Large Scale Data Management for Enterprise Workloads

By

Ashish Tapdiya

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

May 11, 2018

Nashville, Tennessee

Approved:

Daniel Fabbri, Ph.D.

Bradley Malin, Ph.D.

Jules White, Ph.D.

Yuan Xue, Ph.D.

William French, Ph.D.

To My Teachers.

ACKNOWLEDGMENTS

My PhD journey has been a long one and I have been extremely fortunate to have Dr. Daniel Fabbri as my adviser in this journey. I want to thank Dan for his patience, guidance, and the countless hours that he has spent with me to bring this dissertation to its current state. I am amazed by not only Dan's technical expertise, but also his willingness to care for and ensure the overall well-being of his students. I hope some of his enthusiasm and brilliance has rubbed off on to me through this process. I will always be thankful to both Dan and Dr. Vivek Narasayya for providing me an opportunity to intern at Microsoft Research and experience the Microsoft culture.

I am grateful to Dr. Yuan Xue for introducing me to the PhD research and teaching me about how to distill the essence of a technical paper. I would also like to thank Dr. Bradley Malin for his valuable feedback regarding the dissertation document. Thanks to Dr. Jules White and Dr. William French for making time to serve on my committee. I am indebted to Dr. Errin Fulp for introducing me to the world of research.

I was extremely fortunate to have some of the most brilliant and humble people, Jia Bai, Wei Yan, Li Li, Xiaowei Li, Fan Qui, Xujie Si, and Zhijun Yin as my lab mates. I cherish my time spent with them.

Many thanks to my flatmates Saumitra, Milind, and Anirban for making the apartment feel like a home. I appreciate Srivatsan's guidance in helping me settle down during the initial part of my PhD journey.

Thanks to our Nashville family for their friendship and the memorable times they shared with us. We are grateful to Dave and Janice Hoagey, our landlords who always treated us like their kids.

If it was not for these gentlemen, surviving this PhD journey would not have been easy. Thanks to – Siladitya Mukherjee for being my gym buddy and at times a vent for my frustations. Tanmay Misra for increasing productivity by creating clockwork like disturbance

in my life. Parikshit Moitra for numerous visits and hikes to the parks, lakes and rivers. Gaurav Das for being the most easy going person who was always willing to provide company.

I am indebted to my parents for providing unwavering support throughout my life. They taught me to always give my best and to always do the right thing. I want to thank my Grandparents for sharing interesting stories from their lives that provide me with the perspective to see through tough times. I love my nieces and nephew, who provided a regular dose of hilariously innocent conversations. Saving the best for the last, my loving wife Monica Hedda has been a source of continuous support and encouragement for me. Monica's ability to find a silver lining in every situation has helped me stay positive through tough times. Thank you for being my rock. Congratulations to you on earning your Master of Science, I am proud of you. I look forward to globetrotting with you.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES xi

LIST OF FIGURES xiii

1 Introduction 1

1.1 Problem Statement 3

1.1.1 Large Scale Enterprise Data Management for OLTP Workloads 4

1.1.1.1 Online Transaction Processing in Relational Databases 5

1.1.1.2 Online Transaction Processing in Distributed Databases 6

1.1.2 Large Scale Enterprise Data Management for OLAP Workloads 7

1.1.2.1 SQL-on-Hadoop 7

1.1.2.2 SQL-on-Object-Storage 8

1.2 Research Approach and Contributions 9

1.2.1 Putting It All Together 11

2 Background and Related Work 14

2.0.1 Large-Scale Data Processing Frameworks 14

2.0.1.1 HBase and Phoenix 15

2.0.1.2 Amazon S3 18

2.0.1.3 Impala 19

2.0.1.4 Drill 20

2.0.1.5 Spark and Spark SQL 20

2.0.2 Large Scale Enterprise Data Management for OLTP Workloads 22

2.0.2.1 Online Transaction Processing in Relational Databases 22

2.0.2.2 Online Transaction Processing in Distributed Databases 23
2.0.3 Large Scale Enterprise Data Management for OLAP Workloads 27
2.0.3.1 OLAP with Hadoop as Storage Substrate 27
2.0.3.2 OLAP with Object Storage Systems as Storage Substrate 28
3 Performance Variations in Profiling MySQL Server on the Xen Platform: Is it Xen or MySQL? 30
3.1 Experiment Methodology 31
3.1.1 Request Processing Overview 32
3.1.2 Database Server Specifics 34
3.1.3 Xen Hypervisor and Virtual Machine Configuration 34
3.1.4 Client Workload Generator 35
3.1.5 Benchmark SQL Statements 35
3.1.5.1 Simple Query 36
3.1.5.2 Complex Query 37
3.1.6 Performance Metric 38
3.1.7 Experiment Setup 38
3.2 Performance Evaluation of Benchmark Read Statement 39
3.2.0.1 Dom-u Performance Discussion 39
3.2.0.2 Dom-0 Performance Discussion 41
3.2.1 Impact of network I/O on performance variations 41
3.2.2 Impact of disk I/O on performance variations 44
3.2.3 Impact of database population from snapshot on performance variations 46
3.3 Amazon EC2 Experiments 49
3.3.1 Database population from the script 50
3.3.2 Database population from the snapshot 50
3.4 Performance evaluation of benchmark simple query 51
3.5 Performance evaluation of benchmark write statement 51

3.6 Chapter Summary 52
4 A Comparative Analysis of Materialized Views Selection and Concurrency Control Mechanisms in NoSQL Databases 53
4.1 Background 54
4.1.1 Relation, Index and Schema Models 55
4.1.2 Modeling Workload 55
4.1.3 Baseline Database Transformation 55
4.2 Challenges and Design Choices 56
4.2.0.1 Implication of Materialized Views 56
4.2.0.2 Lock Number and Granularity 57
4.2.0.3 View Selection Challenges 57
4.2.1 Design Decisions 58
4.3 System Overview 58
4.4 Generating Candidate Views 59
4.4.1 Roots Selection 61
4.4.2 Candidate Views Generation Mechanism 61
4.4.2.1 Mechanism Overview 62
4.4.2.2 Mechanism Description 63
4.4.2.3 Discussion 65
4.5 Views Selection Mechanism 65
4.5.1 Views Selection 66
4.5.2 Query Re-writing 68
4.5.3 Additional View Indexes 68
4.6 View Maintenance Mechanism 68
4.6.1 Insert Statement 69
4.6.1.1 Applicability Test 69
4.6.1.2 Tuple Construction 69

4.6.2 Delete Statement 69
4.6.2.1 Applicability Test 69
4.6.2.2 Key Construction 69
4.6.3 Update Statement 70
4.6.3.1 Applicability Test 70
4.6.3.2 Tuple Construction 70
4.7 System Architecture 70
4.7.1 Lock Implementation 71
4.7.2 Write Transaction Procedures 72
4.7.3 Transaction Isolation Level 73
4.8 Experimental Evaluation 73
4.8.1 Experiment Environment 74
4.8.1.1 Testbed 74
4.8.1.2 Performance Metric 74
4.8.2 Micro Benchmark Evaluation 75
4.8.2.1 Schema and Workload 75
4.8.2.2 Experiment Setup and Results 76
4.8.3 Locking Overhead Evaluation 77
4.8.4 TPC-W Benchmark Evaluation 78
4.8.4.1 Benchmark 78
4.8.4.2 Systems Evaluated 79
4.8.4.3 Performance Evaluation of Joins in the TPC-W Benchmark . . 80
4.8.4.4 Performance Evaluation of Write Statements in the TPC-W Benchmark 81
4.8.4.5 Performance Comparison of All Evaluated Systems 82
4.9 Chapter Summary 84

5 A comparative analysis of state-of-the-art SQL-on-Hadoop systems for interactive analytics . . . 85
5.1 Background . . . 87
5.2 Experiment Goals . . . 88
5.3 Micro Benchmark Experiments . . . 89
5.3.1 Hardware Configuration . . . 89
5.3.2 Software Configuration . . . 90
5.3.3 Experiment Setup . . . 90
5.3.4 WDA Benchmark . . . 91
5.3.5 Data Preparation . . . 91
5.3.6 Experiment Results . . . 94
5.3.6.1 Selection Task (Q1) . . . 95
5.3.6.2 Aggregation Task (Q2) . . . 95
5.3.6.3 Join Task (Q3) . . . 97
5.4 TPC-H Benchmark Experiments . . . 98
5.4.1 Hardware and Software Configuration . . . 98
5.4.2 Data Preparation . . . 99
5.4.3 Experiment Results . . . 99
5.4.3.1 Execution Time Breakdown . . . 102
5.4.3.2 Correlated Sub-query Execution in Drill . . . 103
5.4.3.3 Parquet versus Text Performance . . . 106
5.4.3.4 Size-up Characteristic Evaluation . . . 108
5.5 Discussion and Chapter Summary . . . 108

6 Fusion: Implementation and evaluation of block range indexes in a SQL-on-Object-Storage system . . . 111
6.1 Block Range Index . . . 113
6.1.1 BRIN Implementation and Storage Choices . . . 114
6.1.1.1 Key Value Store Based Implementation . . . 114
6.1.1.2 Interval Tree Based Implementation . . . 116
6.2 System Overview . . . 116
6.3 Experimental Evaluation . . . 118
6.3.1 Hardware Configuration . . . 118
6.3.2 Software Configuration . . . 119
6.3.3 Experiment Setup . . . 119
6.3.4 TPC-W Benchmark . . . 120
6.3.5 Experiment Results . . . 120
6.3.5.1 Fusion versus Baseline . . . 120
6.3.5.2 Impact of Data Partition Size in Fusion System . . . 122
6.3.5.3 Fusion versus Baseline: Impact of Database Size . . . 123
6.3.5.4 Write Statement Performance . . . 124
6.4 Chapter Summary . . . 125
7 Future Work . . . 126
7.0.1 Mechanisms . . . 126
7.0.2 Experiments . . . 127
8 Conclusion . . . 129
8.1 Summary of Contributions . . . 129
BIBLIOGRAPHY . . . 132

LIST OF TABLES

Table Page

2.1 Summary and classification of related works. 26

3.1 Notations commonly used throughout this chapter. 36

3.2 Query μ and μ_{se} with dom-u inside Nodes B, C and D 43

4.1 Qualitative comparison of NoSQL, NewSQL and Synergy systems. 54

4.2 Sum of RT of all the statements in the TPC-W benchmark to quantify trade off between read performance gain and write performance overhead of using MVs in each evaluated system. VoltDB is excluded since it does not support all queries in the benchmark. 82

4.3 Database sizes across different evaluated systems. 82

4.4 Specification of joins in the TPC-W Benchmark. 83

4.5 Specification of write statements in TPC-W Benchmark. 83

5.1 Qualitative comparison of evaluated SQL-on-Hadoop systems. 86

5.2 Data preparation times (seconds) in evaluated systems for WDA benchmark. 91

5.3 Query RTs (in seconds) in evaluated systems using WDA benchmark for 2, 4 and 8 worker nodes in the cluster. RT in bold text denotes the fastest system for each query and cluster size combination. AM denotes the arithmetic mean. To compute the normalized AM: for each query, we normalize the query RTs in each system and for each cluster size by the query RT in Impala with 2 worker nodes. 93

5.4 Data preparation times (seconds) in systems for the TPC-H benchmark . . . 98

5.5 Query RTs (seconds) in evaluated systems using the TPC-H benchmark at 125, 250 and 500 scale factors. RT in bold text denotes the fastest system for each query, scale factor and file format combination. To compute the normalized AM–Q{2,11,13,16,19,21,22}: for each query, we normalize the query RTs in each system, at all scale factors and for each storage format by the query RT in Impala, for the parquet storage format, at scale factor 125. . . 100
5.6 Size-up property evaluation in each system. SF denotes scale factor. 107

6.1 Example Block range index for *id* column of Customers table. 113
6.2 Total time to compute index intervals for 10GB TPC-W database with 16MB table partitions. 121
6.3 Total time to create indexes in the Fusion system. 121
6.4 Fusion versus Baseline: Performance of TPC-W queries that could use block range indexes. 122
6.5 Fusion versus Baseline: Overall performance of TPC-W benchmark queries. 122
6.6 Index intervals computation in Fusion system for 10GB TPC-W database with 4MB and 16MB partitions. 122
6.7 Total query execution time in Fusion system for 10GB TPC-W database with 4MB and 16MB partitions. 123
6.8 Fusion versus Baseline: Overall performance of TPC-W benchmark queries for 10GB and 50GB TPC-W databases. 123
6.9 Write statement performance in Fusion system for 4MB and 16MB chunks. . 124

LIST OF FIGURES

Figure Page

1.1 Large Scale Data Management Frameworks Ecosystem. 3

1.2 Summary of both challenges faced by data architects in large scale data management of each enterprise workload class and the mechanisms proposed in this dissertation to address these challenges. 11

2.1 HBase architecture overview 15

2.2 An example table in HBase. 16

2.3 Physical view of HBase table in Figure 2.2. 16

2.4 Single Table Aggregation Query Processing in Phoenix 17

2.5 Join query processing in Phoenix 18

2.6 Impala architecture overview 18

3.1 Laboratory test bed 32

3.2 Request Processing Overview With MySQL Server Hosted Inside Xen Domains . 33

3.3 Simple Query 36

3.4 Complex Query 37

3.5 Write Statement 38

3.6 Profiling model 39

3.7 Read statement performance comparison between dom-0 and dom-u VMs for $\gamma =$ 0 and $S = S_{default}$ 40

3.8 Measuring query execution time in a client session 42

3.9 Read statement performance comparison between dom-u's for different cache allocations. 43

3.10 Percentage reduction in τ for different values of γ relative to τ for $\gamma = 0$ on Node B's dom-u 44

3.11 Read statement performance comparison between dom-0 and dom-u VMs for database populated using *mysqldump* snapshot 46
3.12 Read statement performance comparison between database populated from script and snapshot on Amazon EC2 cloud platform 47
3.13 Simple query performance comparison between dom-0 and dom-u VMs with $\gamma =$ 0 and $S = S_{default}$ 48
3.14 Write statement performance comparison between dom-0 and dom-u VMs for $\gamma =$ 0 and $S = S_{default}$ 49

