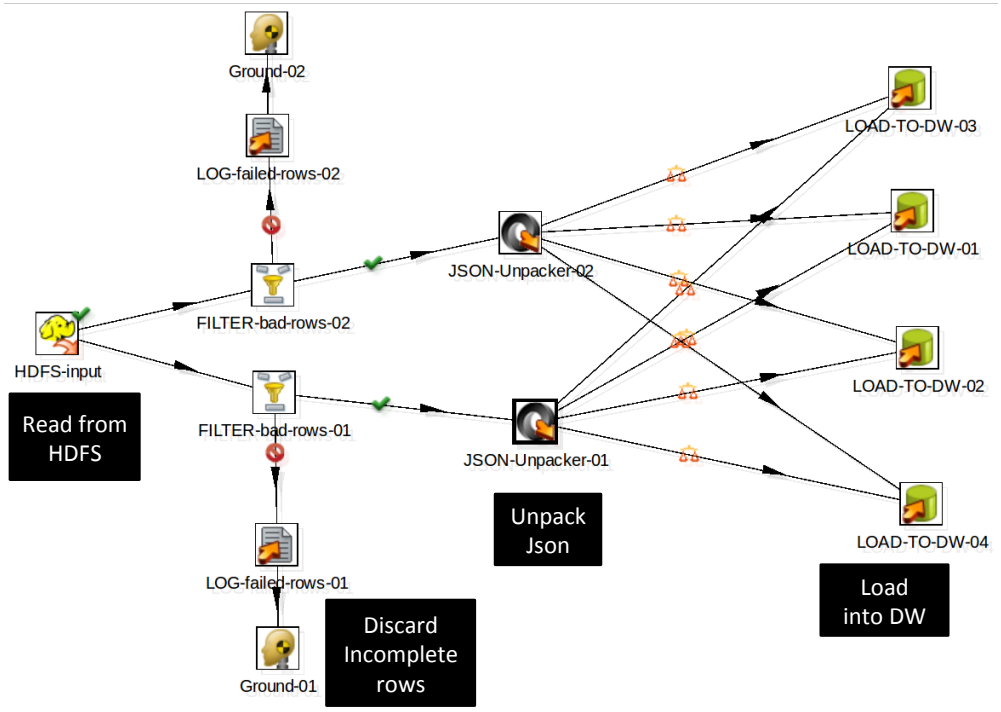

Figure 5.16: Our Pentaho/Kettle ETL flow for loading a JSON log into DW.

interfaces, so we selected the Pentaho/Kettle tool.

For our experimental setup, we installed and configured Pentaho/Kettle on an independent machine (2.66GHz Dual-core, 8GB RAM). We used a 1GB file containing a Twitter JSON log, containing one table with 13 attributes (int, double, and text fields). We then stored the file in HDFS on a small cluster of 3 machines running the latest Cloudera Hadoop release (release CHD5). Each machine in the HDFS cluster has 4 cores, 8 GB RAM, and one disk. We then performed ETL of that table into our DW using our HV approach and the the Pentaho tool and compare the total execution time of the ETL process for each approach. The DW we used is the same 9-node DW used in all of our other experiments, and is described in Section 5.3.1.

The ETL flow we designed for Pentaho/Kettle is shown in Figure 5.16. Our parallelized flow reads the file directly from our 3-machine Hadoop cluster, filters bad/incomplete rows, unpacks JSON, and parallel loads into DW. To obtain maximum parallelization from the tool, we kept adding parallel operators to the flow until we observed that the DW load rate was saturated.

It took the Pentaho/Kettle tool 1,475 seconds to load the data, while our method took 76 seconds. The Pentaho/Kettle tool could have improved significantly had it moved the filter-bad-rows and JSON unpacker into an MR process (i.e. scale-out) and used a bulk-load utility instead of JDBC connections. [2] But had it done this, it would be very similar to our ETL approach.

We have observed that many commercial enterprises use Pig or Hive to ETL big data into their data warehouses. Some examples include Netflix, Yahoo, and Intel. [3] [4] [5] One reason Hadoop has become a popular ETL tool is its ability to scale-out, compared with traditional ETL tools that can only scale-up on the machine where the tool runs. [6] These reasons along with our above experimental results indicate that the ETL approach we use is competitive and in-line with the current industry trends of folding ETL into Hadoop via Pig or Hive.

[2] In "Polybase: What, Why, How", David DeWitt reports similar loading performance issues with another open-source tool, Sqoop `http://gsl.azurewebsites.net/Portals/0/Users/dewitt/talks/PolybasePass2012.pptx`

[3] `http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html`

[4] `http://developer.yahoo.com/blogs/hadoop/pig-hive-yahoo-464.html`

[5] `http://hadoop.intel.com/pdfs/ETL-Using-Hadoop-on-Intel-Arch.pdf`

[6] `http://baboonit.be/blog/pigs-fly-how-to-build-a-blazingly-fast-etl-on-hadoop`

5.3.5.2 Comparison with Impala

Although we use HV (Hive) for our experiments, there is much interest recently in Impala. Here we compare our approach with Impala, but to perform a more direct comparison we do not use DW and only compare Impala directly with Hive in order to understand the tradeoffs with Impala and its applicability for our scenario.

For our experimental setup, we again use 3 machines with the latest Cloudera release of Hive and Impala (release CHD5). Each machine has 4 cores, 8 GB RAM, and one disk. This results in a total cluster memory size of 24 GB. We vary the base data size from 1 GB to 30GB since 30 GB is larger than the available memory in the cluster. Since Impala required the data to be ETL'd into a CSV format, we first converted our JSON Twitter and Foursquare logs to CSV format and then loaded into HDFS, so that both Hive and Impala execute the query on the same CSV data in HDFS. We chose one query for our experiment, query A_1v_1, which we simplified by removing the UDF and modifying the syntax in order to make it compatible with Impala SQL. The query counts the number of times users checkin to restaurants and the number of friends they have, identifying those users who checkin a lot and have a lot of friends. The query has 2 selections, 2 aggregations and 1 join. We executed this query for various data sizes from 1–30 GB, and report the results in Figure 5.17.

Figure 5.17 shows that for small data set sizes, Impala significantly outperforms Hive. This is because Impala achieve its speedups in part by relying on all of the data being memory-resident; hence when all data fits comfortably within the total

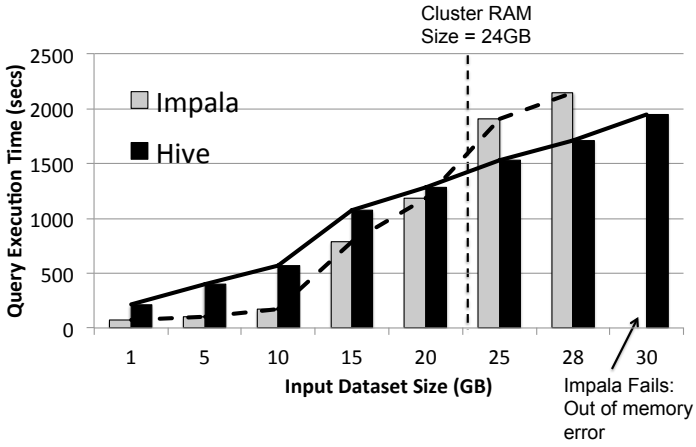

Figure 5.17: Execution time comparison of Hive and Impala for a single query with varying data sizes.

available memory Impala provides excellent performance. However, as the data set size approaches the total cluster memory size of 24 GB (indicated by the vertical dotted line in the figure), both systems begin to have similar performance. The dotted trend lines in Figure 5.17 highlight the cross-over point where Impala begins to perform worse than Hive. This is because once the data size becomes larger than the total cluster memory size, Impala performance suffers until it reaches the point where it fails to execute the query due to an out-of-memory error (indicated in the figure at the 30 GB mark along the x-axis). This is a known limitation of Impala since it currently requires one input to a JOIN to fit entirely within the main memory of every machine in the cluster, or the query will fail. This result shows that when dataset sizes are larger than the main memory size, Hive actually outperforms Impala, indicating that Hive is still a good candidate for processing large-scale data, as it easily scales to very large data sizes such as those used in our main experimental evaluation in Section 5.3. A recent benchmark study of

Impala and Hive (`https://amplab.cs.berkeley.edu/benchmark/`) corroborates our results.

Although Impala can provide excellent performance improvements over Hive, with the very large data sizes we consider in our work it may be impractical to expect all data to fit in memory during query processing. Since total cluster memory size is a resource constraint for an Impala system, it may not be a good substitute for Hive in our scenario with very large data. Impala also currently has other limitations with regard to join processing (as seen in the out-of-memory error in the figure) as well as query language restrictions (currently there is no support for UDFs, which are common in big data analytics). This indicates that least for the time being, Impala has limited functionality.

5.4 Discussion

Prior work on multistore systems has considered optimizing queries over data sets that span multiple stores. In this work we showed how DW can be used as an accelerator for big data queries. Our main insight was that tuning the multistore system is key to achieving good performance. For a multistore system to be effective, it should use both stores toward their strengths: HV is best used for sifting through massive volumes of big data and extracting a small yet relevant subset of the data; DW is best used for processing this small subset of data efficiently since it can be loaded quickly with a short-term impact on the DW queries. We argue that identifying this changing subset and determining where to store it is at the core of the concept of multistore

systems. Our lightweight adaptive tuning method realizes the benefits of multistore processing by leveraging the strengths of each store to ensure that the current relevant subset of data (i.e., opportunistic views) is present in the "best" store based on the observed workload which is changing dynamically.