4.1 Relations in the Company Schema. 54
4.2 Database transformation workflow in Synergy system. 58
4.3 Input and output of the candidate views generation mechanism for the Company database with roots set $Q_{company} = \{Address,\ Department\}$. 61
4.4 Intermediate results of the candidate views generation mechanism for the Company database with roots set $Q_{company} = \{Address, Department\}$. Relations: Department (D), Department_Location (DL), Employee (E), Works_On (WO), Project (P), Dependent (DP), Address (A) 62
4.5 Illustration of view selection and query re-writing procedure for an example equi join query using an example rooted tree. 66
4.6 Synergy System Architecture Overview. 70
4.7 Micro benchmark schema graph. 74
4.8 Micro-Benchmark Workload. 75
4.9 Micro benchmark results to show that performance of join algorithms is slow in HBase. Y axis is drawn at log scale. 76
4.10 Experiment to show overhead associated with two phase row locking in HBase. 77
4.11 Materialized views selection mechanism and concurrency control mechanism used in each evaluated system. 78

4.12 Evaluation and comparison of join performance across different systems using join queries in the TPC-W benchmark. Y axis is drawn at log scale. Join queries $\{Q_3, Q_7, Q_9, Q_{10}\}$ are not supported in VoltDB. 80
4.13 Performance Evaluation of the write statements in the TPC-W benchmark to exhibit the overhead of lock management and updating views in the Synergy system. Comparison of write statement performance across different systems. 81

5.1 Query execution model in each evaluated system. 88
5.2 Query text and logical plans for the WDA benchmark queries. 94
5.3 The breakdown of query RT into aggregated processing time for each operator type in evaluated systems. The TPC-H scale factor is 500 and the storage format is parquet. For each query, the left bar, the middle bar and the right bar represent Impala, Spark SQL and Drill systems, respectively. 101
5.4 Query text and execution plans with profiled operator times in evaluated system for TPC-H query 4. The storage format is parquet and the SF is 500 104
5.5 The breakdown of query RT into aggregated processing time for each operator type in evaluated systems. The TPC-H scale factor is 500 and the storage format is text. For each query, the left bar, the middle bar and the right bar represent Impala, Spark SQL and Drill systems, respectively. 106

6.1 Range query sub cases in a block range index. 115
6.2 Fusion system overview. 117
6.3 Query processing in Fusion system. 118

Chapter 1

Introduction

Continual proliferation of mobile devices, sensors, gaming consoles, and social media combined with the ever-increasing online and mobile user population has resulted in a data deluge. Web 2.0 applications enable users to generate content by sharing media (images, videos, news, etc.), placing orders online, and interacting through social media dialogue. For example, Facebook has a global user base of more than 1.4 billion users and handles more than 4 million posts per minute [1]. In addition to user generated data, server farms, financial systems, telecom network equipments, and sensors generate continuous streams of machine data (event log messages, trades, CDRs, resource utilizations, etc.). As an example, NaviSite processes 5-10K server event log messages generated per second by a server farm of thousands of servers [2].

Traditionally, relational databases are used as the backend for web and mobile application development. However, as an application becomes popular and its user base widens, the database system must scale up to handle the increasing workload. While relational databases are well suited for vertical scaling, they require specialized hardware that can be expensive. The need to process high volumes of concurrent user transactions and store huge amounts of data in a scalable and cost-efficient manner has led to the development of several NoSQL [3, 4, 5, 6, 7] and NewSQL [8, 9] distributed databases. NoSQL databases trade high transaction latency for horizontal scalability, flexible data modeling, and automated data partitioning as compared to relational databases. In contrast, NewSQL databases trade limited query expressiveness for high transaction throughput and horizontal scalability.

Storing and analyzing large amounts of data in a timely and cost efficient manner enables organizations to discover patterns, build analytical models to streamline business processes, identify customer sentiment to drive targeted marketing campaigns, etc. Enterprises

traditionally use parallel DBMSs (row-oriented/column-oriented) for large scale data analytics. Column-oriented systems [10, 11, 12] generally outperform row-oriented systems [13, 14, 15] for read heavy analytical workloads by trading write performance degradation for increased read performance. However, in case of a mid-query failure, parallel databases lose all the work and need to re-run the query from beginning.

To overcome the limitation of complete-restart on failure, several new large scale data analytics frameworks [16, 17, 18, 19, 20] have been proposed recently. Hadoop [21] is a distributed file system (HDFS) modeled after the GFS design [22]. MapReduce [16] (MR) is a fault tolerant batch-oriented data processing framework built on top of HDFS for processing large data sets in-parallel on clusters of commodity hardware. Spark [19] is a cluster computing framework that gains significant efficiency over MR framework by pipelining data between operations and keeping intermediate results in memory.

The developer familiarity and the query expressiveness of SQL has led to the emergence of several SQL-on-Hadoop query execution engines. Hive [17] enables users to write queries in HiveQL (SQL-like) language that can be compiled into a directed acyclical graph of MR/Tez/Spark jobs and run on the respective runtime. Spark SQL [20] proposes a DataFrames API for relational query processing on top of Spark execution engine. Although Hive and Spark-SQL enable users to write SQL queries to get answers from the data, the query response times exceed expected latency for interactive analytics. To this end, Impala [23] and Drill [24] have been proposed recently. Impala is a massively parallel processing (MPP) query execution engine built on top of HDFS. Similar to Impala, Drill is an MPP query execution engine that leverages vectorization and processes columnar data in memory with lazy materialization. Impala and Drill trade fault intolerance for increased performance as compared to Hive and Spark-SQL.

In addition to Hadoop, distributed object storage systems like Amazon S3 [25] and Microsoft Azure Blob storage [26] have been proposed recently. This has led to both, emergence of new SQL-on-Object-Storage systems like Snowflake [27] and implementa-

tion of I/O subsystems in existing query engines like Impala [23], Spark SQL [20], etc., to process the data stored in object stores like S3.

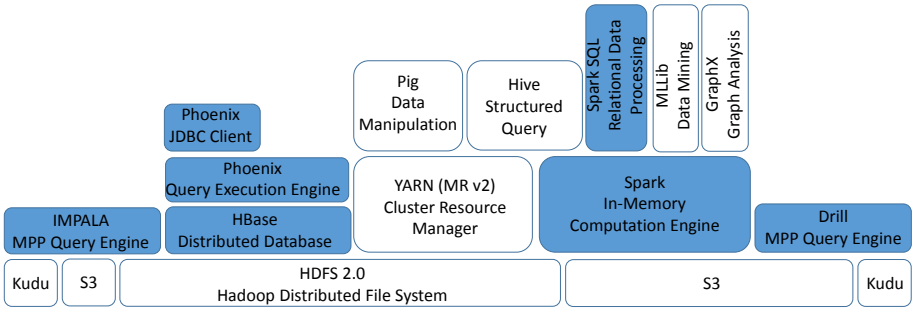


Figure 1.1: Large Scale Data Management Frameworks Ecosystem.

Concomitant with the development of big data systems has been the emergence of cloud computing [28]. Cloud platforms enable networked access to a shared pool of computing resources (compute, storage, network, etc.). Users can dynamically provision and release resources in response to the demand. The pay-as-you-go metering service charges users based on the lease duration of each resource used. Cloud providers harness commodity hardware and virtualization to build a shared platform. Cloud computing synergizes large scale data management by enabling users to dynamically create and scale a cluster for recently proposed horizontally scalable databases and data processing frameworks.

1.1 Problem Statement

Enterprises model each business process as a workload. Enterprise workloads are generally classified into online transaction processing (OLTP) and online analytical processing (OLAP). Each workload type is characterized by the mix of read/write requests, query complexity, request rate and service level objectives. The OLTP workloads are generally characterized by a large volume of highly concurrent transaction requests and a low expected response latency (e.g. order processing). In contrast, the OLAP workloads generally involve a low volume of complex queries for creating business intelligence (e.g. future sales prediction). Although, several large scale data management systems have been pro-

posed recently for each workload type, data architects still face various challenges in both 1) scaling the proposed systems for enterprise workloads, and 2) choosing the right system for their workloads. First, the proposed systems make different design decisions regarding data partitioning, storage format, query processing, etc., resulting in query performance trade offs. Second, systems trade query expressiveness for performance and vice-versa, resulting in limited practical utility. Third, novel cloud based storage architectures trade performance for cost; this begets the need to develop new mechanisms that augment the performance of the system in order to get practical use out of it. Hence, for each enterprise application, system architects must design or choose a data management solution that can meet the service level objectives of the application. We call this the "large scale enterprise data management" problem.

In this dissertation, we solve this problem for each enterprise workload type by studying the design choices and the query performance trade offs across the proposed systems. For *OLTP* workloads, we study the query performance, query expressiveness, scalability and disk utilization trade offs across relational, NoSQL and NewSQL databases to develop mechanisms for scalable transaction processing. For *OLAP* workloads, we evaluate four state-of-the-art SQL-on-Hadoop systems using standard analytical processing benchmarks to understand the characteristics of two primary components of a SQL-on-Hadoop system (query optimizer and query execution engine) and their impact on the query performance. In addition, we study different block range index implementations to evaluate and understand their impact on the performance of SQL-on-Object-Storage systems.

1.1.1 Large Scale Enterprise Data Management for OLTP Workloads

Relational and distributed databases represent two different architectures for OLTP. Relational databases are well suited to vertical scaling whereas distributed databases are designed to scale out. Our goal is to evaluate the query performance, query expressiveness, scalability and disk utilization trade offs across both architectures using standard OLTP

benchmarks and develop mechanisms for scalable transaction processing.

1.1.1.1 Online Transaction Processing in Relational Databases

We study the performance of relational databases using a standard OLTP benchmark. Our goal is to develop mechanisms for efficient resource management on the cloud hosting platforms by creating robust performance profiles.

With the ever increasing popularity of cloud hosting platforms, more and more enterprises are moving to the cloud to reduce infrastructure and operational costs. Traditionally, relational databases represent the primary choice as the storage tier of a web application. Cloud service providers lease database appliance VMs (e.g. Amazon RDS [29]) to the users and provide Service Level Agreement (SLA) guarantees. To manage the SLA, service providers rely on resource management mechanisms. At the center of a resource management mechanism lies a performance model. A model predicts system performance by taking a set of input parameters that capture the expected workload, as well as the allocated resource. SLA is specified in accordance with the performance predicted by the model.

Performance profiling is an established method for building the performance model. Consistency and repeatability of the profiled results is quintessential to the utility of the constructed performance model. In contrast with the dedicated hosting, shared access to the compute resources, hardware heterogeneity and indirect I/O in cloud hosting platforms raises novel challenges in creation of a repeatable and consistent performance profile.

- Guest VMs utilize interfaces in host VM to perform I/O operations. In addition, Logical Volume Manager (LVM) is used to virtualize disk storage across hosted VMs. These two indirections can result in significant performance variations for disk I/O heavy workloads.

- Contention for the shared I/O resource between guest VMs hosted on the same physical machine can cause temporal performance variations, rendering the profile created during one time interval to be useless in another.

1.1.1.2 Online Transaction Processing in Distributed Databases

Our goal is to design a scalable data store with a standard SQL interface that enables users to express common queries occurring in OLTP workloads while ensuring high performance and ACID transaction semantics.

NewSQL architectures enable a database to scale out linearly while providing ACID transaction guarantees. However, their schema design requires careful consideration when choosing partition keys, since joins are restricted to partition keys only, resulting in limited query expressiveness. Similarly, NoSQL databases can also scale out linearly, but are limited to single key atomic operations and expose a simple data manipulation API (i.e., get, put, delete, and scan operations). A SQL skin (e.g. Apache Phoenix) on top of a NoSQL database could be utilized to perform a Strawman transformation of a relational database to a NoSQL database; however, such a Strawman transformation is limited by join performance due to the distribution of data across the cluster and data transfer latency.

Materialized views represent a standard method to improve join performance through pre-computation. However, deploying materialized views on top of a NoSQL store, while ensuring ACID semantics, presents many challenges:

- NoSQL databases are limited to atomic key-based operations (i.e. multiple rows cannot be updated atomically in a single operation). Hence, additional concurrency controls are required to ensure data consistency between materialized views and base tables.

- Additional care must be taken when selecting views to materialize, due to the distribution of data in a NoSQL database, to avoid high view maintenance and storage costs. For example, materializing a many-to-many join may lead to large view maintenance costs, because an update to a single base table row may propagate view updates across the cluster.

1.1.2 Large Scale Enterprise Data Management for OLAP Workloads

SQL-on-Hadoop and SQL-on-Object-Storage denote two different architectures for online analytical processing. SQL-on-Hadoop represents a shared nothing architecture where as SQL-on-Object-Storage represents a decoupled architecture in which compute and storage can scale independently. For SQL-on-Hadoop systems, our goal is to understand the impact of file formats, size-up, and scale-up characteristics in each evaluated system. For SQL-on-Object-Storage systems, our goal is to empirically evaluate the impact of block range indexes on the query performance.

1.1.2.1 SQL-on-Hadoop

We study and compare four SQL-on-Hadoop systems (Impala, Drill, Spark SQL and Phoenix) for interactive analytics using standard benchmarks. Our goal is to evaluate and understand the characteristics of two primary components of a SQL-on-Hadoop system (query optimizer and query execution engine) and their impact on the query performance.

Hadoop has emerged as the central data repository in enterprises. Enterprises rely on analytical data processing engines to gain actionable insights from the data. The reliance of several enterprise tools on SQL along with the developer familiarity has led to the development of numerous SQL-on-Hadoop systems (Impala, Spark SQL, etc.) for analytical data processing. SQL-on-Hadoop systems take various forms. One class of system relies on a batch processing runtime for query execution. Systems like Shark [30] and Spark SQL [20] employ the Spark runtime to execute queries specified using the standard SQL syntax. Another class of SQL-on-Hadoop system is inspired by Google's Dremel [31], and leverages a massively parallel processing (MPP) database architecture. Systems like Impala [23] and Drill [32] avoid overhead associated with launching jobs for each query by utilizing long running daemons. However, even within this class, differing design decisions, such as early versus late materialization, impact query performance.