By periodically tuning the physical design of a multistore system, we can achieve a significant speedup for big data queries simply by selectively utilizing existing DW resources. This means that as data analysts pose exploratory queries over big data, follow-up queries by the same analyst or queries by other analysts that access the same subset of data can benefit from the high-quality query processing of DW.

As noted in Section 5.3.3.1, choosing a good value for the view transfer budget B_t depends on many factors. Since DBAs use automated design tools for their enterprise RDBMS installations and hence are already familiar with setting values for physical design storage budgets (e.g., B_h, B_d), next we provide some guidance for setting the new parameter B_t, based on our experience.

Although B_t is expressed in terms of GB moved, it can be equivalently thought of in terms of total reorganization time. A small value for B_t is a natural choice for the problem setup we consider here. Larger values of B_t have several undesirable consequences: a larger impact to DW during reorganization phases (slowing down DW queries), tuning consumes a large portion of TTI, and each reorganization moves more data across the network. Given these observations, B_t should be kept small and DBAs can use the following three guidelines: (1) Set B_t to limit the amount of impact on the DW since reorganizations can cause significant but temporary impact as we show in the

next section. For example, setting B_t to a small value (i.e., a few GB) minimizes the DW impact, limiting the reorganization time. (2) Set B_t to a fraction of the average query execution time (e.g., $\frac{1}{3}$). Since we retune periodically (every R queries) as the workload changes, keeping tuning time small allows the system to quickly adapt the physical design to short-term changes in the workload. (3) Set B_t as a fraction of B_d by considering the maximum portion of the DW design to replace during each reorganization period. For example, setting B_t to $\frac{1}{4}$ of B_d, potentially allows the DW design to be completely replaced every 4 reorganization periods.

Finally, the experiments in Section 5.3.4 and Table 5.2 show that our approach has minimal impact on an existing DW workload. However, if this impact is still a concern, many commercial DW products allow queries to be assigned to priority classes, and the big data queries could be given a lower priority in the DW. Our cost model would take this query execution slowdown into account during calibration.

5.5 Related Work

Here we describe the related work categorized by each relevant area.

Multistore query processing. Earlier work from Teradata [99] and HadoopDB [3] tightly integrates Hadoop and a parallel data warehouse, allowing a query to access data in both stores by moving (or storing) data (i.e., the working set of a query) between each store as needed. These approaches are based on having corresponding data partitions in each store co-located on the same physical node. The purpose of co-locating data partitions is to reduce network transfers and improve locality for data access and loading

between the stores. However, this requires a mechanism whereby each system is aware of the other system's partitioning strategy; the partitioning is fixed, and determined up-front. More recent work [36, 89] presents multistore query optimizers for Hadoop and a parallel RDBMS store. These systems "split" a query execution plan across multiple stores. Data is accessed either in its native store, or via an external table declaration. A single query is executed across both systems by moving the working set between stores during query processing. These systems represent "connector approaches" between Hadoop and an RDBMS, using a data transfer mechanism such as Sqoop [11] (or their own in-house equivalent).

Recent work on Invisible Loading [2] addresses the high up-front cost of loading raw files (Hadoop data) into a DBMS (column-store). In the spirit of database cracking [51], data is incrementally loaded and reorganized in the DBMS by leveraging the parsing and tuple extraction done by MR jobs, and a small fraction of the extracted data is silently loaded by each MapReduce query. The data loaded represents horizontal and vertical sub-partitions of the Hadoop base data. This approach also represents opportunistic design tuning, but the views belong to a very specific class, i.e., horizontal and vertical fragments of base data. Furthermore, this approach requires the corresponding processing nodes of each store to be co-located on the same physical node (similar to the Teradata work). In this approach, the query acceleration obtained is primarily due to the improved data access times for HadoopDB since data is stored as structured relations in an RDBMS rather than flat files in HDFS. In contrast, our approach supports a more general class of materialized views, which essentially allows

us to share computation across queries instead of only improving access to base data. Moreover, we allow the big data system and the RDBMS to remain separate without modifying either system or requiring them to be tightly integrated.

Finally, multistore systems share some similarities with federated databases in that both use multiple databases working together to answer a single query. However, with a federated system there is typically a strong sense of data and system ownership, and queries are sent to the location that owns the data to answer the query. With our multistore scenario, the roles of the stores are different — queries are posed on the base data in HDFS, while the RDBMS is only used as a query accelerator.

Reusing MapReduce computations. Previous work has shown that intermediate results materialized during query processing in Hadoop can be cached and reused to improve query performance [37, 61]. The work in [37] reuses these results at the syntactic level. This means that a new query's execution plan must have an identical sub-plan (operations and data files) to a previous query in order to reuse the previous results. Some recent work [61] utilizes Hadoop intermediate results as opportunistic materialized views at the semantic level and reuses them by a query rewriting method. These methods all focus on a single system and it is not straightforward to extend these approaches to a multistore setup. In MISO we treat both the intermediate materializations in Hadoop and the working sets transferred during multistore query processing as opportunistic materialized views, and we reuse them at a semantic level. We then use these views to tune the physical design of both stores together.

Physical design tuning. Much previous work [15, 83, 84] has been done on online physical design tuning for centralized RDBMS systems. The objective of online tuning is to minimize the execution cost of a dynamically changing workload, including the cost to tune (reconfigure) the physical design. New designs may be materialized in parallel with query execution or during a reconfiguration phase. A reconfiguration phase may be time-constrained. For example, a new design may be materialized during a period of low system activity. Each of these prior works considers a single database system and uses indexes as the elements of the design. Our work considers a multistore system and utilizes opportunistic materialized views as the design elements. There may be a distinct reorganization phase [15, 83], whereby the new design is materialized, or it may be created in parallel with query processing [84]. Tuning a multistore system adds interesting dimensions to the online tuning problem, particularly since data is being transferred between stores (similar to reconfiguration) during query processing. Obtaining good performance hinges upon a well-tuned design that leverages the unique characteristics of the two systems.

Other recent work on tuning has addressed physical design for replicated databases [32] where each RDBMS is identical and the workload is provided offline. In contrast, our work addresses online workloads and the data management systems in a multistore scenario are not identical. Moreover, with multistore processing the systems are not stand-alone replicas but rather data (i.e., views) and computation may move fluidly between the stores during query processing.

Recent work [72] has shown how to apply multi-query optimization (MQO)

methods to identify common sub-expressions in order to produce a beneficial scheduling policy for a batch of concurrently executing queries. Due to the materialization behavior of MapReduce, these approaches can be thought of as a form of view selection, since the common sub-expressions identified are effectively beneficial views for a set of in-flight queries. However, MQO approaches require that the queries are provided up-front as a batch, which is very different from our online case with ad-hoc big data queries; hence, MQO approaches are not directly applicable to our scenario.

To summarize, all previous work has focused on either multistore processing for a single query or tuning a single type of data store. By dynamically tuning the multistore physical design, our approach utilizes the combination of these two stores to speed up a big data query workload.

Chapter 6

Towards a Workload for Evolutionary Analytics

A new analytical landscape has emerged, exemplified by the popularity of "big data" systems such as Hadoop as well as the recently added support for big data processing by all major data warehouse vendors [43, 48]. Data volumes are growing rapidly and log files are an important data source, e.g., social media or sensor data. Queries are often exploratory in nature, and system-facilitated data exploration has been proposed in [21,54,86,94]. Given this scenario, new requirements for analysis have been noted in [31], including the need to access "disparate, decentralized data" [40]. Analysis frequently includes complex processing methods such as user defined functions (UDFs), e.g., [10, 47, 67], created by expert users for domain-specific processing needs.

In this new analytical setting, data analysts and data scientists are becoming increasingly important to businesses [13, 39, 76]. Because analysts are tasked with finding value within their growing data sources, the speed at which an analyst can iterate through successive investigations to gain insight is crucial [60]. Recent work has shown that business decisions based upon this analysis can directly lead to improved business

performance [17]. To measure system performance, there is a need for a workload and metrics to capture this emerging type of analytics. It is important to understand the features of this new type of workload and effective ways to evaluate system performance in this space. We term this scenario *evolutionary analytics* and identify the following three important characteristics of evolutionary analytics that are not captured by existing benchmarks.

1. *Query Evolution.* Queries are exploratory and evolve over time. A query may go through multiple evolutions (versions) whereby an analyst iteratively formulates, tests, and refines hypotheses during investigation. Query revisions appear as a sequence of mutations to the original query, and this temporal nature is a key feature. While traditional interactive OLAP may perform operations such as roll-up or drill-down to slightly modify the query, revisions in exploratory analysis can include more types of changes to the query and a longer sequence of changes, as we define in Section 6.2.1. Typical revisions are minor refinements as well as more significant changes such as augmented functions, addition (or removal) of sub-queries, or incorporating new data sources to obtain richer answers.