We chose to study these systems due to a variety of reasons: 1) Each evaluated system

has a large user base as they are part of major Hadoop distributions including Cloudera, MapR, Hortonworks, etc. 2) Each system is open source, targets the same class of interactive analytics, and is optimized for the same storage substrate. 3) Each system employs cost based optimization and advanced run time code generation techniques. 4) These systems have significantly different architectures (e.g. batch processing versus long running daemons) and make varying design decisions for query processing (e.g. vectorized [33] versus volcano model [34]).

1.1.2.2 SQL-on-Object-Storage

We use block range index to improve the read performance of a SQL-on-Object-Storage system by skipping irrelevant data. Our goal is to study different block range index implementations and understand their impact on the performance of SQL-on-Object-Storage systems.

A cloud deployed Hadoop cluster can either utilize instance storage or network attached block storage (e.g. Amazon EBS). Instance storage is ephemeral where as network attached block storage can be expensive for big and growing data. In contrast, object storage systems (e.g. Amazon S3) are cheaper than the network attached block storage and provide better performance per dollar [35]. However, despite object storage system's better performance per dollar as compared to the network attached block storage, object storage may not be suitable for interactive analytics due to the high access latency and low read throughput. Hence, additional mechanisms are required to improve the read performance of SQL-on-Object-Storage systems.

Read performance of a query engine can be improved by increasing data read throughput and skipping irrelevant data. Data read throughput can be increased by employing caching, parallelizing reads, and storing data in a compressed format. On the other hand, columnar storage and indexing structures are generally used to skip reading the irrelevant data. In this work, we focus on exploring the use of indexing structures that can effec-

tively skip irrelevant data stored in the object storage systems like S3. Although B+-tree and R-tree based indexes have been shown to be very effective in traditional data warehousing systems, high maintenance overhead and poor random I/O performance of object storage systems like S3, limits their utility in object stores. In contrast, Block Range Indexes (BRIN) are a natural fit for skipping irrelevant data in object stores. They exhibit low maintenance overhead and their size is a fraction of the table size. However, BRIN can be implemented in a variety of ways in the cloud environment. Hence, it is interesting to both evaluate the impact of BRIN on the performance of a SQL-on-Object-Storage system and understand performance characteristics of different block range index implementations.

1.2 Research Approach and Contributions

The high level contributions of this dissertation are as follows. For OLTP workloads, we have developed mechanisms for scalable transaction processing in relational and distributed databases. For relational databases, we present a simple mechanism to stabilize the performance of cloud hosted databases. For distributed databases, we present the design of a novel data store that utilizes materialized views and a specialized concurrency control mechanism to improve read performance without significantly degrading write performance. For OLAP workloads, we empirically evaluate SQL-on-Hadoop and SQL-on-Object-Storage systems and illustrate their performance characteristics. For SQL-on-Hadoop systems, we demonstrate the size-up behavior, scale-up behavior, optimizer attributes, execution engine efficiency and impact of file formats. For SQL-on-Object-Storage systems, we empirically evaluate the performance impact of block range index implementations.

The detailed contributions of this dissertation are as follows:

- In Chapter 3, we study the performance of MySQL server on the Xen platform using an OLTP benchmark. We observe significant performance variations across repeated runs in spite of contention free hosting of a single guest VM on a physical machine.

Also, notable performance difference between guest VMs created equal on physical machines of a homogeneous cluster is noticed. We present and evaluate a black box approach based on database population from a snapshot to reduce perceived performance variations and create a consistent and repeatable performance profile.

- In Chapter 4, we present the Synergy system that leverages MVs and a light-weight concurrency control on top of a NoSQL database to provide for scalable data management with familiar relational conventions and more robust query expressiveness. Synergy harnesses databases' hierarchical schemas to generate candidate MVs, and then uses a workload driven selection mechanism to select views for materialization. To provide ACID semantics in the presence of views, the system implements concurrency controls on top of the NoSQL database using a hierarchical locking mechanism that only requires a single lock to be held per transaction. The Synergy system provides ACID semantics with the read-committed transaction isolation level.

- In Chapter 5, we evaluate and understand the characteristics of two primary components of a SQL-on-Hadoop system (query optimizer and query execution engine) and their impact on the query performance. For the query optimizer, we characterize the execution plan generation, the join order selection and the operator selection in each evaluated system. For the query execution engine, we evaluate the efficiency of operator implementations in each system and identify the performance bottlenecks. We examine the impact of text (row-wise) and parquet (columnar) storage formats on the query performance in each system. To understand the *size-up* characteristic in each system, we increase the data size in a cluster and examine the query performance changes. We evaluate the *scale-up* behavior in each system by proportionally increasing both the cluster and the data size. The results from this study can be utilized in two ways: (i) to assist practitioners choose a SQL-on-Hadoop system based on their workloads and SLA requirements, and (ii) to provide data architects insight

into performance impacts of evaluated SQL-on-Hadoop systems.

- In Chapter 6, we evaluate the performance of SQL-on-Object-Storage systems for interactive data analytics. To improve the baseline performance, we empirically evaluate the utility of block range indexing structures. We harness interval trees and key value stores as two different mechanisms to implement the block range indexes. In addition, we empirically evaluate different block range index implementations to understand their performance trade offs.

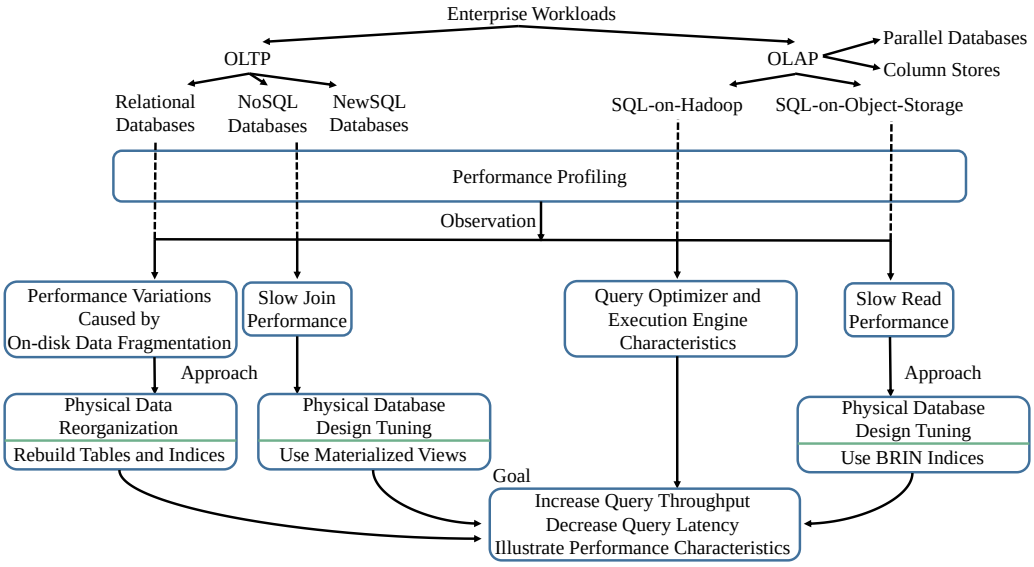


Figure 1.2: Summary of both challenges faced by data architects in large scale data management of each enterprise workload class and the mechanisms proposed in this dissertation to address these challenges.

1.2.1 Putting It All Together

An enterprise workload is characterized by the mix of read and write statements, query complexity, etc. OLTP and OLAP represent two different types of enterprise workloads. Traditionally, a relational database management system (RDBMS) is used to manage the data for OLTP workloads. However, RDBMSs are hard to scale horizontally. To overcome this limitation, NoSQL (e.g. HBase) and NewSQL (e.g. VoltDB) data stores have been

proposed recently. Parallel database management systems (PDBMSs) and column stores (e.g. Vertica) represent conventional choices for OLAP. For scalable storage of big data, Distributed file systems (e.g. HDFS) and Object storage systems (e.g. Amazon S3) have been proposed recently. User familiarity with SQL for data analytics has led to the development of SQL engines (e.g. Impala, Drill, Spark SQL, etc.) that can query the data stored in scalable storage engines, such as S3 and HDFS. SQL query engines like Impala, Drill, etc., harness the architecture and query processing techniques that are developed for both PDBMSs and column stores.

In this work, our goal is to enable scalable data management for each enterprise workload class. Next we look at the general steps taken in this dissertation to address the challenges faced in achieving this goal. For each workload type (e.g. OLTP) and database system type (e.g. NoSQL) combination, we first profile the performance of proposed system(s) (e.g. HBase) using standard benchmarks (e.g. TPC-W). Then, we study the profiled information to identify the performance bottlenecks. Next, we survey the traditional methods that can be used to overcome these bottlenecks and improve the system performance. Finally, we adapt the traditional methods for use with the proposed system(s) by developing novel mechanisms. The proposed mechanisms result in an increase in query throughput and a reduction in query latency. Figure 1.2, summarizes the challenges faced in large scale data management of each enterprise workload class and the mechanisms proposed in this dissertation to address these challenges.

We profile the performance of MySQL RDBMS inside a cloud VM instance and observe significant variations caused by the on-disk data fragmentation. To reduce variations and stabilize performance, we perform physical data re-organization by rebuilding tables and indices. We evaluate the performance of NoSQL databases and observe that the joins are slow. To improve the join performance, we leverage a physical database design tuning technique based on the materialized views. For OLAP workloads, we evaluate and illustrate the performance characteristics of four state-of-the-art SQL-on-Hadoop systems. The per-

formance evaluation of SQL-on-Object-Storage systems shows that the reads are slow due to the high data transfer latency. To this end, we use BRIN indices as a physical database design tuning mechanism to enable data skipping and improve the read performance.

The remainder of this dissertation is organized as follows. Chapter 2 surveys related works and provides an overview of the large scale data processing systems used in this dissertation. We profile the performance of MySQL server on Xen platform for OLTP workloads in Chapter 3. In Chapter 4, we present the design of Synergy system that enables scalable data processing with relational conventions on top of a NoSQL data store. We perform a comparative analysis of four state-of-the-art SQL-on-Hadoop systems using standard analytical processing benchmarks in Chapter 5. In Chapter 6, we evaluate the impact of BRIN on the performance of SQL-on-Object-Storage systems. We present our future work in Chapter 7. Finally, Chapter 8 concludes this dissertation.

Chapter 2

Background and Related Work

In this chapter we first provide an architectural overview of the different large-scale data processing frameworks that we examine and utilize in this thesis. Next, we review works related to the performance evaluation of applications on the virtualized cloud platforms, online transaction processing in distributed databases, performance comparison and benchmarking of different SQL-on-Hadoop systems for OLAP workloads and query processing in the cloud with an object storage system as the storage substrate.

2.0.1 Large-Scale Data Processing Frameworks

Web 2.0 applications are characterized by user generated content and the interaction among users. Users place orders, share opinion, build social connections and interact through social media dialogue. The growth in user population combined with the proliferation of mobile devices result in an increased number of transactions and an unprecedented growth of the data footprint. The need to support a large number of concurrent transactions and store huge amounts of data in a scalable manner has led to the development of several horizontally scalable, shared nothing distributed databases (Bigtable [3], HBase [4], Cassandra [6], MongoDB [5], H-Store [8], Spanner [9], etc.). In addition, enterprises rely on analysis of data to identify valuable patterns and propose new products to the customers. The quality of extracted patterns is directly dependent on the amount of data analyzed; however both storage and analysis of high volumes of data in a cost efficient and timely manner raises novel challenges. To address these challenges, several large scale analytics frameworks have been proposed recently (MapReduce [16], Pig [18], Hive [17], Impala [23], Kudu [36] etc.). Next, we introduce the distributed database (HBase), object storage

system (S3) and the large scale analytics frameworks (Impala, Drill, and Spark) utilized in this thesis.

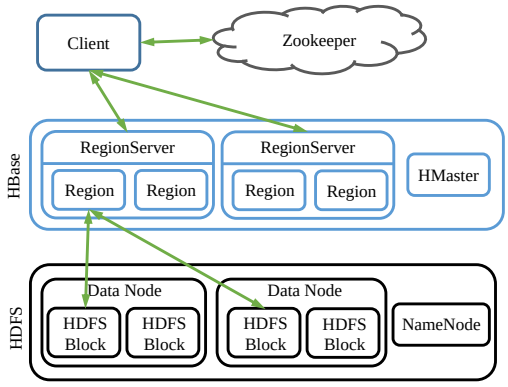


Figure 2.1: HBase architecture overview

2.0.1.1 HBase and Phoenix

HBase is a column family-oriented distributed database modeled after Google's Bigtable. A distributed HBase cluster comprises of different components including Region servers, HBase master, Zookeeper ensemble and Hadoop Distributed File System (HDFS), as depicted in Figure 2.1. An HBase table is split into smaller chunks called regions, that are distributed across the cluster. A region server is responsible for hosting and serving regions. A region server exposes an interface for data manipulation and region maintenance. The HBase master monitors the region servers, manages region distribution among the region servers and exposes an interface for all metadata changes. Zookeeper is a highly available distributed coordination service that is integral to the proper functioning of both the HBase client and the HBase cluster. HBase master and region servers rely on zookeeper ensemble to manage cluster membership. HBase clients query zookeeper to identify the region server hosting the root region. HBase harnesses HDFS as the persistent storage layer; hence, data blocks of a region hosted by a region server may be distributed across different cluster nodes. HBase achieves durability & high availability by persisting its write ahead log (WAL) in the HDFS and utilizing HDFS supported data replication.

Row Key	Column Family Personal		Column Family Home	
Id	Name	Email	Phone	} Column Qualifier
1	Alice	alice@gmail.com	1234	

Figure 2.2: An example table in HBase.

"1",	"Personal",	"Name",	1234456778901,	"Alice"
"1",	"Personal",	"Email",	1234456778900,	"alice@gmail.com"

(a) HFile for '**Personal**' column family

"1",	"Home",	"Phone",	1454456778901,	"1234"

(b) HFile for '**Home**' column family

Figure 2.3: Physical view of HBase table in Figure 2.2.

HBase organizes data into tables. A table consists of rows that are sorted alphabetically by the row key. HBase groups columns in a table into column families such that each column family data is stored in its own file. A column is identified by a column qualifier. Also, a column can have multiple versions of a data sorted by the timestamp. Figure 2.3 depicts the physical storage view of the data in HBase corresponding to the table shown in Figure 2.2.

The HBase data manipulation API comprises of five primitive operations: Get, Put, Scan, Delete and Increment. The Get, Put, Delete and Increment operations operate on a single row specified by the row key. HBase provides ACID transaction semantics for row-level operations. However, scan operations do not exhibit snapshot isolation. A scanner only ensures that it returns a consistent and complete version of a row that existed when it read it.

HBase filters enable users to harness server side compute power in reducing the data returned to the client. Filters can be combined with the scanner to push data filtering criteria down to the server. Coprocessors extend HBase from a scalable data store to a distributed storage and computational platform. HBase coprocessors are classified into observers and endpoints. An observer is similar to a trigger in relational database management systems

(RDBMSs). It acts as a proxy between the client & the region server that can be invoked after every get and put operation to modify the data access. Endpoints are analogous to the stored procedures in RDBMSs. Endpoints enable users to push an arbitrary computational task to the region servers.

Phoenix [37] is a client side relational database layer on top of HBase. It compiles a SQL query into a series of HBase scans and coordinates the execution of scans to generate a standard JDBC result set. Phoenix gathers a set of keys per region that are at equal byte distance and utilizes this information to split a large scan into smaller chunks that can be run in parallel. Phoenix harnesses HBase features of scan predicate pushdown and coprocessors to push maximum processing on the server side. The default transaction semantics in Phoenix with base tables only is same as HBase; however, recent integration with Tephra [38] enables multi-statement transactions in Phoenix through MVCC. Note, the MVCC transaction support in Phoenix can be turned on/off by starting/stopping Phoenix-Tephra transaction server.

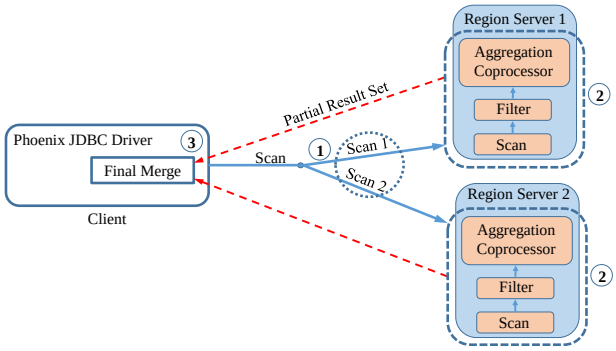


Figure 2.4: Single Table Aggregation Query Processing in Phoenix

Figure 2.4 depicts the processing for a single table aggregation query in Phoenix: 1) client begins by issuing parallel scans to the servers, 2) the results of each scan are then partially aggregated on each server using an aggregation coprocessor, and 3) partially aggregated results are then merged on the client to produce final result. Phoenix join support includes broadcast hash join and merge join. Figure 2.5 depicts the hash join processing in Phoenix: 1) client issues parallel scans to read one input of the join operation and creates

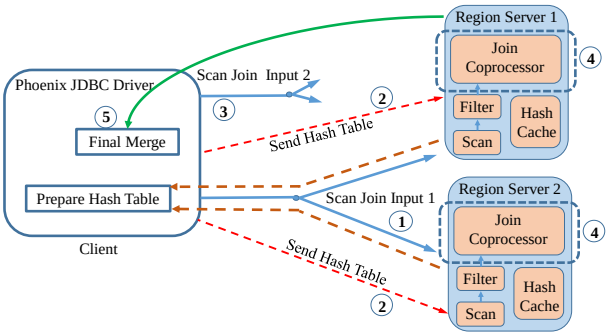


Figure 2.5: Join query processing in Phoenix

hash table of the intermediate result set, 2) the prepared hash table is then sent to and cached on each server hosting the regions of other input, 3) the client then issues parallel scans for the other input of the join operator, 4) the results of each scan are then joined with the cached hash table (of intermediate results) on each server using the join coprocessor, and 5) the joined result sets from each parallel scan are then merged in the client. The broadcast hash join currently requires one side of the join input to completely fit in memory. Phoenix also supports global and local covered secondary indexes, which are used seamlessly by the query optimizer. Phoenix currently cannot recover from mid-query failures.

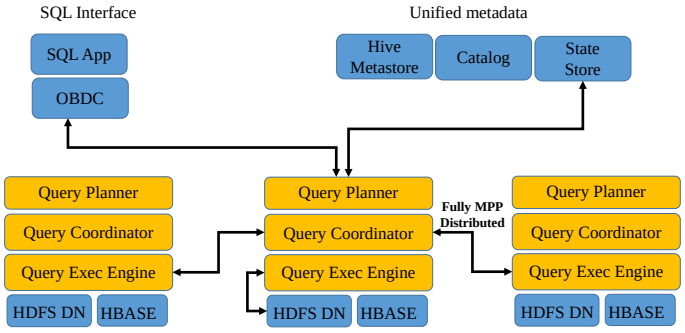


Figure 2.6: Impala architecture overview

2.0.1.2 Amazon S3

Amazon S3 is a object storage system that provides scalability, availability and durability guarantees at commodity costs. S3 exposes a simple API with GET/PUT/DELETE

operations and users can interact with S3 using a REST-style HTTP interface or SOAP interface. S3 objects are immutable and cannot be appended to or changed. Hence, in order to modify a object, a new object with changes needs to be written in full. However, GET operation enables users to fetch parts of an object. Hence, for the columnar storage formats like parquet, inclusion of an object header with start and end points for each column can be utilized to skip irrelevant columns.

2.0.1.3 Impala

Impala is a massively parallel processing (MPP) SQL query execution engine inspired by the Dremel [31] system. Impala architecture includes three daemon services: impalad, catalog and statestore, as depicted in Figure 2.6. Generally, a single impalad service is deployed on each worker node in the cluster. impalad daemons are symmetric, i.e. each impalad daemon can execute a query fragment on behalf of the other daemon, as well as act as a query coordinator for the queries submitted to it. catalog service publishes metadata information to the impalad daemons using statestore publish/subscribe service.

The Impala query planner constructs an execution plan in two steps. In the first step, the query parse tree is transformed into a single node execution plan using available operators. Then, simple heuristics are used to generate single node plans with different join orders and cost-based optimization is then utilized to select the plan with the least cost. In the second step, the selected plan is made distributed by choosing a join procedure (broadcast or hash partitioned) for each join in the single node plan. Impala query planner is implemented in Java.

Impala query execution engine makes extensive use of LLVM library: 1) to generate query specific code for functions that are called numerous times for a single query (e.g. record parser), and 2) to minimize the performance impact of virtual functions by directly calling the required function. Impala query execution engine is implemented in c++; hence query performance is not affected by the garbage collection issues.

Impala can process data stored in a variety of file formats including RCFile, SequenceFile, Avro, Parquet, CSV etc. Impala execution engine is decoupled from the underlying storage engine; hence, it can read data stored in HBase, HDFS, Kudu etc. Impala currently cannot recover from mid-query faults.

2.0.1.4 Drill

Drill is a MPP query execution engine inspired by the Dremel system. The Drill runtime comprises of long running symmetrical daemons. The query optimizer in Drill is derived from the Apache Calcite and utilizes rule (filter pushdown into storage engine) and cost based optimization techniques to generate an efficient execution plan.

The Drill execution engine utilizes vectorized query processing to achieve peak efficiency by keeping CPU pipelines full at all times. The run time code compilation enables Drill to generate efficient custom code for each query. Drill harnesses a pipelined execution model where all tasks are scheduled at once and data moves through the task pipelines. Drill currently cannot recover from mid-query faults.

Drill optimizes for columnar storage as well as columnar execution through an in-memory hierarchical columnar data model. In addition, Drill provides native support for nested data. Drill has the ability to discover schema on the fly. Also, Drill enables federated query processing by allowing users to combine and query the data stored in various underlying storage engines including but not limited to HDFS, NoSQL (e.g. HBase, MongoDB) and Cloud Storage (e.g. S3, Azure Blob).

2.0.1.5 Spark and Spark SQL

Spark is a cluster computing framework that executes data-parallel computations in a scalable and fault-tolerant manner. Resilient distributed datasets (RDDs) represent the primary abstraction in Spark, that leverages the distributed cluster memory and enables efficient data reuse between computations.

An RDD is a read only, partitioned collection of records. Spark API defines operations (*map, join etc.*) to transform an existing RDD into a new RDD. Spark evaluates RDDs in a lazy manner and no computation is initiated until an action operation (*reduce, save etc.*) is invoked, enabling pipelined execution across transformations. Spark achieves significant efficiency as compared to MapReduce framework by harnessing cluster memory to store intermediate results; however, as a downside, it also makes itself susceptible to mid-query failures. Hence, to recover from mid-query faults, Spark logs the lineage of transformations used to derive an RDD and utilizes the transformation history to efficiently recover lost partitions.

Spark SQL is a component in the Spark ecosystem for working with the structured data. Spark SQL introduces the DataFrames API that enables users to seamlessly combine relational queries and procedural API in a single Spark program. DataFrames represent the primary abstraction in Spark SQL. A DataFrame is a distributed collection of data organized into named columns using columnar format. Hence, a DataFrame is analogous to a table in the relational database. In comparison with the Spark's RDD API, Spark SQL's DataFrames API leverages the structure of data to execute operations more efficiently.

The catalyst query optimizer in Spark SQL follows the traditional optimization steps of query parsing/analysis, logical planning, physical planning and code generation. The logical planner in catalyst applies rule based optimizations (predicate pushdown, projection pruning etc.) to optimize the input logical plan. The physical planner generates multiple physical plans for the logical plan using physical operators (e.g. different join algorithms) available in the execution engine. Then, cost-based optimization is used to evaluate and select the plan with the least cost. Finally, Spark SQL utilizes quasiquotes feature of scala language to convert SQL expressions into abstract syntax trees (ASTs), which are then fed into scala compiler to generate the bytecode at runtime. Spark SQL relies on Spark execution engine's lineage based architecture to recover from mid-query faults.

Spark SQL can process data stored in a variety of formats including Avro, Parquet,

CSV, JSON etc. In addition, Spark SQL can read and process data from several storage engines (HDFS, HBase, Hive, MySQL etc.) in a single Spark program.

2.0.2 Large Scale Enterprise Data Management for OLTP Workloads

2.0.2.1 Online Transaction Processing in Relational Databases

Cloud computing has emerged as a new paradigm for on-demand access to shared compute resources (server, storage, network etc.). Virtualization enables platform providers to maximize hardware utilization by hosting multiple virtual machines (VMs) on a physical server and sharing hardware resources across hosted VMs. Since, access to a shared resource by VMs hosted on a physical server may overlap in time, it is interesting and valuable to understand what class of applications are affected by crosstalk between VMs and to what extent. Traditionally, relational databases are used as a backend for web application development; hence, profiling database performance on cloud platforms is important. In addition, since database workloads are generally disk I/O heavy and a para-virtualized host VM performs disk I/O on behalf of guest VM, it is interesting to understand how this indirection affects performance.

Several studies [39],[40],[41] have been undertaken to characterize the impact of virtualization on the network, disk and CPU performance of a VM. Wang et al. utilize micro benchmarks to evaluate network performance in the Amazon EC2 data center and find that network behavior is anomalous with high variance in the network delay. In [41], authors evaluate the performance of latency-sensitive multimedia applications and show that the contention for shared I/O resource between VMs hosted on the same physical node can degrade the CPU, disk jitter and throughput experienced by the application. In [40], El-Khamra et al. explore the performance of HPC applications on the cloud benchmark and observe high variance in communication times. In [42], authors compare the performance of a data intensive MapReduce application on two different platforms: 1) Amazon EC2, 2) Private cloud, and conclude that the variance on EC2 is so high that the wall clock

experiments may only be performed with considerable care.

In [43],[44], authors have reported performance difference between VMs of the same abstract type in data centers like Amazon EC2, attributing the difference in performance to the placement of VMs and the heterogeneous hardware in evolving data centers like Amazon EC2. In contrast with the prior studies, we profile the performance of MySQL database server (hosted inside a guest VM) in a controlled virtualized environment by hosting a single guest VM on a physical server and show that the disk fragmentation can be significantly more pronounced for guest VM, affecting the consistency of developed performance profile.

2.0.2.2 Online Transaction Processing in Distributed Databases

Relational databases represent the primary choice as the storage tier of a multi-tier web application [45], since RDBMs provide ACID transaction semantics, principled mechanism for schema design, and a declarative language for data manipulation. As the application user base and the data footprint grows, the database system must scale up. Relational databases are well suited for vertical scaling; however, hardware and licensing costs could be prohibitively high. As a result, several horizontally scalable distributed databases (NoSQL and NewSQL) [4, 8, 5, 9] have been proposed recently. Distribution of data across the cluster in the distributed databases requires data shuffle over the network to perform the join operation, resulting in slow join performance. NewSQL databases like [8] avoid expensive joins by restricting the joins to the partition keys only; however, this results in limited query expressiveness. Therefore, to support the most common type of joins (inner-join) occurring in the OLTP workloads in an unrestricted manner, we choose and focus on NoSQL databases. However, the join performance must improve to get practical use out of the system. Materialized views represent a standard method for improving the join performance. Next we investigate existing works related to the creation and maintenance of materialized views.

Materialized Views. MVs have been studied from multiple standpoints in the SQL domain: view maintenance, view matching, automated views selection, dynamic view maintenance etc. In [46, 47, 48, 49] authors explore the problem of efficient view maintenance in response to the base table updates. The dynamic views [50] introduce storage efficiency by automatically adapting the number of rows in the view in response to the changing workload. The view matching techniques are utilized in query optimization to determine query containment and query derivability [51, 52, 53]. In [54], authors propose a workload driven mechanism to automate the task of selecting an optimal number of views and indexes for decision support system applications. The MVs selection and maintenance in a transaction processing NoSQL data store raises novel challenges since most of the existing views selection approaches are oblivious to the relationship between schema relations which can lead to heavy view maintenance costs and can shift the bottleneck from reads to writes. To this end, Synergy proposes a novel, schema relationship aware view selection mechanism.