2. *Data Evolution.* Queries may incorporate new data from external sources such as raw logs or local data files. New data sources should be easily ingested or accessible for use during query processing, and these data sources may have evolving schemas. A formal ETL (extract, transform, load) project for a data warehouse can have a very high cost in dollars and design time. Enabling access to diverse data sources

via ETL on-the-fly is a key feature of this new analytical environment.

3. *User Evolution.* A flexible and accessible system should enable new users to get started posing queries testing different hypotheses, potentially over old or new data sets. New users in the system arrive less frequently than query revisions, and their queries do not closely resemble another user's queries.

In this work, we propose a workload with these features and metrics to test how well a system supports them. Query response time is a primary metric but it is useful to understand system performance for other metrics as well. For example, response time may hide several other system overheads, such as the overhead to tune the physical design (if this happens online as queries get executed) or the cost to load data (if it has to be ETL'ed on-the-fly). By separating out these overheads in different metrics, we can see where each system excels and also understand how to develop hybrid systems that combine their best features.

Figure 6.1 shows our proposed metrics as dimensions (although not completely orthogonal). Query response time indicates how quickly analysts can arrive at answers when testing hypotheses. Tuning overhead represents the time expended for physical design tuning, i.e., creating indexes and materialized views, to improve query processing speed. Data-arrival-to-query-time indicates the time until newly arrived data is query-able. Storage in terabytes indicates the overhead for all data and auxiliary data structures (indexes and views). Cost in dollars represents the system cost to process the workload. Along each dimension, we indicate the relative performance of a traditional

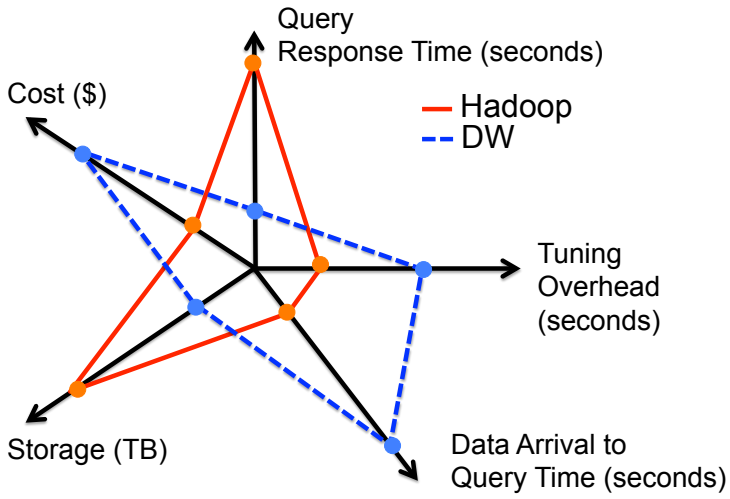

Figure 6.1: System metrics for evolutionary analytics.

data warehouse (DW) and a HADOOP system. This illustration shows how to compare and contrast different systems using our proposed metrics.

The outline of the remainder of this chapter is as follows. In Section 6.1 we examine real-world big data queries from several domains and describe their common characteristics. We then propose several relevant metrics for evolutionary analytics and describe their relative tradeoffs. In Section 6.2 we propose building blocks for our query workload that represent the common types of changes that we observed for query revisions in real-world environments. We then provide an overview of evolving queries along with an example showing several subsequent revisions of a query. In Section 6.3, we propose a benchmarking methodology that uses our metrics to capture and highlight the various performance tradeoffs made by a system under test; followed by Section 6.3.2 which provides experimental results using the benchmark and proposed metrics to compare and contrast four different system architectures. In Section 6.4 we

provide a discussion of the results and highlight how our benchmark metrics can be used to guide the design of future hybrid systems by analyzing the tradeoffs made by different system architectures. Lastly, in Section 6.5 we provide a detailed description of the queries in our proposed workload.

6.1 Workload characteristics and system metrics

In this section we first describe our workload properties and contrast with other benchmarks, then we describe our metrics to test system support for evolutionary analytics and highlight the various tradeoffs for each metric.

6.1.1 Workload Characteristics

Data analysis queries dealing with low-structured or log data must often perform data extraction tasks as well as analytical tasks, including the application of machine learning algorithms. Some recent examples of such queries are given in [75, 89]. These queries reference Twitter data and static data such as IMDB or historical business sales data. They use several UDFs which perform sentiment analysis and classification tasks. One query infers movie rating trends for two consecutive months, and the other computes the impact of a marketing campaign in different sales regions. These queries are representative of current data processing tasks.

Given our previously described workload needs and examples from recent data analysis tasks, we determine the following desirable characteristics for the query workload of the benchmark.

- Significant query complexity, including common UDFs, performing non-trivial anal-

ysis tasks to gain insights and find value within unproven data sources.

- Several successive versions for each query, representing data exploration during hypothesis testing and query refinement.
- Access realistic data sets during query processing. These should include raw logs and static/historical data sets.

Current analytical benchmarks such as TPC-H and TPC-DS [95] do not adequately capture this type of analytical workload. For instance, TPC-DS queries focus on known data and known reporting tasks, with a carefully designed, fixed schema for a data warehouse. This is not always possible in the current analytical scenario, as the use-case may not afford the up-front, top-down design of a traditional data warehouse. In contrast to query evolution where a query goes through an ordered sequence of mutations, the set of reporting queries in TPC-DS represent independent tasks where the ordering of one query is not dependent on the previously executed query. In contrast to data evolution, TPC-DS maintenance workloads reflect table inserts from its counterpart OLTP database, but they do not reflect a growing log or arrival of a new data source.

6.1.2 System Metrics

We propose the following metrics to evaluate a system for evolutionary analytics.

Query response time Query performance is a key metric of the benchmark, and measures total workload execution time. This metric serves as the primary indicator of how well a system is able to support the workload features and process the workload efficiently.

Tuning overhead. Physical design tuning can greatly improve query performance. Tuning might be considered offline during a system maintenance window or online during workload processing. This metric reports the cumulative time spent on tuning, which is the time spent to run a tuning tool and the time to materialize all indexes and views.

Data arrival to query time. This metric reports the time until newly arrived data is available to query. Data preparation is an atomic operation that enables the data to be accessed by a query. This may include the schema definition such as a CREATE TABLE statement and a LOAD operation.

Storage size. This metric indicates the total storage required in terabytes. Total storage includes that required for all base data, and all indexes and materialized views. Storage size can be asymmetrical even for base data, considering some systems replicate data by design (e.g., Hadoop).

Monetary cost. This metric indicates the total system cost for query processing and data storage. For simplicity, in this work we use dollar cost to include only machine time and storage cost. A better cost metric could be total cost of ownership (TCO), which includes system administration cost as well as hardware cost. The cost metric

can guide tradeoffs that are tolerable for a given environment, i.e., exploratory analysis where return on investment may not be known.

6.1.2.1 Metric tradeoffs.

Our metrics can be used to understand the various tradeoffs to consider for system design. Previous studies [77] have considered load times and query response time. Here we introduce additional metrics and show how they interact with each other. Clearly response time interacts with all of the other metrics of data loading, physical design tuning, storage space, and cost. Reducing response time can be achieved through a combination of tradeoffs among the other metrics.

For example, tuning overhead impacts both query response time and storage size. A good physical design can consume multiple times the size of the base data, but may reduce workload cost dramatically. Due to their size, the choice of indexes and views will also appear as a tradeoff along the storage metric. Loading may require data cleaning, transformation, and copying/storing the data, which is a typical ETL task in a data warehouse. In contrast, using Hive [94] requires only the schema definition to be provided before a query can access the data. This presents a tradeoff between query response time and data load time.

The cost metric leads to interesting tradeoffs for system design. In particular, the advent of the cloud enables pay-as-you-go performance, allowing for a rich set of choices for query processing. For example, Hadoop [8,42], databases [8,69], and recently even petabyte-scale data warehouses (e.g.,Redshift [8]) are all available on-demand.

Moreover, a mixture of systems may be used for query processing as we show later in Section 6.3.2.

The importance of each metric may be weighed differently for a particular environment. The purpose of including all five of them is to help understand the impact of various tradeoffs in order to guide system design. Next we describe the specifics of our workload and how to evaluate system performance using these metrics.

6.2 The Workload

Our workload considers 8 hypothetical analysts who write queries for marketing scenarios involving restaurants using social media data and static data. For social media data we use a sample of the Twitter data stream and user check-in data from Foursquare. For static data we include a Landmarks data set (landmark locations). Each analyst poses one query which is then revised multiple times. There are 4 versions of each query, representing the original query and 3 subsequent revisions. Next we define the types of changes allowed for each revision, and then provide a workload that uses these changes.

6.2.1 Query building blocks

Queries that evolve during exploratory data analysis may follow certain patterns of common changes. As an analyst revises a query, she may tweak the selectivity to produce greater or fewer answers, include additional data sources for stronger evidence of hypothesis, add a UDF to perform a specialized processing function, or refine the results by including or removing a query sub-goal as more is learned about the data after each query revision.