Introduction of views in a NoSQL database generates the requirement for view maintenance and new concurrency controls to ensure consistency between the materialized views and base tables. Next, we examine transactions related works in the NoSQL databases.

Transactions. Transaction support in the majority of the first generation NoSQL stores [4, 55] and Big Data systems [56] is limited to single-keys. G-Store [57] extends HBase to support multi-key transactions in a layer on top using a 2 phase locking protocol. Similar to G-Store, we implement write transactions in a layer on top of HBase. CloudTPS [58] supports multi-key read/write transactions in a highly scalable DHT based transaction layer using optimistic concurrency control (OCC). In [59], authors extend CloudTPS to support consistent foreign-key equi-joins. ecStore [60] provides snapshot isolation using MVCC based on the global timestamp transaction ordering in a decoupled layer on top of an ordered key-value store BATON. ElasTras [61] proposes a novel key-value store that implements MVCC based transactions. Percolator [62] extends Bigtable to allow cross-row, cross-table ACID transactions and enables incremental updates to the web index data

stored in BigTable. Megastore [63] introduces entity groups as a granule of physical data partitioning and supports ACID transactions with in an entity group. F1 [64] is built on top of Spanner [9] and supports global ACID transactions for the Google AdWords business. In contrast with Spanner, Synergy is limited to single data center use; however, Synergy enables enhanced SQL query expressiveness and does not require sophisticated infrastructure including atomic clocks, GPS etc. The NewSQL databases [8, 65] scale out linearly while ensuring ACID semantics; however, the join support is limited to partitioning keys. The first generation of NewSQL systems required all data to reside in main memory; however, recent work [66] overcomes this limitation by keeping cold data on the disk.

Recent works have explored data partitioning as a mechanism to minimize distributed transactions. Next, we survey works related to data partitioning in distributed databases.

Data Partitioning. Megastore [63], F1 [64] and Elastras [61] harness hierarchical schema structure to cluster related data together and minimize the distributed transactions. On the contrary, Synergy generates MVs utilizing hierarchical schema structure to reduce query run times. In [67], authors automate the task of data partitioning by developing a technique for automated selection of root relations in a schema. Schism [68] proposes fine grained data partitioning by co-locating related tuples based on workload logs.

NoSQL databases generally expose a simple data manipulation API including *scan, get, put and delete* operations. Hence, to support SQL on top of a NoSQL database, several SQL layers have been developed. Next, we investigate SQL layers on top of NoSQL databases that enable a simple mapping of database schema and workload.

Database Transformation. In [69], authors present a model to map and store relational tables in a key-value store; however, the mechanism to transform the SQL based workload using the key-value API is not provided. Phoenix [6] and FoundationDB [7] implement a SQL skin that executes the SQL workload using NoSQL primitives. Synergy utilizes Phoenix as a SQL skin to operate on the NoSQL data store.

Table 2.1: Summary and classification of related works.

Approach	Query Engines Compared	Benchmarks	Cluster Specification	DB Size (Max.)	Experiment Metrics
Pavlo et al. [70]	MR, DBMS-X, Vertica	WDA	100 nodes (max), Private Cluster	2.1 TB	RT, Scale Up, Speed Up
Floratou et al. [71]	Hive-MR, SQL Sever PDW	TPC-H	16 nodes, Private Cluster	16 TB	RT, Size Up
Floratou et al. [72]	Hive-Tez, Hive-MR, Impala	TPC-H, TPC-DS derived	21 nodes, Private Cluster	1 TB	RT
[73]	Redshift, Shark, Impala, Hive-MR, Hive-TEZ	AMP Lab BigData Benchmark (ALBB)	5 nodes, Amazon EC2	127.5 GB	RT
Wouw et al. [74]	Hive-MR, Shark, Impala	ALBB, Real World	5 nodes, Amazon EC2	523 GB	RT, Speed Up, Size Up
Pirzadeh et al. [75]	AsterixDB, System-X, Hive-MR, MongoDB	BigFun Micro Benchmark	10 nodes, Private Cluster	800 GB	RT, Scale Up
Shi et al. [76]	Spark, MR	Word Count, Page Rank, K-Means, Sort	4 nodes, Private Cluster	500 GB	RT
Our Work	**P-HBase, Drill, Spark SQL, Impala**	**WDA, TPC-H**	**21 nodes (max), Amazon EC2**	**500GB**	**RT, Scale Up, Size Up**

2.0.3 Large Scale Enterprise Data Management for OLAP Workloads

2.0.3.1 OLAP with Hadoop as Storage Substrate

With the widespread popularity and adoption of Hadoop as the centralized data repository in enterprises, a large number of systems, such as MapReduce [16], Hive [17], Pig[18], Spark [77], Impala [23], HadoopDB [78], etc. have been developed over the past decade for efficient analytics processing on top of Hadoop. SQL-on-Hadoop systems enable users to write queries using familiar SQL syntax; however each system makes different architectural choices in implementing the query optimizer and the query execution engine. As a result, it is valuable to evaluate and benchmark different SQL-on-Hadoop systems against each other. In addition, since parallel DBMSs too are designed for large scale data analysis, it is interesting to compare parallel DBMSs with SQL-on-Hadoop systems. To this end, several recent studies have compared a variety of SQL-on-Hadoop systems and parallel databases using standard benchmarks. Table 2.1 presents a classification of the related works.

In [70] authors compare MR framework with parallel databases and notice that MR is compute intensive and high task startup costs dominate the execution time of short duration jobs. In [71], authors compare Hive with SQL Server PDW and observe that although Hive achieves better scalability, high CPU overhead associated with the RCFile format in Hive results in a slower query performance as compared to the SQL Server PDW. In [72], authors compare Hive and Impala and attribute the disk I/O efficiency, long running daemons and run time code generation in Impala as the reasons for its better performance than Hive. In [74] authors compare Shark, Hive and Impala and observe that Impala exhibits the best CPU efficiency and the join performance worsens as the cluster size increases due to the increased data shuffle. In [75], authors propose a social media inspired micro benchmark to compare AsterixDB [56], System-X, Hive-MR and MongoDB systems. Experimental results show that MongoDB becomes unstable for large aggregations due to memory issues

and the lack of index support in Hive-MR causes point and range queries to be expensive. In [76], authors evaluate MR and Spark for iterative and batch workloads using micro benchmarks to show that the CPU overhead associated with the de/serialization of intermediate results is the primary resource bottleneck.

To the best of our knowledge, this is the first work to thoroughly evaluate, understand and compare the performance of Drill, Phoenix and Spark SQL v2.0 systems for SQL workloads using standard analytics benchmarks including WDA and TPC-H. In addition, we compare the performance of aforementioned systems with Impala, a mature and well studied [72, 74] SQL-on-Hadoop system.

2.0.3.2 OLAP with Object Storage Systems as Storage Substrate

The SQL-on-Hadoop movement has gained traction in the recent years for OLAP. A Hadoop cluster in the cloud environment can either utilize instance storage or network attached Block storage (e.g. Amazon EBS) to store the data. Instance storage is ephemeral, which makes it primarily suitable for adhoc data analysis where as network attached Block storage is expensive and better suited for latency sensitive OLTP workloads. On the contrary, object storage systems like Amazon S3 [25], Microsoft Azure Blob Storage [79], etc., are cheaper than the network attached Block storage and provide better performance per dollar [35] for OLAP workloads. This has led to the emergence of SQL-on-Object-Storage systems like Snowflake [27]. In addition, SQL query engines like Impala [23], Spark SQL [20], Drill [24], etc., have also implemented I/O subsystems to access and process the data stored in Object Stores like S3.

Although object storage systems like S3 deliver better performance per dollar than the network-attached block storage systems like EBS, high access latency and low read throughput can lead to exessively high and non-interactive query response times. Traditionally, use of a caching layer improves the read throughput of data warehousing applications. To this end, distributed caching systems like Tachyon [80] and Alluxio [81] have

been proposed recently, which can enhance the read throughput of query engines that use an object store as a storage backend. The use of an indexing structure to read only relevant data represents another well established method to improve the query performance in a data warehousing system. To this end, the use of horizontal data partitioning in conjunction with min-max indexes or block range indexes [82] represents an effective mechanism to skip reading irrelevant data in object stores like S3. In [83], authors propose and demonstrate efficacy of small materialized aggregates as a lightweight indexing structure to speed-up aggregation queries for OLAP workloads. In [84], authors propose workload driven data partitioning method to enable maximal data skipping during query processing. In [85], authors present a mechanism to implement a scalable and distributed segment tree using a key value data store for answering interval queries.

Chapter 3

Performance Variations in Profiling MySQL Server on the Xen Platform: Is it Xen or MySQL?

Published in International Journal of Computer Science and Information Technologies (IJCSIT) 2014.

Reliability of a performance model is essential to robust resource management of applications on the cloud platform. Existing studies show that the contention for shared I/O induces temporal performance variations in a guest VM and heterogeneity in the underlying hardware leads to relative performance difference between guest VMs of the same abstract type. In this chapter, we demonstrate that a guest VM exhibits significant performance variations across repeated runs in spite of contention free hosting of a single guest VM on a physical machine. Also, notable performance difference between guest VMs created equal on physical machines of a homogeneous cluster is noticed. Systematic examination of the components involved in the request processing identifies disk I/O as the source of variations. Further investigation establishes that the root cause of the variations is linked with how MySQL manages the storage of tables and indexes on the guest VM's disk file system. The observed variations in performance raise the challenge of creating a consistent and repeatable profile. To this end, we present and evaluate a black box approach based on database population from a snapshot to reduce the perceived performance variations. The experimental results show that the profile created for a database populated using a snapshot can be used for performance modeling up to 80% CPU utilization. We validate our findings on the Amazon EC2 cloud platform.

The rest of this chapter is structured as follows. We describe our experiment methodology in section 3.1. In section 3.2, we present our observations related to the complex read

statement performance. We describe our experiments performed on the Amazon EC2 cloud platform in section 3.3. In section 3.4, we evaluate the performance of simple read statement. We discuss our performance observations related to the write statement in section 3.5, and conclude in section 3.6.

3.1 Experiment Methodology

We perform experiments in two environments: laboratory test bed and the Amazon EC2 cloud platform. Our laboratory test bed represents a controlled execution environment which allows us to do fine grained resource allocation. We utilize the laboratory test bed to perform extensive experiments and present the performance observations in sections 3.2, 3.4 and 3.5. We utilize Amazon EC2 to perform experiments on a large cluster and validate our findings on a commercial cloud platform. Our Amazon EC2 experiment setup and performance observations are presented in section 3.3. Next, we describe our laboratory test bed.

Our laboratory experiment environment consists of a Xen based four node homogeneous cluster connected through Linksys 5-Port 10/100 switch as depicted in Figure 3.1. Each physical node in the cluster is a Dell work station configured with Intel quad core 3.1 GHz processor, 8GB RAM and 500 GB 6.0 GB/s SATA hard drive. Each node is running Xen v4.1.2 as the virtual machine monitor in the paravirtualized mode and Linux kernel v3.0.69 as the host VM (dom-0). We reserve one node for running the client workload generator (CWG) as depicted in Figure 3.1. It has a single VM (dom-0) residing on it. CWG runs inside that VM. Remaining three nodes, each have a host VM (dom-0) and a guest VM (dom-u) residing on it. In order to prevent the variations caused by the contending guest VMs, we host a single dom-u on each node. Guest VM is running Linux kernel v3.0.69. In order to profile MySQL server in the guest VM, it is hosted inside the dom-u as depicted in Figure 3.2(a). Similarly, to evaluate the performance of MySQL server in the host VM, it is hosted inside the dom-0 as depicted in Figure 3.2(b).

The following sub-sections describe our design choices related to the software components that run on our test bed including Xen, MySQL Server and CWG. In addition, we give query processing overview, describe our performance metric and SQL statements that constitute our workload. We conclude this section with the description of our experiment setup.

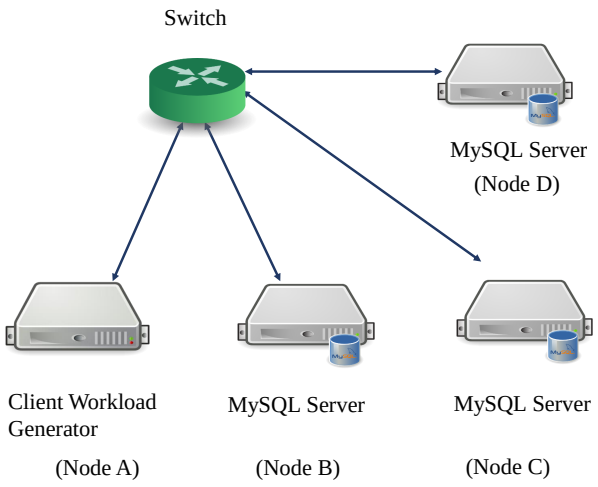
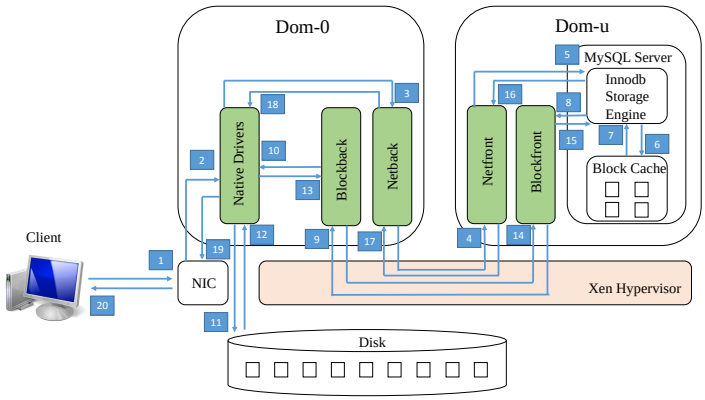


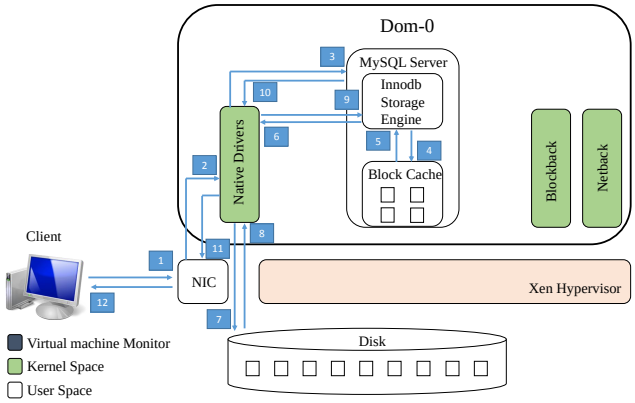
Figure 3.1: Laboratory test bed

3.1.1 Request Processing Overview

First, we consider the scenario where MySQL server is hosted inside dom-u, as depicted in Figure 3.2(a). Client request is initially received by the native driver residing in dom-0 (1), please refer to the step number in the figure for each step here. Then, native driver transfers the request to the netback driver residing inside dom-0 (2). Request is then demultiplexed by the netback driver to the netfront driver residing in dom-u (3). Finally, MySQL server receives the request from the netfront driver (4). In order to process the request, Innodb storage engine first checks the block cache to see if the data blocks required to prepare the reply reside in it (6,7). However, if a cache miss occurs, then the storage engine sends a request for the required blocks to the blockfront driver (8). Blockfront driver then requests the blockback driver in dom-0 to perform the disk I/O on its behalf (9). Blockback driver forwards the request to the native driver (10). Native driver reads



(a) MySQL Server Hosted Inside Dom-u



(b) MySQL Server Hosted Inside Dom-0

Figure 3.2: Request Processing Overview With MySQL Server Hosted Inside Xen Domains

the blocks from the disk (11,12) and send them to the blockback driver (13). Blockback driver demultiplexes blocks to the blockfront driver in dom-u (14). Finally, storage engine receives the data blocks from the blockfront driver (15). Once a reply is prepared then, it is sent to the netfront driver (16) which in turn forwards it to netback driver in dom-0 (17). Netback driver forwards the reply to the native driver (18) from where it is finally received by the client (20).

Next, we consider the scenario where MySQL server is hosted inside dom-0, as depicted in Figure 3.2(b). Native driver in dom-0 receives the client request (2) and forwards it directly to the MySQL server (3). Innodb storage engine then checks the cache for the required data blocks (4,5). If a cache miss occurs then, storage engine directly requests

the native driver to read the disk data blocks (6). Once a reply is prepared, MySQL server sends the reply to the native driver (10) from where it is delivered to the client (12).

3.1.2 Database Server Specifics

We use MySQL v5.1 as the database server with innodb storage engine. MySQL server is configured with the *max_connections* parameter set to 2000 and the *query_cache_size* parameter set to 0 for all experiments. *max_connections* parameter defines the maximum number of permitted simultaneous client connections. *max_connections* parameter value is increased so as to stress the server and evaluate performance at higher request rates. *query_cache_size* parameter defines the amount of memory allocated to store the query results. Query cache is turned off as we are interested in understanding the performance impact of innodb specific buffer pool cache. Innodb buffer pool is a block level cache for caching data and indexes in memory. *innodb_buffer_pool_size* (γ) parameter defines the size of the block cache. γ is an input parameter to our profiling model.

Database is populated utilizing the population script provided with the TPC-W benchmark [86]. Population script mimics the behavior where users modify the database state by issuing INSERT statements. We modify the script to use the innodb storage engine. Database size (S) can be modulated by varying two parameters in the script: NUM_EBS and NUM_ITEMS. To perform experiments with the default database size ($S_{default}$), we populate database with NUM_EBS set to 100 and NUM_ITEMS set to 10000.

3.1.3 Xen Hypervisor and Virtual Machine Configuration

We have installed Xen v4.1.2 in the paravirtualized mode with standard netback/netfront and blockback/blockfront devices as the networking and storage systems respectively. Both dom-0 and dom-u run Ubuntu with Linux kernel v3.0.69. Logical volume manager (LVM) is the industry standard for creating dom-u file system due to its flexible management [87]. Hence, we create LVM backed dom-u file system. Specifically, we create a

single physical partition and assign it to a logical volume group. Then, we create a single logical volume for dom-u inside that volume group.

We profile the database performance by hosting the MySQL server inside both dom-0 and dom-u. To profile the MySQL server inside a dom-u residing on a physical node, we allocate one virtual CPU (vcpu) to the dom-u and pin it down on a physical CPU (pcpu). Also, we allocate one vcpu to the dom-0 residing on the same physical node and pin it down on a pcpu. This results in a symmetric vcpu and pcpu allocation across both dom-0 and dom-u. Similarly, in order to profile the MySQL server inside a dom-0, we destroy the dom-u residing on a physical node and host only dom-0. Again, we allocate one vcpu to the dom-0 and pin it down on a pcpu.

3.1.4 Client Workload Generator

Workload generator emulates the client behavior by invoking SQL statement requests at the MySQL server. Workload (λ) is generated through an open model where each new request is invoked independent of any previous requests. The open workload generator model is characterized by request arrival into the system according to some arrival process. Recently, a poisson process was utilized to model the open workload generator [88], [89]. Hence, we adopt the same model and choose poisson as the arrival process where request inter arrival times are exponentially distributed.

3.1.5 Benchmark SQL Statements

We utilize the TPC-W benchmark to identify the benchmark SQL statements. TPC-W is a transactional web benchmark modeled after e-commerce web applications. We analyze the application tier servlets to extract all SQL statements that can be invoked at the database tier. Extracted SQL statements can be classified into two categories: *read* and *write*. Based on the query execution plan obtained through *EXPLAIN* statement [90], read statements can be further classified into two query types: *simple* and *complex*.

Table 3.1: Notations commonly used throughout this chapter.

τ	Mean response time
τ_{se}	Standard error of τ
λ	Workload
γ	innodb_buffer_pool_size
S	Database size
$S_{default}$	Default database size with, NUM_EBS=100 & NUM_ITEMS=10000
dom-0	Host VM
dom-u	Guest VM
θ	CPU utilization
θ_{mean}	Mean CPU utilization
μ	Mean network delay
μ_{se}	Standard error of μ
d	Experiment duration Set to 180 seconds

3.1.5.1 Simple Query

A simple query utilizes indices to prepare the final result set. We evaluate the performance of all extracted *simple* queries and observe that all queries exhibit similar performance trend. Also, the difference between the query response times is negligible. Hence, we select and describe a representative *simple* query. Specifically, the query is a *SELECT* statement that filters records from the *Customer* table based on the specified unique user name (*customer.i_uname = ?*). The filtered *Customer* table is then inner joined with the *Address* and *Country* tables. Please refer to the TPC-W documentation [86] for the description of table schemas. The query syntax is presented in Figure 3.3.

```
SELECT *
FROM customer, address, country
WHERE customer.c_addr_id = address.addr_id AND
address.addr_co_id = country.co_id AND customer.c_uname = ?
```

Figure 3.3: Simple Query

3.1.5.2 Complex Query

In addition to the use of indexes, a complex query may involve operations like creation of internal temporary table, sorting etc. Depending on the size of temporary table and data set to be sorted, disk file system may be used to prepare the final result set. We evaluate the performance of all extracted *complex* queries and observe significant performance variations. The magnitude of variation is different across queries; however, the performance trend is similar. Next, we select and describe a representative *complex* query. Specifically, query is a *SELECT* statement that filters records from the *Item* table based on the specified subject (*item.i_subject = ?*). Filtered *Item* table is then inner joined with the *Author* table followed by the sorting operation on a specified column *item.i_pub_date*. Final result set is then limited to a maximum of 50 records. Query syntax is presented in Figure 3.4.

```
SELECT i_id, i_title, a_fname, a_lname
FROM item, author
WHERE item.i_a_id = author.a_id AND item.i_subject = ?  ORDER
BY item.i_pub_date DESC item.i_title LIMIT 0,50
```

Figure 3.4: Complex Query

Any realistic workload will have a mix of both *simple* and *complex* queries. Now, magnitude of response time for a complex query is significantly larger than that of a simple query. Therefore, *complex queries represent the heavy hitters and variance in their performance can lead to the violation of SLA negotiated response time. Hence, we choose the complex query presented in Figure 3.4 as the benchmark read statement*. In section 3.4, we evaluate performance of the *simple* query presented in Figure 3.3.

We evaluate the performance of all extracted *write* statements and observe that all statements exhibit similar performance trend. Hence, we select an *INSERT* as a representative benchmark write statement. Specifically, statement is a *INSERT* into table *order_line*. Statement syntax is presented in Figure 3.5.

```
INSERT IGNORE INTO order_line (ol_id, ol_o_id, ol_i_id, ol_qty,
ol_discount, ol_comments)
```

Figure 3.5: Write Statement

3.1.6 Performance Metric

We subject MySQL server to the requests generated by the CWG. For each generated request, we record its response time. Mean response time (τ) for all the requests generated during a time interval represents our performance metric. Let k requests are generated during a time duration of d seconds and r_i denotes the response time for the i^{th} request. Then, τ for an experiment with duration d is,

$$\tau = (\sum_{i=1}^{k} r_i)/k \, .$$

3.1.7 Experiment Setup

We design experiments by varying parameters both intrinsic and extrinsic to the MySQL server. Intrinsic parameter includes γ and extrinsic parameter includes λ. Hence, each individual experiment is represented as a set of two parameter values (γ, λ). For each parameter setting, we perform experiments both on dom-0 and dom-u. Experiments on dom-0 serve as a reference, since dom-0 performance is expected to be close to native performance. However, the focus of this study to understand performance in dom-u since dom-u corresponds to a VM instance that cloud platform leases. Also, we repeat each experiment for both the read and the write statements selected in section 3.1.5. During the read statement profile generation, we restart MySQL server before the start of each new experiment. In case of write statement profiling, we drop the existing database and repopulate before the start of each new experiment.

For each experiment, we record the CPU utilization (θ) at regular intervals inside the VM hosting the MySQL server. We utilize the *vmstat* utility provided with the *sysstat*

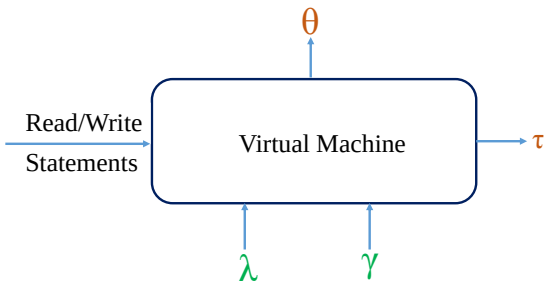


Figure 3.6: Profiling model

package for Linux environment to record θ. For an experiment starting at time t, with a duration of d seconds, we take measurements at regular intervals of 3 seconds starting at time $t+30$ seconds and ending at time $t+(d-30)$ seconds. As an example, for an experiment duration of 180 seconds, we take 40 measurements. We set duration d of each experiment to 180 seconds. Also, we repeat each experiment 20 times and report the mean value and the standard error of the mean. Figure 3.6 depicts our profiling model for an experiment with the input and the output parameters.

3.2 Performance Evaluation of Benchmark Read Statement

In this set of experiments, we profile the benchmark complex query presented in Figure 3.4, on our laboratory test bed. Innodb buffer pool cache is turned off by setting the value of γ to 0. We increase λ in increments of 10 starting at 10 req/sec until θ in the VM is less than 100%. Database is populated for $S_{default}$.

3.2.0.1 Dom-u Performance Discussion

Figures 3.7a and 3.7b depict the impact of λ on query τ and θ_{mean} respectively for MySQL server hosted inside dom-u on nodes B, C and D. VMs exhibit different request handling capacity since nodes B and C saturate at 40 req/sec where as node D saturates at

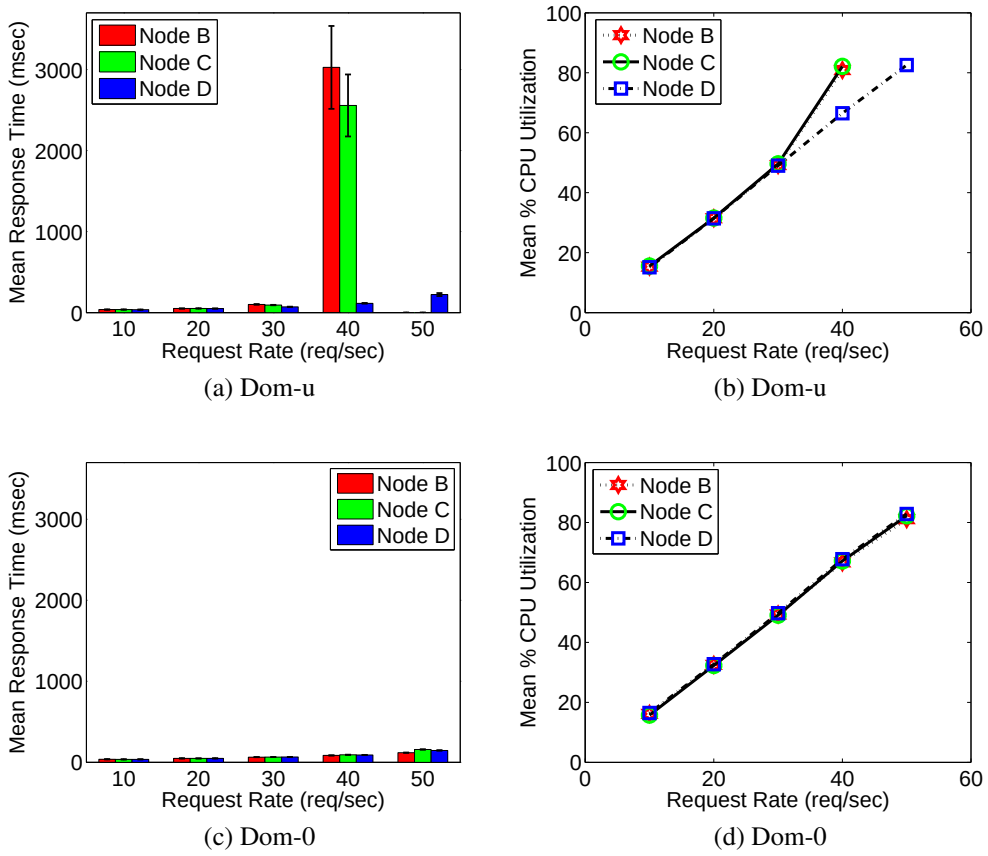


Figure 3.7: Read statement performance comparison between dom-0 and dom-u VMs for $\gamma = 0$ and $S = S_{default}$

50 req/sec. Also, a direct correlation between τ and θ_{mean} is observed in each VM. Figure 3.7a shows a notable difference in the performance of three VM's for a request rate greater than 30 req/sec. Also, for dom-u on node A, at a request rate of 40 req/sec and 80% θ_{mean}, τ_{se} is in excess of 500 msec, exhibiting a high variance across repeated experiment runs in a VM.