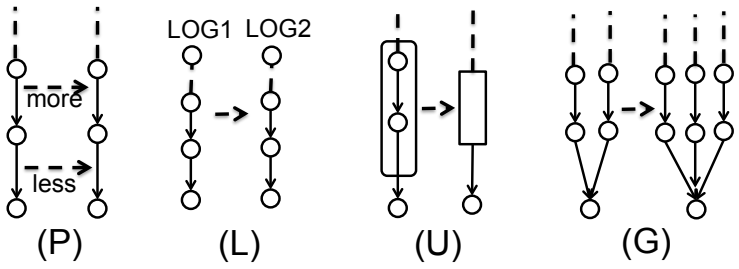

Figure 6.2: Dimensions of change for query revisions

To make these changes concrete, we evaluated complex analytical queries from several sources to find evidence of the manner in which queries evolve. The TPC-DS [95] workload includes 4 interactive OLAP queries that go through 2 revisions each. Taverna [92] queries on MyExperiment [71] are scientific queries that retain all of their revisions. Each of the top 10 most-downloaded Taverna queries had 2–11 revisions. Yahoo! Pipes [100] has many versions of user queries over open-access web data, with more than 99 data sources. Queries in Pipes are easily clone-able and modified by any user, and in one instance we observed a query with more than 49 revisions. These observations suggest that queries in evolutionary analytics typically go through several revisions.

Specifically, we commonly observed the following 4 types of changes during query revisions from a sampling of [92, 95, 100]. We note these changes are not mutually exclusive nor exhaustive but representative.

(P) Parameters: The query parameters are modified to obtain slightly different results (Figure 6.2P). For example, the analyst may alter a selection predicate or a top-k value to allow more or less data in the output.

(L) Logs: An analyst may make use of an additional data source in the query to obtain richer results (Figure 6.2L).

(U) UDFs: An analyst may add or replace a set of operations in the query with a specialized UDF.

(G) Sub-Goals: Typically, an analyst writes several sub-queries that each achieves a single goal and then joins these to obtain the final output (Figure 6.2G). A revision may add or remove a sub-goal.

These four dimensions {P,L,U,G} serve as our evolving query building blocks. For each query revision, the changes are expressed by one or more of these dimensions.

6.2.2 Queries

We give a high-level description of each query scenario in Table 6.1 left column, and the right column specifies the change from one version to the next in terms of our query building blocks. For instance, let Q represent the analyst's first version of the query. Then each subsequent version (i.e., 2, 3, 4) is represented by indicating the dimensions that were changed during each revision in the following way:

$$Q \rightarrow \{P, L, G\} \rightarrow \{P\} \rightarrow \{P, G\}$$

In this representation version 2 of the query Q revises the query along the P, L, and G dimensions. Version 3 revises the query along the P dimension. Version 4 revises the query along the P and G dimensions. The representation captures the types of revisions and the temporal order in which they are applied.

Analyst 1 wants to identify a number of "wine lovers" to send them a coupon for a new wine being introduced in a local region. The evolution of this query focuses on focuses on finding suitable users to whom sending a coupon would have the most business impact.	$Q \rightarrow \{P, L, G\} \rightarrow \{P\} \rightarrow \{P, G\}.$
Analyst 2 wants to find influential users who visit a lot of restaurants for inclusion in an advertisement campaign. The evolution of this query focuses on identifying users who are "foodies" by using increasingly sophisticated methods.	$Q \rightarrow \{L, U, G\} \rightarrow \{P, G\} \rightarrow \{P, G\}.$
Analyst 3 wants to start a gift recommendation service where friends can send a gift certificate to a user u_1. We want to generate a few restaurant choices based on u_1's preferences and his friend's preferences. The evolution of this query focuses on generating a diverse set of recommendations that would cater to u, and u's close set of friends.	$Q \rightarrow \{P, G\} \rightarrow \{P, L, G\} \rightarrow \{G\}.$
Analyst 4 wants to identify a good area to locate a sports bar. The area must have a lot of people who like sports and check-in to bars, but the area does not already have too many sports bars in relation to other areas. The evolution of this query focuses on identifying a suitable area where there is high interest but a low density of sports bars.	$Q \rightarrow \{U, G\} \rightarrow \{L, U, G\} \rightarrow \{U, G\}.$
Analyst 5 wants to give restaurant owners a customer poaching tool. For each restaurant r, we identify customers who go to a "similar" restaurant in the area but do not visit r. The owner of r may use this to target advertisements. The evolution of this query focuses on determining "similar" restaurants and their users.	$Q \rightarrow \{L, G\} \rightarrow \{L, U\} \rightarrow \{P, G\}.$
Analyst 6 tries to find out if restaurants are losing loyal customers. He wants to identify those customers who used to visit more frequently but are now visiting other restaurants in the area so that he can send them a coupon to win them back. The evolution of this query focuses on identifying prior active customers.	$Q \rightarrow \{L, G\} \rightarrow \{P, G\} \rightarrow \{P, G\}.$
Analyst 7 wants to identify the direct competition for poorly-performing restaurants. He first tries to determine if there is a more successful restaurant of similar type in the same area. The evolutionary of this query focuses on identifying good and bad restaurants in an area, as well as what customers like about the menu, food, service, etc. about the successful restaurants in the area.	$Q \rightarrow \{L, G\} \rightarrow \{G\} \rightarrow \{U, G\}.$
Analyst 8 wants to recommend a high-end hotel vacation in an area users will like based on their known preferences for restaurants, theaters, and luxury items. The evolution of this query focuses on matching a user's preferences with the types of businesses in a geographical area.	$Q \rightarrow \{L, G\} \rightarrow \{U, G\} \rightarrow \{P, L, U, G\}.$

Table 6.1: Business marketing scenarios for 8 analysts, along with the dimensions modified during each of the 4 evolutions.

To illustrate this process, we start with Example 1 as the first version of Analyst 1's query in Table 6.1. This version is indicated by Q above. Analyst 1 first desires to find users who like wine, are affluent, and have many good friends.

Example 1. *(a): EXTRACT user from Twitter log. Apply UDF-CLASSIFY-WINE-SCORE on each user tweet to obtain a wine-score. Groupby user, compute a wine-sentiment-score for each user.*

(b): From Twitter log, apply UDAF-CLASSIFY-AFFLUENT on tweets to classify a user as affluent or not.

(c): From Twitter log, create social network between every user pair using tweet source and dest. GROUPBY user pair in social network, count tweets. Assign friendship-strength-score to each user pair.

JOIN (a),(b), and (c). Threshold based on wine-sentiment-score, friendship-strength-score.

Next the analyst wants to find more evidence that the user likes wine. She revises the query by changing {P,L,G}, adding two new data sources $\{L\}$ (Foursquare and Landmarks), a new sub-goal $\{G\}$ that computes a checkin-count for users who go to wine places, and decreases the threshold parameter $\{P\}$ for wine-sentiment-score since she will have evidence a user likes wine from 2 data sources. Example 2 below describes version 2 of the query.

Example 2. *(d): EXTRACT from Foursquare log. For each checkin, obtain the user and restaurant name. Using the Landmarks data, filter by checkin to places of type wine-bar. Groupby user, compute checkin-count.*

(e): Decrease wine-sentiment-score threshold

JOIN (a),(b),(c) and (d). Threshold based on new wine-sentiment-score in (e), friendship-strength-score.

Versions 3 and 4 of this query are revised in a similar way. A more detailed description of all queries in the workload is provided in later in Section 6.5.

6.3 Running the Benchmark

We now present our benchmark methodology for query evolution, user evolution, and data evolution and we show an example of benchmark results. We consider the initial system state to be *idle*, with no previously loaded data or executed queries.

6.3.1 Benchmark methodology

Query evolution. This test will use all analysts 1–8 and all query versions from each analyst. (1) From initial system state, execute analyst 1 query versions 1 through 4 in succession, returning to initial system state before each version. (2) From initial system state, execute analyst 1 query versions 1 through 4 in succession, without returning to initial system state before each version. Compare metrics from (1) and (2), and repeat for each remaining analyst. This comparison highlights a system's ability to process any repeating tasks from the same user.

User evolution. This test will use all analysts 1–8 but only version 1 of each analyst's query. First, assume some order of analysts 1–8. (1) From initial system state, execute each analyst's query in the chosen order, returning to initial system state before each query. (2) From initial system state, execute each analyst's query in the chosen order, without returning to initial system state before each query. Compare metrics from (1) and (2). This comparison highlights a system's ability to process similar tasks from different users.

Data evolution. This test will use a single data source, e.g., Twitter log, and the subset of data requested in the first step should be a number of columns equal to half of the total number of columns in the log schema. The columns should be randomly chosen each time. (1) From an initial system state, an analyst requests a subset of data from a new data source. (2) An analyst requests one *additional* attribute from the data source in (1), in each successive version of the query. (3) A new analyst requests a subset of data previously accessed by the analyst in (1). (4) Repeat (1), (2), (3) returning to the initial state after each query. Compare metrics from (1), (2), (3) with (4). This comparison highlights a system's ability to access subsets of data from a new data source on demand.

6.3.2 Example benchmark results

Next, we briefly show a sample reporting on the relative performance of four data systems using our workload and metrics for a user evolution scenario. The experimental setup consists of 9 nodes running a widely used commercial parallel data warehouse (DW) and 14 nodes running Hadoop. The ratio of Hadoop nodes to DW nodes is 1.5×. The DW and Hadoop clusters are independent, and nodes are connected with 1 GbE. Each node has two 2.4 GHz XEON CPUs and a local 2 TB disk. In this test, our data includes a 1 TB Foursquare log, a 1 TB Twitter log and 12 GB Landmarks log. We use Hive [94] to execute our queries on the Hadoop system. Since all systems utilize the base data stored in Hadoop, we omit this from the storage metric.