Query τ_{se} is less than 5 msec for request rate up to 30 req/sec in all dom-u's. Also, relative difference in query τ is less than 30 msec up to a request rate of 30 req/sec between any pair of dom-u's. Hence, profiled relationships can be utilized for performance modeling up to 50% θ_{mean} since, 50% θ_{mean} corresponds to a request rate of 30 req/sec.

3.2.0.2 Dom-0 Performance Discussion

Figures 3.7c and 3.7d depict the performance profiles for the MySQL server hosted inside dom-0 on nodes B, C and D. Similar to dom-u, a direct correlation between τ and θ_{mean} is observed in each VM. However, in contrast to dom-u, a dom-0 VM exhibits negligible performance variance between repeated runs of an experiment. In Figure 3.7c, query τ_{se} is less than 5 msec up to a request rate of 50 req/sec for all dom-0's. Also, difference in query τ between any pair of dom-0's is less than 40 msec up to a request rate of 50 req/sec. Hence, profiled relationships can be utilized for performance modeling up to 80% θ_{mean} since, 80% θ_{mean} corresponds to the request rate of 50 req/sec.

Summary: Query performance varies significantly across the dom-u's and between repeated runs on the same dom-u for a θ_{mean} greater than 50%. This behavior inhibits us from directly using the profiled relationship for performance modeling on dom-u.

When MySQL server is hosted inside dom-u, a client request is processed utilizing components residing both inside the Xen I/O virtualization subsystem and the MySQL server as described in section 3.1.1 and depicted in Figure 3.2(a). Therefore, in the next three subsections, we systematically investigate each component utilizing black box techniques to identify the root cause of the performance variations. Specifically, we ascertain whether it is the network I/O or the disk I/O, that is responsible for the perceived variations. In addition, we identify the software component to assign fault to.

3.2.1 Impact of network I/O on performance variations

In this section, we aim to confirm if the network I/O subsystem in the Xen is causing the observed performance variations. The Xen network I/O subsystem comprising of the *netback/netfront* drivers, processes the packet streams for the client request and the MySQL response. Now, the response time of a query is the sum of network delay and the query execution time. Therefore, in order to determine the network delay for a query, we record

both its response time and the execution time.

Query response time is recorded in the CWG. However, for recording the query execution time, we utilize the MySQL server's built in profiler. MySQL profiler is specific to an individual client session and profiles all the statements sent to the server during a session. We send each client query request in an independent session. We turn the profiling on for the session before sending the query. After receiving the query response, we fetch the profile for the executed query from the server. Figure 3.8 depicts our procedure to obtain both the response time and the execution time of a query in a client session. Next, we describe our experiment design.

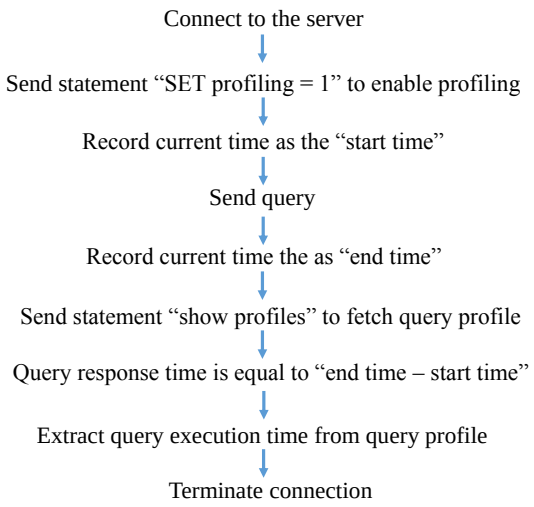
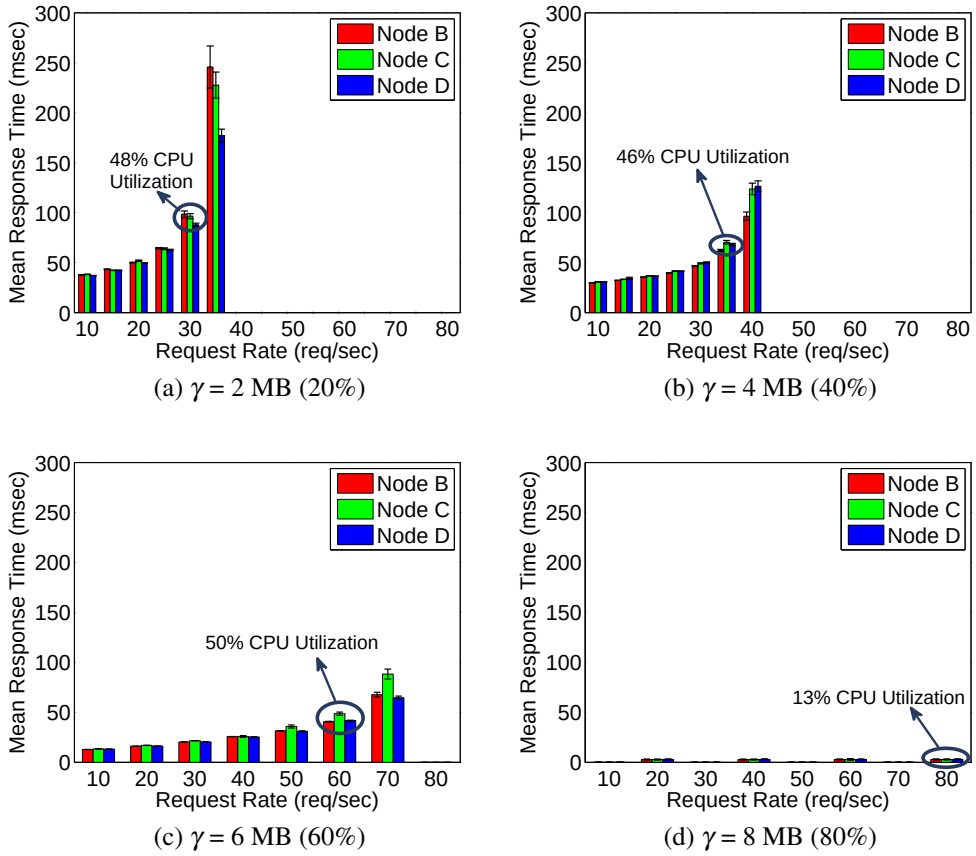


Figure 3.8: Measuring query execution time in a client session

In this set of experiments, we measure the impact of workload on the query network delay. We increase λ in increments of 10 starting at 10 req/sec until θ in the VM is less than 100%. Also, for each value of λ, we run the experiment for 180 seconds. We compute the mean network delay (μ) and the standard error of the mean (μ_{se}) for each experiment. Also, each experiment is repeated on nodes B, C and D.

Table 3.2 presents the summary of experiment results. Query μ increases minimally with the increase in λ. Also, μ_{se} is inconsequential for all experiments. In addition, Query



(a) $\gamma = 2$ MB (20%) (b) $\gamma = 4$ MB (40%)

(c) $\gamma = 6$ MB (60%) (d) $\gamma = 8$ MB (80%)

Figure 3.9: Read statement performance comparison between dom-u's for different cache allocations.

μ is negligible in comparison with the query τ for all values of λ. Also, since, the time spent by a query in the netback/netfront drivers is always bounded by the network delay. Therefore, we conclude that the Xen network I/O subsystem is not responsible for the observed variations.

Table 3.2: Query μ and μ_{se} with dom-u inside Nodes B, C and D

λ	Node B			Node C			Node D		
	Number of requests generated	μ	μ_{se}	Number of requests generated	μ	μ_{se}	Number of requests generated	μ	μ_{se}
10	1755	1.48	0.01	1837	1.47	0.01	1857	1.43	0.01
20	3613	1.47	0.007	3513	1.46	0.007	3655	1.56	0.02
30	5488	1.51	0.01	5509	1.54	0.01	5481	1.55	0.01
40	7133	2.05	0.05	7175	1.61	0.03	7341	1.81	0.02
50							8977	3.68	0.07

Summary: Insignificant values of μ and μ_{se} suggest that the netback/netfront drivers are not inducing the perceived variations.

3.2.2 Impact of disk I/O on performance variations

In this section, our goal is to ascertain whether disk I/O is the source of noticed performance variations. It is well understood that the Innodb buffer pool cache improves the database performance by keeping the frequently accessed data in memory [88],[91]. However, we employ buffer pool cache to reduce the disk I/O performed by the storage engine and evaluate its impact on the observed performance variations. Specifically, our objective is to understand the impact of different cache allocations on the performance of MySQL server hosted inside the dom-u. Next, we describe our experiment design.

In this set of experiments, we evaluate the server performance only inside dom-u since server exhibits significantly larger variations in dom-u as compared to dom-0. Again, we profile the benchmark complex query presented in Figure 3.4. It involves join on two tables: *item* and *author*. The combined size of *data+index* for the two tables is 10 MB. Hence, setting the value of γ to 10 MB (100%) makes both the tables memory resident. As a result, workload becomes compute bound. We increase γ in increments of 2 starting at 2 MB. λ in increased until θ in the VM is less than 100%.

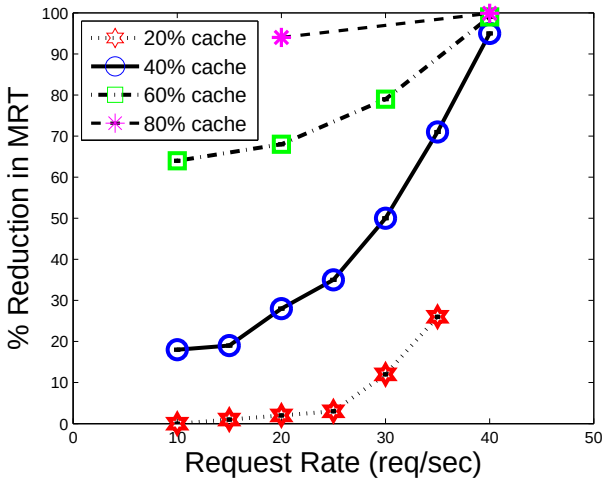


Figure 3.10: Percentage reduction in τ for different values of γ relative to τ for $\gamma = 0$ on Node B's dom-u

Figure 3.9 illustrates the impact of λ on query τ for different cache allocations. With

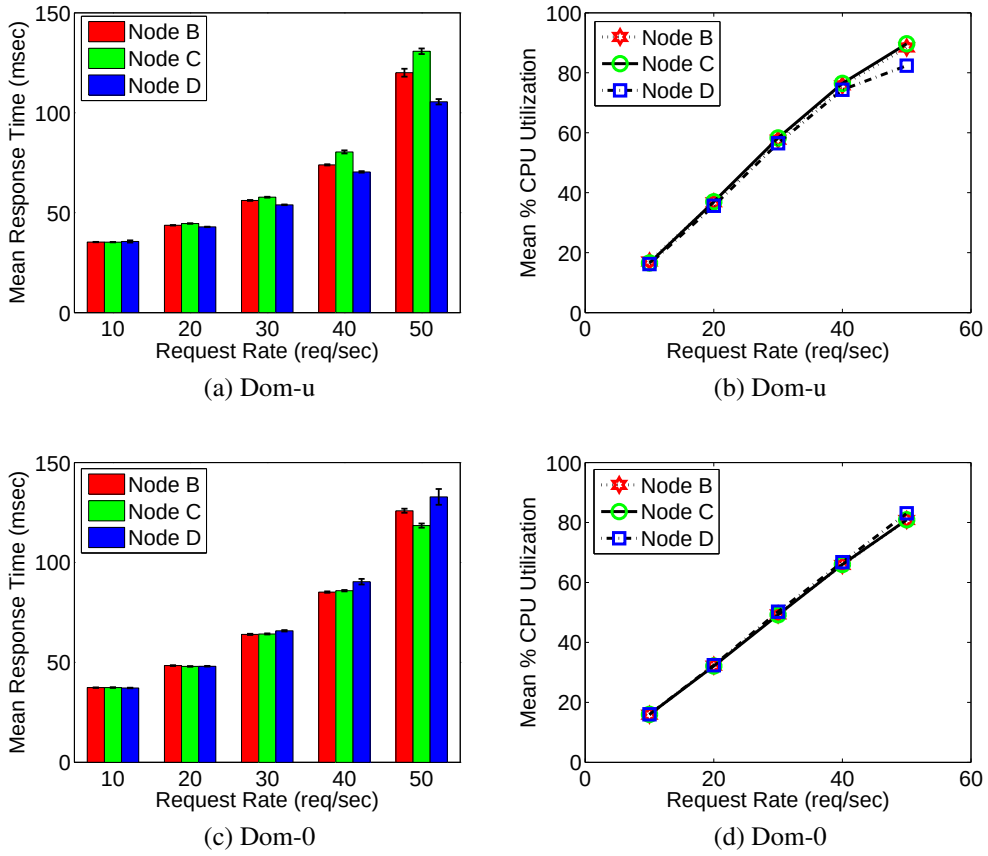
the increase in the value of γ, both, τ and τ_{se} monotonically decrease in each dom-u for the same value of λ. Also, with the increase in cache, inter dom-u performance difference monotonically reduces. For 2 and 4 MB values of γ, query τ increases non-linearly with the increase in λ across all dom-u's. For $\gamma = 6$ MB, query τ increases almost linearly with the increase in λ. Query τ remains nearly constant with the increasing λ for $\gamma = 8$ MB and higher cache allocations.