Figure 6.3 reports the results for the user evolution scenario. HADOOP corre-

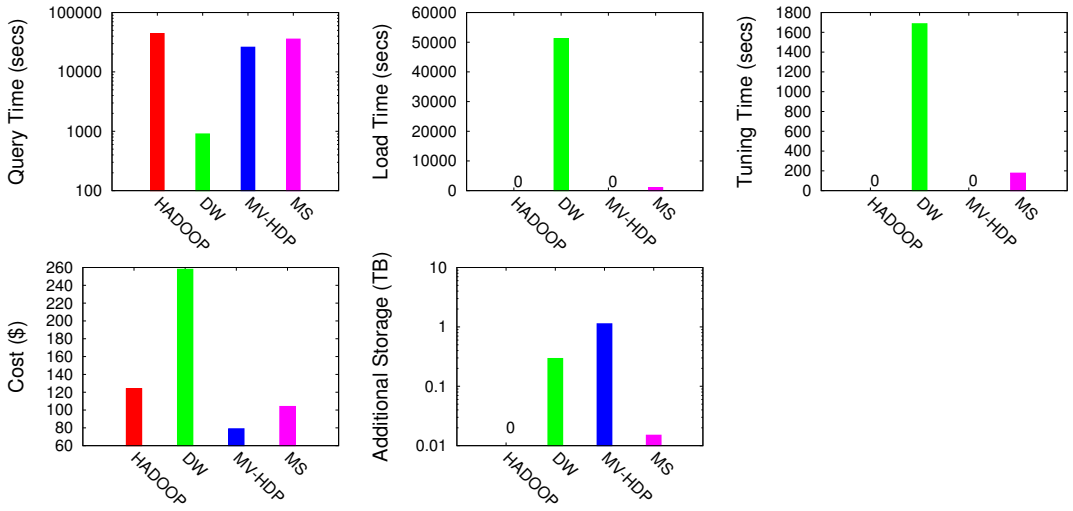

Figure 6.3: Sample benchmark reporting of 4 data systems on a user evolution scenario.

sponds to a Hadoop-only execution of the query. DW executes the query on the DW but uses Hadoop as an ETL tool to extract the subset of data required by the query. MV-HDP corresponds to a system that we developed in [63] that rewrites Hadoop queries based on opportunistic materialistic views left behind from prior execution runs. MS is an implementation of a multistore query optimizer (similar to [89]) that uses both Hadoop and a data warehouse together to execute each query. For MS, the load time refers only to the time to move and then load data on-the-fly from HADOOP to DW after partial execution in HADOOP.

The benefit of reporting on all 5 metrics in Figure 6.3 is that it exposes their various tradeoffs, which are not easily captured when reporting just on the query execution time. Since Hadoop performs ETL on the fly, query performance is quite poor compared to the DW. On the other hand, the superior performance of the DW is offset by the high cost of loading the data into the data warehouse. Both MV-HDP and

MS show tradeoffs that reduce query response time. HADOOP and MV-HDP do not incur any tuning overhead whereas DW and MS require a tuning phase to provide good performance. We used Amazon EC2 and Redshift [8] pricing to approximate the dollar cost of the machines and storage (using cost for machines similar to those in our in-house clusters). The cost values show that HADOOP is far cheaper than DW while MS is cheaper than both, and MV-HDP has the lowest cost. Finally, it can be seen that MV-HDP incurs a significant storage overhead by retaining results as opportunistic views from all the prior executions runs. The tradeoff with storage size improves query response time compared to HADOOP.

6.4 Discussion

In the new analytical space, the key question is how to design systems to address emerging needs. The continued popularity of Hadoop and data warehouses notwithstanding, these are only suitable when the required use-case matches either of their starkly different characteristics. One focuses on being able to query the data right away, tolerating lesser performance. The other focuses on performance at the expense of significant delay in being able to query the data. These systems represent two ends of a spectrum, and the influx of so many new data processing systems shows that these two distinct choices are not meeting all current needs.

Our proposed metrics highlight the various tradeoffs among many design choices and the metrics can be used to guide system development. For example, we show 2 systems that remedy one dimension by shifting the tradeoff with another dimen-

sion. With MV-HDP we show that increased storage leads to better performance than Hadoop. With MS we show that one can remedy the loading time of a data warehouse to an extent by sacrificing some of DW performance. The design of future system architectures should consider combining the best properties of different systems to create new or hybrid systems, keeping these tradeoffs in mind. The best system design will depend upon the needs of a specific deployment environment, but it may be possible to design systems that dynamically self-adjust the tradeoffs made, adapting to changing needs. For instance, by configuring the multistore tuning parameters B_h, B_d, B_t presented in Chapter 5.2.

6.5 Workload Details

In this section we describe our proposed workload in more detail. First, in Section 6.5.1 we describe the datasets referenced by the queries. Second, in Section 6.5.2 we describe the UDFs used by the queries. Lastly, in Section 6.5.3 we describe each of the queries in detail.

During a typical exploration, and analyst may spend a lot of time identifying the data distribution to get an understanding of where the density and sparsity lies. For this reason, the queries below have parameters indicated by an underline that an analyst would modify in order to obtain a representative answer set. Choosing an appropriate value or interpretation of the underlined parts of the queries is a necessary step an analyst performs. This may require a trial and error process resulting in additional versions of the queries. We leave these values unspecified as they are a function of the

real-world datasets used.

Finally, it is important to note that the queries are not rigid interpretations of a goal, but rather one approach toward answering a question. For example, there may be several ways to interpret what it means to be "good" friends in a social network. Furthermore, even for a particular interpretation, there may be several ways to express it as a query. Hence, other variations of a query are possible.

6.5.1 Datasets

The queries utilize three datasets:

1. A Twitter data stream of user tweets, denoted as TWTR.
2. A Foursquare data stream of user checkins, denoted as 4SQ.
3. A Landmarks log of locations and their types, denoted as LAND.

We assume that the identity of users is common across the TWTR and 4SQ datasets, while identity of locations is common across the 4SQ and LAND datasets. Furthermore, we assume that the data is log data, e.g., flat text data perhaps stored in JSON format.

6.5.2 User Defined Functions referenced by queries

The following queries reference multiple UDFs, for which we provide a brief description below. Text classifiers can be implemented using a *bag-of-words* method. For example, classes such as COFFEE-DRINKER may include the words (*coffee, espresso, latte, french press*) while WINE-LOVER may include (*cabernet, vineyard, merlot, chardonnay*). By accepting a bag of words as an argument, these UDFs are easily reusable. However,

this does not exclude other classification methods. UDAFs perform a groupby on a key and apply similar classification on the elements of a group.

1. UDF-CLASSIFY-WINE-SCORE: Input is text and output is wine-score indicating a strong presence of wine-terms.
2. UDF-CLASSIFY-FOOD-SCORE: Input is text and output is food-sentiment-score indicating a strong presence of food-terms.
3. UDF-GRID-CELL: Input is lat-lon coordinates and grid resolution, and output is a grid-cell number.
4. UDF-CLASSIFY-BEER-SCORE: Input is text and output is beer-score indicating a strong presence of beer-terms.
5. UDF-MENU-SIMILARITY: Input is two lists of menu items and output is a score indicating the similarity of the lists.
6. UDF-NLP-ENTITITY-SENTIMENT: Input is text and output is the entities extracted from the text with a sentiment-score for each entity.
7. UDF-CLASSIFY-LUXURY-SCORE: Input is text and output is a binary value indicating if the text concerns luxury items.
8. UDF-SENTIMENT: Input is text and output is a sentiment-score expressing positive or negative sentiment with the score indicating the strength of the sentiment.
9. UDAF-CLASSIFY-AFFLUENT: Input is all text from a given user and output is a binary value indicating if the user is affluent or not.

10. UDAF-CLASSIFY-SPORTS: Input is all text from a given user and output is a binary value indicating if the user is interested in sports or not.

6.5.3 Query Descriptions

Here we describe each of the 32 queries of our workload in more detail. Given a different business scenario for each of the 8 analysts as noted in Table 6.1, we describe four query versions. These represent the analyst's original query and three subsequent revisions of the query as it is refined during data exploration. They are denoted by $A_i v_1, A_i v_2, A_i v_3, A_i v_4$ indicating *Analyst i* query versions 1–4. Each query version first states a high-level goal of what the analyst is trying to achieve, followed by a description of the query toward the stated goal. For each query, we describe the task rather than provide an implementation in particular language since many are possible, e.g., SQL, HiveQL, Pig, Java, etc.

6.5.3.1 Analyst 1

Analyst 1 wants to identify a number of "wine lovers" to send them a coupon for a new wine being introduced in a local region. The evolution of this query focuses on focuses on finding suitable users to whom sending a coupon would have the most business impact.

Analyst 1, version 1 ($A_1 v_1$)

- Analyst goal: Find users that like wine, have strong friendships, and are affluent.
- Query: From TWTR, apply UDF-CLASSIFY-WINE-SCORE on each user's tweets and groupby user to produce wine-sentiment-score for each user. Threshold on wine-

sentiment-score above $\underline{x_1}$.

From TWTR, compute all pairs $\langle u_1, u_2 \rangle$ of users that communicate with each other, assigning each pair a friendship-strength-score based on the number of times they communicate. Threshold on friendship-strength-score above $\underline{x_2}$.

From TWTR, apply UDAF-CLASSIFY-AFFLUENT on users and their tweets.

Join results by user.

Analyst 1, version 2 (A_1v_2)

- Analyst goal: Next, consider users to be wine-lovers if they checkin to many wine places.
- Query: From previous version, reduce wine-sentiment-score threshold to $\underline{x_2'}$ since now there will be additional evidence a user likes wine.

  From 4SQ, identify places that users checkin. Join with places in LAND. Select users that checkin to places of type wine-bar. For each user, count the number of checkins. Threshold on checkin-count above $\underline{x_3}$.

  Join these with users from previous version.

Analyst 1, version 3 (A_1v_3)

- Analyst goal: Now find users that are also in the San Francisco area as well as prolific on Twitter.
- Query: From previous version, select users local to San Francisco. Threshold on tweet-count above $\underline{x_4}$. Adjust x_1, x_2, x_3 appropriately to produce "enough" answers.

Analyst 1, version 4 (A_1v_4)

- Analyst goal: Finally, require that a user's friends must also visit wine-places.
- Query: For user pairs $\langle u_1, u_2 \rangle$, threshold on u_2 checkin-score above $\underline{x_5}$. For each user u_1, count the number of friends with checkin-count above threshold. Retain u_1 if count above $\underline{x_6}$. Join these with users from previous version.

6.5.3.2 Analyst 2

Analyst 2 wants to find influential users who visit a lot of restaurants for inclusion in an advertisement campaign. The evolution of this query focuses on identifying users who are "foodies" by using increasingly sophisticated methods.

Analyst 2, version 1 (A_2v_1)

- Analyst goal: Find users who frequently visit restaurants.
- Query: From 4SQ, identify places that users checkin. Join with places in Landmark log. Select users that checkin to places of type restaurant. For each user, count the number of times they checkin to a place of type restaurant. Compute the normalized-count based on the maximum count across all users. Threshold on normalized-count above $\underline{x_1}$.

Analyst 2, version 2 (A_2v_2)

- Analyst goal: Additionally, user also likes food if they talk positively about food.
- Query: From TWTR, apply UDF-CLASSIFY-FOOD-SCORE on each user's tweets and groupby user to produce food-sentiment-score for each user. Threshold on food-

sentiment-score above $\underline{x_2}$.

Join these users with users in previous version.

Analyst 2, version 3 (A_2v_3)

- Analyst goal: Further define that a user likes food if they dine at many different types of restaurants.
- Query: Revise previous version by counting the number of times a user has visited each distinct type of restaurant. Select users who have visited $\underline{x_3}$ distinct restaurant types at least $\underline{x_4}$ times.

Analyst 2, version 4 (A_2v_4)

- Analyst goal: Finally, require that these users do not frequently visit restaurants with low ratings.
- Query: From previous version, compute the percentage of each user's checkins to restaurants with ratings less than $\underline{x_5}$. Threshold on percent below $\underline{x_6}$.

6.5.3.3 Analyst 3

Analyst 3 wants to start a gift recommendation service where friends can send a gift certificate to a user u. We want to generate a few restaurant choices based on u's preferences and u's friend's preferences. The evolution of this query focuses on generating a diverse set of recommendations that would cater to u, and u's close set of friends.

Analyst 3, version 1 (A_3v_1)

- Analyst goal: For each user u, identify those restaurants that u's good friends frequently visit.
- Query: From TWTR, compute all pairs $\langle u_1, u_2 \rangle$ of users that communicate with each other, assigning each pair a friendship-strength-score based on the number of times they communicate. Threshold on friendship-strength-score above $\underline{x_1}$.

  From 4SQ, identify places that users checkin. Join with places in LAND. Select users that checkin to places of type restaurant.

  For each user u_1, find all the restaurants that her friends u_2 have visited. For each restaurant, count the number of checkins. Threshold on count above $\underline{x_2}$.

Analyst 3, version 2 (A_3v_2)

- Analyst goal: Next, only consider users that have friends in the same area as well as other friends in common.
- Query: Revise the previous version by redefining what it means to be good friends. From TWTR, recompute all pairs $\langle u_1, u_2 \rangle$ of users that live in the <u>same</u> area, and have a friendship-strength-score above $\underline{x_3}$. Additionally, a user pair $\langle u_1, u_2 \rangle$ are said to be good friends if they have more than $\underline{x_4}$ friends in common.

Analyst 3, version 3 (A_3v_3)

- Analyst goal: Next, identify only those restaurants that are the same type as a user's favorite restaurant.

- Query: From 4SQ, for each user u_1, find favorite restaurant type by counting the number of checkins to each restaurant, and select the restaurant with the max number of checkins as u_1's favorite restaurant r.

  Join with LAND to obtain r's type.

  From the previous version, select only those restaurants for u_1 that belong to the same type as u_1's favorite type.

6.5.3.4 Analyst 3, version 4 (A_3v_4)

- Analyst goal: Finally, find additional restaurants that are similar to those visited by a user's friends.

- Query: From 4SQ, for all restaurant pairs $\langle r_1, r_2 \rangle$, count the number of users that have visited both restaurants. Threshold on count above $\underline{x_5}$. All remaining pairs $\langle r_1, r_2 \rangle$ are considered to be similar since they have many common customers. For each user u_1, suggest r_2 if r_1 is a restaurant frequently visited by u_1's friends.

6.5.3.5 Analyst 4

Analyst 4 wants to identify a good area to locate a sports bar. The area must have a lot of people who like sports and check-in to bars, but the area does not already have too many sports bars in relation to other areas. The evolution of this query focuses on identifying a suitable area where there is high interest but a low density of sports bars.

Analyst 4, version 1 (A_4v_1)

- Analyst goal: Find users who like beer and where they live.
- Query: From 4SQ, identify users and their location that frequently mention the word "beer" in their text. For each user, count the occurrences of the word. Threshold on count above $\underline{x_1}$.

Analyst 4, version 2 (A_4v_2)

- Analyst goal: Next, find areas where there are many beer lovers.
- Query: From the previous version, use UDF-GRID-CELL to map user locations to a grid cell. Count number of users in each grid cell. Threshold on count above $\underline{x_2}$.

Analyst 4, version 3 (A_4v_3)

- Analyst goal: Next, find areas with many users that like beer and sports but do not have many sports bars.
- Query: From TWTR, apply UDAF-CLASSIFY-SPORTS on users and their tweets. Then apply a UDF-CLASSIFY-BEER-SCORE to better identify users that like beer, and produce a beer-score for each user. Join sports and beer users. Threshold on beer-score above $\underline{x_3}$. Next, apply UDF-GRID-CELL to map user locations to a grid cell. Count number of users in each grid cell. Threshold on count above $\underline{x_4}$.

  From LAND, obtain restaurant name, type and location. Select places that are type equal to sports bar. Next, apply UDF-GRID-CELL to map place locations to a grid cell. Count number of restaurants in each grid cell. Threshold on count below $\underline{x_5}$.

Join grid cells from user locations and sports bar locations.

Analyst 4, version 4 (A_4v_4)

- Analyst goal: Finally, find area with high user interest but few popular sports bars relative to the number of users.
- Query: From 4SQ, identify places that users checkin.

  Join with places in LAND. Select places that are type equal to sports bar. For each place, count the number of checkins. Threshold on count above $\underline{x_6}$.

  Next, apply UDF-GRID-CELL to map place locations to a grid cell. Count the number of places per grid cell.

  Join this with the grid cells from previous version. Threshold on ratio of user to sports bars count above $\underline{x_7}$.

6.5.3.6 Analyst 5

Analyst 5 wants to give restaurant owners a customer poaching tool. For each restaurant r, we identify customers who go to a "similar" restaurant in the area but do not visit r. The owner of r may use this to target advertisements. The evolution of this query focuses on determining "similar" restaurants and their users.

Analyst 5, version 1 (A_5v_1)

- Analyst goal: Find similar restaurants based the overlap of users that checkin to each place.

- Query: From 4SQ, for all restaurant pairs $\langle r_1, r_2 \rangle$, count the number of users that have visited both restaurants. Threshold on count above $\underline{x_1}$.

Analyst 5, version 2 (A_5v_2)

- Analyst goal: Next, find restaurants that are similar as indicated by a user or the user's friends frequently visiting the same places.

- Query: From TWTR, compute all pairs $\langle u_1, u_2 \rangle$ of users that communicate with each other, assigning each pair a friendship-strength-score based on the number of times they communicate. Threshold on friendship-strength-score above $\underline{x_2}$.

  From 4SQ, for all restaurant pairs $\langle r_1, r_2 \rangle$, count the number of users that have visited both restaurants, as well as the number of times a user u_1 has visited r_1 and one of their friends u_2 has visited r_2. Threshold on count above $\underline{x_3}$.

Analyst 5, version 3 (A_5v_3)

- Analyst goal: Next, find restaurant pairs that are also similar based on the similarity of their menus.

- Query: From LAND, create restaurant pairs $\langle r'_1, r'_2 \rangle$ that have the same zip code and type. For each pair, apply UDF-MENU-SIMILARITY to obtain menu-similarity-score. Threshold on menu-similarity-score above $\underline{x_4}$.

  Join pairs $\langle r'_1, r'_2 \rangle$ with pairs $\langle r_1, r_2 \rangle$ from the previous version.

Analyst 5, version 4 (A_5v_4)

- Analyst goal: Finally, find users that visit one restaurant but not a similar restaurant.
- Query: From 4SQ, for each restaurant r, identify the users that have visited r and the count of times they have visited. For each restaurant pair $\langle r_1, r_2 \rangle$ from the previous version, select users u that have visited r_1 more than $\underline{x_5}$ times and visited r_2 less than $\underline{x_6}$ times.

6.5.3.7 Analyst 6

Analyst 6 tries to find out if restaurants are losing loyal customers. He wants to identify those customers who used to visit more frequently but are now visiting other restaurants in the area so that he can send them a coupon to win them back. The evolution of this query focuses on identifying prior active customers.

Analyst 6, version 1 (A_6v_1)

- Analyst goal: For each restaurant, identify other restaurants with the same zip code and type that are less popular.
- Query: From LAND, create restaurant pairs $\langle r_1, r_2 \rangle$ that have the same zip code and type, and r_2 has a $\underline{\text{much}}$ lower checkin count than r_1.

Analyst 6, version 2 (A_6v_2)

- Analyst goal: Now, identify restaurants that have lately become less popular.
- Query: From 4SQ, identify places that users checkin.

Join with places in LAND that have type restaurant. For each restaurant, compute the average number of checkins per month in the last $\underline{x_1}$ months and the number of checkins in the last 1 month.

Threshold on the ratio of recent checkins to historical average checkins below $\underline{x_2}$.

Analyst 6, version 3 (A_6v_3)

- Analyst goal: Next, find users that stopped visiting those restaurants.
- Query: For restaurant r identified as becoming less popular in the previous version, and a user u that visited r, compute the average number of checkins per month by user u in the last $\underline{x_3}$ months and the number of checkins by user u in the last 1 month. Compute the ratio of recent checkins to historical average checkins

  Threshold on ratio below $\underline{x_4}$.

Analyst 6, version 4 (A_6v_4)

- Analyst goal: Finally, find users that no longer frequent a particular restaurant but still visit other restaurants in the same area.
- Query: From 4SQ, for each user u, count the number of checkins by zip code. Threshold on count above $\underline{x_5}$.

  For each less popular restaurant r identified in previous version, retain u only if u still frequently visits restaurants in the same zip code as r.

6.5.3.8 Analyst 7

Analyst 7 wants to identify the direct competition for poorly-performing restaurants. He first tries to determine if there is a more successful restaurant of similar type in the same area. The evolutionary of this query focuses on identifying good and bad restaurants in an area, as well as what customers like about the menu, food, service, etc. about the successful restaurants in the area.

Analyst 7, version 1 (A_7v_1)

- Analyst goal: For each zip code, identify good and bad restaurants.
- Query: From LAND, identify places that are restaurants, and apply UDF-SENTIMENT on the restaurant comments to obtain a sentiment-score. For each zip code, retain restaurants with sentiment-score above $\underline{x_1}$ or below $\underline{x_2}$ as good and bad restaurants.

Analyst 7, version 2 (A_7v_2)

- Analyst goal: Next, refine the discrimination of restaurants as good and bad based on their popularity.
- Query: From 4SQ, obtain the checkin count for every restaurant.

  Threshold on count above $\underline{x_3}$. Join with the good restaurants from the previous version.

  Threshold on count below $\underline{x_4}$. Join with the bad restaurants from the previous version.

Analyst 7, version 3 (A_7v_3)

- Analyst goal: Further discriminate restaurants as good and bad based repeat checkins.
- Query: From 4SQ, obtain the checkin count for every restaurant. For each restaurant, count the number of users that checkedin only once, and the number of users that checkedin more than $\underline{x_5}$ times. Compute the ratio of single checkins to multiple checkins.

  Threshold on ratio below $\underline{x_6}$. Join with the good restaurants from the previous version.

  Threshold on ratio above $\underline{x_7}$. Join with the bad restaurants from the previous version.

Analyst 7, version 4 (A_7v_4)

- Analyst goal: Next, for each restaurant, find the most frequent entities with positive and negative comments.
- Query: From 4SQ, apply UDF-NLP-ENTITITY-SENTIMENT per user checkin. For each restaurant and each entity, aggregate the sentiment-score. Threshold on sentiment-score above $\underline{x_5}$.

  Join with good and bad restaurants from previous version.

6.5.3.9 Analyst 8

Analyst 8 wants to recommend a high-end hotel vacation in an area users will like based on their known preferences for restaurants, theaters, and luxury items. The evolution of this query focuses on matching a user's preferences with the types of businesses in a geographical area.

Analyst 8, version 1 (A_8v_1)

- Analyst goal: Find users who talk about 'luxury items'.
- Query: From TWTR, apply UDF-CLASSIFY-LUXURY-SCORE on user tweets. For each user, count the number of tweets about luxury-items. Threshold on count above $\underline{x_1}$.

Analyst 8, version 2 (A_8v_2)

- Analyst goal: Next, identify restaurants those users frequently visit.
- Query: From 4SQ, for each user, count the number of checkins per restaurant. Threshold on count above $\underline{x_2}$.

  Join with users from previous version.

  For each restaurant, count the total number of checkins by all these users. Threshold on count above $\underline{x_3}$.

Analyst 8, version 3 (A_8v_3)

- Analyst goal: Next, find areas that have a high density of these restaurants and identify the distribution of restaurant types in the area.

- Query: For the restaurants from the previous version, apply UDF-GRID-CELL. Count the number of restaurants per grid cell. Threshold on count above $\underline{x_4}$.

  For each grid cell, compute a histogram on the restaurant type and count.

Analyst 8, version 4 (A_8v_4)

- Analyst goal: Finally, match users to grid cells and find luxury hotels in their matching grid cell.

- Query: For each user u from previous version, identify location, and compute a histogram on the restaurant type and u's checkin count.

  Match u to a grid cell such that the grid cell is sufficiently far away from u's location, and there is a significant overlap between u's histogram and grid-cell g histogram from previous version.

  From LAND, find hotels with rating greater than $\underline{x_5}$ stars, and apply UDF-GRID-CELL to convert hotel location to grid cell g'.

  For each user u, join grid cell g with g' to identify hotels in an area matching u's restaurant preferences.

Chapter 7

Conclusion and Future Work

Physical design tuning is key to obtaining good performance for a database management system. Recently, there have been disruptions to the types of data, queries, and systems used for analytical data management and processing. The advent of the cloud and database-as-a-service along with these recent disruptions have changed the analytical landscape significantly. Physical design tuning methods that were developed to address traditional RDBMS architectures are inadequate for the newly emerging system architectures and analytics. While tuning the physical design remains crucial for good performance, the problem acquires new and interesting dimensions in these new contexts.

In this dissertation, we introduced new methods for physical design tuning for emerging system architectures. Our methods exploit the unique characteristics of different architectures, leveraging them toward good physical design. For replicated database architectures, our concept of divergent design presented in Chapter 3 exploited database replication to specialize the design of replicas to efficiently process subsets of the workload while still affording the opportunity to load balance the workload across

replicas. For MapReduce architectures, our concept of opportunistic design presented in Chapter 4 exploited the by-products of query processing in MapReduce through our semantic UDF model and novel query rewriting algorithm to enable the effective reuse of previous results. For hybrid MapReduce–RDBMS architectures, our concept of multistore design presented in Chapter 5 exploited the unique strengths of both stores by periodically reorganizing the data in each store, adapting the physical design as the workload changes dynamically. Lastly, based on our experiences with the changing analytical landscape, in Chapter 6 we examined big data exploratory queries in current real-world scenarios and presented a workload modeled upon our findings. We also presented a set of system performance metrics and a benchmark that helps to highlight the various tradeoffs made by different system architectures, and in better understanding these tradeoffs, the benchmark can be used to help guide the design of future hybrid architectures.

There are many interesting areas to explore for future work. For instance, view maintenance is an important aspect to consider. In our work on opportunistic design, the views are generated opportunistically, the queries are exploratory, and the base data (e.g., logs stored in HDFS) is not updated in place but rather append-only. These each have interesting implications for view maintenance. First, we do not explicitly request to materialize the views in our system and thus they have no creation cost. In contrast, in a typical physical design process, beneficial views are identified and then selected for materialization only after considering their expected benefit as well as their overheads such as creation cost and maintenance cost. Since our views did not

take these overheads into account before their creation but were just produced as by-products of query processing, it is not clear that they should be immediately maintained as they may have little future value. Their benefit values would be computed during a tuning/storage reclamation phase, but since computing the benefits of designs is an expensive process and these phases may not occur after every query (in our multistore work, reorganization phases are less frequent), the question of whether to perform view maintenance in-between tuning phases remains. Moreover, our opportunistic design policy of retaining all by-products can lead to a large number of views in the system, and their combined maintenance cost may needlessly overburden system resources. An interesting idea to consider would be "opportunistically" maintaining views, whereby new query plans could be modified such that they help to maintain existing views at minimal cost. Furthermore, the very process of maintaining the views itself may generate additional views as by-products, leading a rich problem space to explore.

Second, since the queries represent exploratory analysis on big data, some level of view staleness may be expected or tolerated in this analytical space. Additionally, varying levels of staleness may be tolerated for different types of data, in particular since the data may be generated "outside" the business (i.e., by a third-party) and different data sets may have very different update rates which are beyond the control of the business analyzing these data sources. Synchronous and asynchronous policies could be considered, as well as their interplay with the above mentioned cost-benefit considerations. Third, due to the append only nature of HDFS data, this may have interesting implications for view maintenance policies, in particular with opportunistic

views. There has been a long history of research in view maintenance for data warehousing, including view maintenance in big data systems [5] that could be leveraged here.

As noted in Chapters 5 and 6 on multistore design and our benchmark, another area of future work is to consider various tradeoffs that can be made in the cloud in order to minimize monetary cost or minimize overall processing time (i.e., our TTI metric). Note that these are not the same thing, as different types of processing systems and storage have different pricing models. When combining systems into a multistore system, there are many tradeoffs to consider as our metrics in Chapter 6 show. Given the characteristics of systems can be very different, a multistore processing system that is utilized on-demand can be very dynamic in that the scales of each store need not be fixed, but can be continuously adjusted in order to minimize some metric. This elasticity leads to many exciting possibilities for optimization, and could lead to a rich area of future work.

Bibliography

[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, 1998.

[2] A. Abouzeid, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1), 2009.

[4] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *NSDI*, 2012.

[5] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *SIGMOD*, 2009.

[6] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, volume 2000, 2000.

[7] Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/.

[8] Amazon Web Services (EC2, RDS, Redshift). http://aws.amazon.com/.

[9] Apache Hadoop. http://hadoop.apache.org/.

[10] Apache Mahout. http://mahout.apache.org/.

[11] Apache Sqoop. http://sqoop.apache.org/, 2013.

[12] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL server for cloud computing. In *ICDE*, 2011.

[13] D. Bowie. Do you need a data scientist? *CIO*, Sept. 2012.

[14] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD*, 2005.

[15] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[16] N. Bruno and S. Chaudhuri. Constrained physical design tuning. *VLDBJ*, 1(1), August 2008.

[17] E. Brynjolfsson, L. M. Hitt, and H. H. Kim. Strength in numbers: How does data-driven decision making affect firm performance? *SSRN eLibrary*, 2011.

[18] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.

[19] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC*, 1977.

[20] G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, N. Polyzotis, and J. S. V. Varman. The QueRIE system for personalized query recommendations. *IEEE Data Engineering Bulletin*, 34(2), 2011.

[21] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *SSDBM*, 2009.

[22] S. Chaudhuri, E. Christensen, G. Graefe, V. R. Narasayya, and M. J. Zwilling. Self-tuning technology in Microsoft SQL server. *IEEE Data Engineering Bulletin*, 22(2), 1999.

[23] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: a hardness study and a principled heuristic solution. *Knowledge and Data Engineering, IEEE Transactions on*, 16(11), 2004.

[24] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing SQL workloads. In *SIGMOD*, 2002.

[25] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for Microsoft SQL Server. In *VLDB*, volume 97, 1997.

[26] S. Chaudhuri and V. Narasayya. AutoAdmin "what-if" index analysis utility. In *SIGMOD*, 1998.

[27] S. Chaudhuri and V. Narasayya. Index merging. In *ICDE*, 1999.

[28] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. *PVLDB*, 5(12), 2012.

[29] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating MapReduce performance using workload suites. In *MASCOTS*, 2011.

[30] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. *VLDBJ*.

[31] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: new analysis practices for big data. *VLDB*, 2(2), 2009.

[32] M. P. Consens, K. Ioannidou, J. LeFevre, and N. Polyzotis. Divergent physical design tuning for replicated databases. In *SIGMOD*, 2012.

[33] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL tuning in Oracle 10g. In *VLDB*, 2004.

[34] D. Dash, N. Polyzotis, and A. Ailamaki. CoPhy: a scalable, portable, and interactive index advisor for large workloads. *VLDBJ*, 4(6), March 2011.

[35] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[36] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *SIGMOD*, 2013.

[37] I. Elghandour and A. Aboulnaga. ReStore: Reusing results of MapReduce jobs. *PVLDB*, 5(6), February 2012.

[38] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. Iterative parallel data processing with Stratosphere: an inside look. In *SIGMOD*, 2013.

[39] M. Fitzgerald. Tech hotshots: The rise of the IT business analyst. *Computerworld*, July 2012.

[40] M. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *ACM Sigmod Record*, 34(4), 2005.

[41] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, June 2001.

[42] Google Compute Engine. https://cloud.google.com/products/compute-engine.

[43] T. Groenfeldt. Big data knows you, even if you don't know big data. *Forbes*, November 2011.

[44] S. Grumbach and L. Tininini. On the content of materialized aggregate views. In *PODS*, May 2000.

[45] H. Hacigümüs, J. Sankaranarayanan, J. Tatemura, J. LeFevre, and N. Polyzotis. Odyssey: A multi-store system for evolutionary analytics. *PVLDB*, 6(11), 2013.

[46] A. Y. Halevy. Answering queries using views: A survey. *VLDBJ*, 10(4), December 2001.

[47] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library: or MAD skills, the SQL. *PVLDB*, 5(12), 2012.

[48] D. Henschen. IBM beats Oracle, Microsoft with big data leap. *InformationWeek*, October 2011.

[49] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *TODS*, 28(4), December 2003.

[50] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11), July 2012.

[51] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[52] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa. Supporting exploratory queries in databases. In *DASFAA*, 2004.

[53] H. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS*, 1999.

[54] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for a collaborative query management system. In *CIDR*, 2009.

[55] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *VLDBJ*, 4(1), 2010.

[56] N. Khoussainova, Y. Kwon, W.-T. Liao, M. Balazinska, W. Gatterbauer, and D. Suciu. Session-based browsing for more effective query reuse. In *SSDBM*, pages 583–585, 2011.

[57] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. CORADD: Correlation aware database designer for materialized views and indexes. *PVLDB*, 3(1-2), 2010.

[58] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: a graph-based approach. In *SIGMOD*, June 2011.

[59] G. Konstantinidis and J. L. Ambite. Optimizing query rewriting for multiple queries. In *IIWeb*, May 2012.

[60] S. LaValle, E. Lesser, R. Shockley, M.S. Hopkins, and N. Kruschwitz. Big data, analytics and the path from insights to value. *MIT Sloan Management Review*, 52(2), 2011.

[61] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüş, J. Tatemura, and N. Polyzotis. Towards a workload for evolutionary analytics. In *Proceedings of the 2nd ACM SIGMOD Workshop on Data Analytics in the Cloud (DanaC)*, 2013. Extended version *CoRR abs/1304.1838*.

[62] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüş, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: Souping up big data query processing with a multistore system. In *SIGMOD*, 2014.

[63] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüş, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *SIGMOD*, 2014.

[64] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS*, 1995.

[65] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using MapReduce. In *SIGMOD*, 2011.

[66] M. Liihring, K-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. In *ICDE*, 2007.

[67] LinkedIn. http://data.linkedin.com/opensource/datafu.

[68] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1), April 2012.

[69] Microsoft Windows Azure. http://www.windowsazure.com/.

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

[71] MyExperiment. http://myexperiment.org/workflows.

[72] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing across multiple queries in MapReduce. *VLDB*, 3(1-2), September 2010.

[73] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[74] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDE Workshop*, 2007.

[75] H. Park, R. Ikeda, and J. Widom. RAMP: A system for capturing and tracing provenance in MapReduce workflows. *VLDB*, 2011.

[76] D. J. Patil. *Building Data Science Teams*. O'Reilly, 2011.

[77] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[78] PiggyBank. https://wiki.apache.org/pig/PiggyBank.

[79] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB*, 10(2), August 2001.

[80] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *VLDBJ*, 12(2), August 2003.

[81] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads. *PVLDB*, 6(10), 2013.

[82] P. Sarda and J. R. Haritsa. Green query optimization: Taming query optimization overheads through plan recycling. In *VLDB*, 2004.

[83] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE*, 2007.

[84] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: keeping DBAs in the loop. *PVLDB*, 5(5), 2012.

[85] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. *PVLDB*, 2(1), 2009.

[86] T. Sellam and M. Kersten. Meet Charles, big data query advisor. In *CIDR*, 2013.

[87] T.K. Sellis. Multiple-query optimization. *TODS*, 13(1), 1988.

[88] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. QoX-driven ETL design: Reducing the cost of ETL consulting engagements. In *SIGMOD*, June 2009.

[89] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *SIGMOD*, 2012.

[90] J. A. Solworth and C. U. Orji. Distorted mirrors. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. IEEE, 1991.

[91] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *TODS*, 35(1), 2010.

[92] Taverna Workflow Management System. http://www.taverna.org.uk.

[93] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *VLDB*, 2(2), August 2009.

[94] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, 2010.

[95] Transaction Processing Peformance Council. www.tpc.org.

[96] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[97] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, 2011.

[98] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

[99] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and parallel DBMS. In *SIGMOD*, 2010.

[100] Yahoo! Pipes. http://pipes.yahoo.com.

[101] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD*, June 2000.

[102] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, 2004.