CPU utilization guideline for performance modeling does not change with the increase in cache allocation. As depicted in Figure 3.9, for cache allocations of 20% (2 MB), 40% (4 MB) and 60% (6 MB), performance variations start to manifest for θ_{mean} value of more than 50%. Hence, profiled relationships could be utilized for performance modeling up to 50% θ. This behavior is similar to the experiments presented in section 3.2 with $\gamma = 0$. For $\gamma = 8$ MB, we plot results up to 80 req/sec; however, we do not observe any variations up to 120 req/sec and 23% θ.

In order to understand how does the query τ decrease with the increase in cache allocation relative to the query τ for no cache, we plot Figure 3.10. Figure 3.10 depicts the percentage reduction in τ for the different values of γ relative to τ for $\gamma = 0$. We plot results only for the dom-u residing on Node B; however, the trend is similar across all dom-u's. For $\gamma = 2$ MB, we do not observe a significant performance benefit up to 25 req/sec. For $\gamma = 4$ MB, performance benefit increases in a steep manner with the increase in λ. Hence, for a smaller cache allocation, benefit is larger at higher request rates.

Summary: Monotonic reduction in performance variations with the increase in cache size indicate that disk I/O is responsible for the perceived variations.

Now, disk I/O involves two software components: Xen disk I/O subsystem and Innodb storage engine, as described in section 3.1.1. Xen disk I/O subsystem comprises of *blockback* and*blockfront* drivers. *blockback/blockfront* drivers serve the disk block read/write requests generated by the MySQL server. However, the storage layout of the tables and indexes on the disk is managed by the MySQL server's Innodb storage engine. Therefore,



(a) Dom-u (b) Dom-u

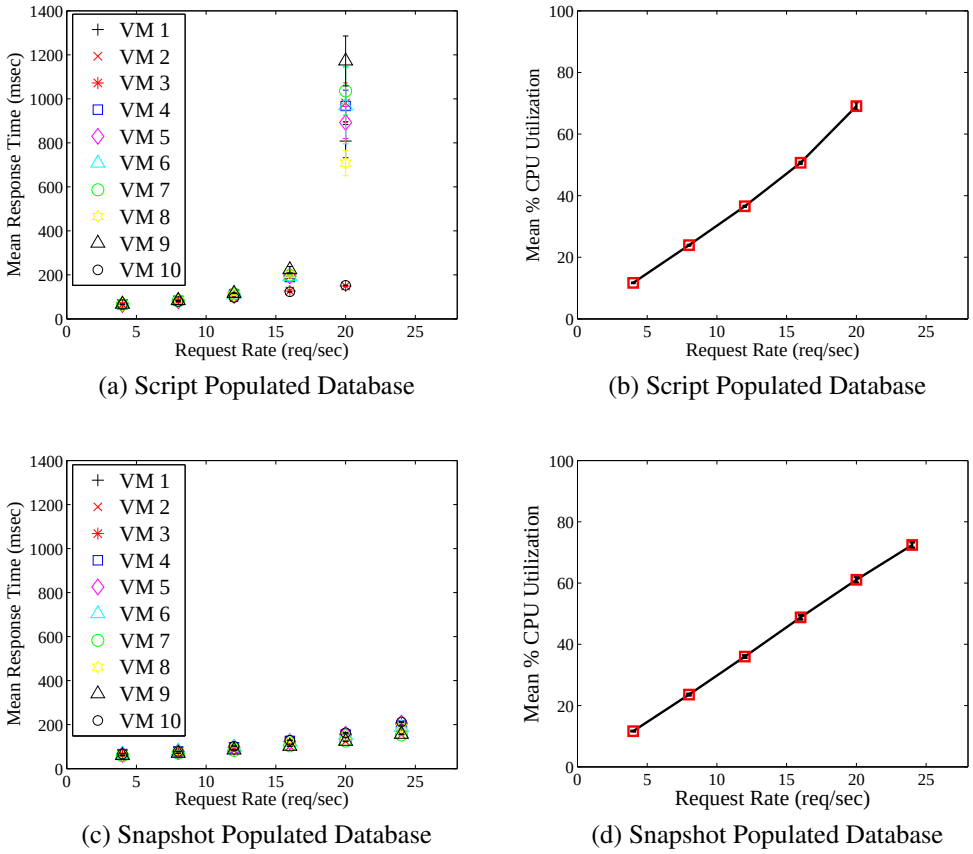
(c) Dom-0 (d) Dom-0

Figure 3.11: Read statement performance comparison between dom-0 and dom-u VMs for database populated using *mysqldump* snapshot

in the next section, we attempt to identify the software component that should be assigned the fault to.

3.2.3 Impact of database population from snapshot on performance variations

In this section, we aim to verify whether the MySQL server's storage engine is responsible for the observed performance variations. MySQL server does not expose a direct mechanism to control the storage layout of tables and indexes on the disk. However, *mysqldump* backup utility can be utilized to take a snapshot of the database and then the database can be repopulated using the snapshot. Therefore, we employ *mysqldump* to alter the database storage on the disk and evaluate its performance impact. Next, we describe our experiment



(a) Script Populated Database

(b) Script Populated Database

(c) Snapshot Populated Database

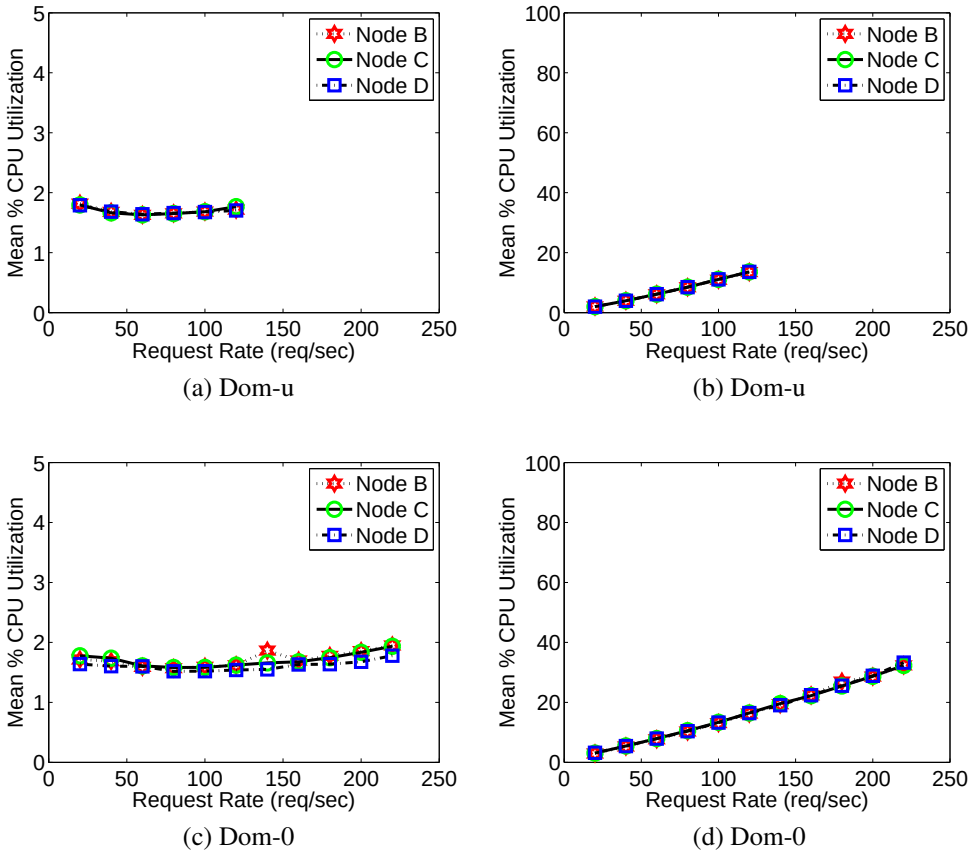
(d) Snapshot Populated Database

Figure 3.12: Read statement performance comparison between database populated from script and snapshot on Amazon EC2 cloud platform

design.

In this set of experiments, we first take the database dump on each VM using the *mysqldump* backup utility. Next, we drop the existing database on each VM and repopulate it using the dump file. Again, we profile the benchmark complex query presented in Figure 3.4. Innodb buffer pool cache is turned off by setting the value of γ to 0. Also, λ is increased in increments of 10 starting at 10 req/sec until θ in the VM is less than 100%.

As depicted in Figures 3.11a and 3.11c, query τ difference between any two VMs is less than 20 msec for λ up to 50 req/sec. Also, query τ_{se} across repeated runs on any VM is less than 3 msec up to 80% θ_{mean}. Hence, relationship profiled on one VM could be utilized for performance modeling on any other VM up to 80% θ. Also, it can be noticed from Figures



(a) Dom-u (b) Dom-u

(c) Dom-0 (d) Dom-0

Figure 3.13: Simple query performance comparison between dom-0 and dom-u VMs with $\gamma = 0$ and $S = S_{default}$

3.7 and 3.11, that query τ in a VM is smaller for a snapshot populated database than the script populated database for the same λ.

Populating the database from a snapshot increases the request handling capacity of the MySQL server hosted inside a dom-u. As a consequence, service providers can manage the SLA with a smaller cluster size. This results in reduced rental cost and increased profit margins for the service provider. However, as a downside, database remains offline during the re-population phase. Also, as the size of database grows, re-population time also increases.

Summary: Database population from a snapshot significantly reduces variance in performance both across dom-u's and between repeated runs on a dom-u. Hence, we conclude

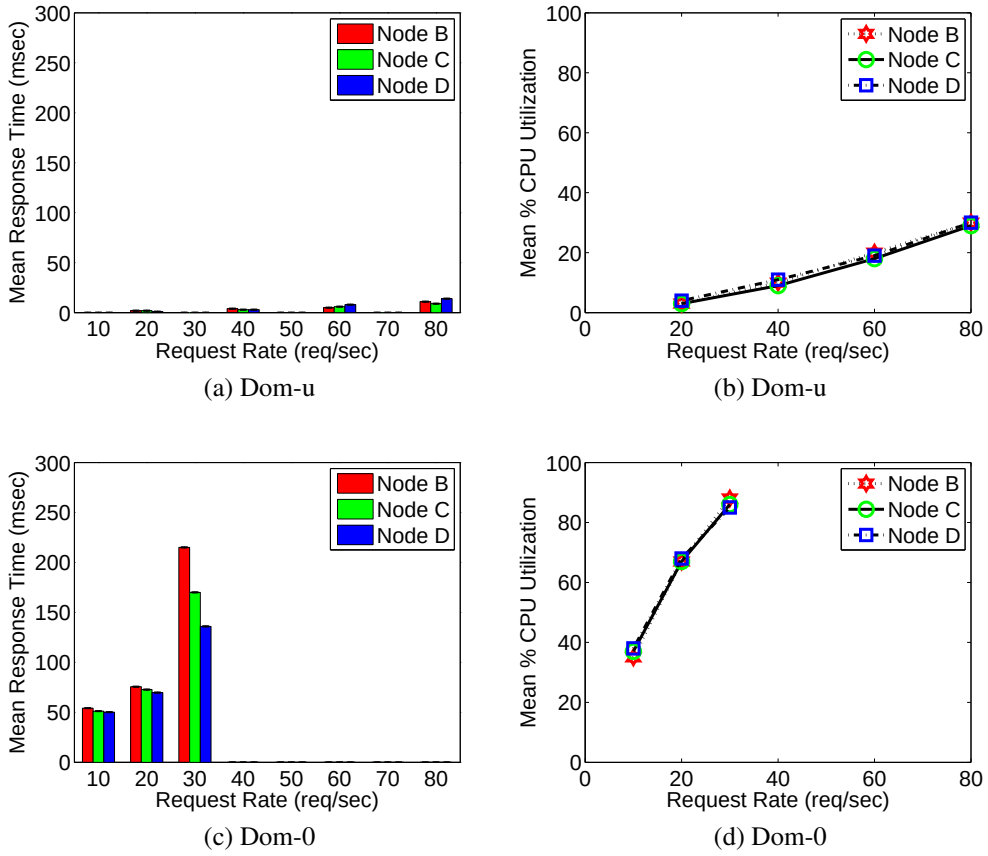


Figure 3.14: Write statement performance comparison between dom-0 and dom-u VMs for $\gamma = 0$ and $S = S_{default}$

that the root cause of variations is linked with how MySQL manages the storage of tables and indexes on the guest VM's disk file system

3.3 Amazon EC2 Experiments

We performed experiments on Amazon EC2 to validate our findings on a commercial cloud platform. Specifically, we leased 20 guest VMs from the *us-east-1b* availability zone in the US-East region. 10 VMs are *m1.medium* instances and the remaining 10 are *m1.small* instances. We boot the VMs with Ubuntu 11.10 Amazon Machine Image (AMI). Each VM has a 8 GB standard Elastic Block Store (EBS) volume attached to it. We harness the *m1.medium* VMs to host the MySQL server since a *m1.medium* VM has a greater request

handling capacity than a *m1.small* VM. This behavior aids in generating a representative query profile. CWG runs inside the *m1.small* VMs. In this study, we profile the benchmark complex query presented in Figure 3.4. Innodb buffer pool cache is turned off by setting the value of γ to 0. Also, we increase λ in increments of 4 starting at 4 req/sec until θ in the VM is less than 100%.

3.3.1 Database population from the script

In this set of experiments, database is populated using the population script for $S_{default}$. As depicted in Figure 3.12a, for λ higher than 12 req/sec, τ of VMs 2 and 10 differs significantly from the rest of the VMs. Also, for a request rate of 20 req/sec, query exhibits a τ_{se} of 173 msec in VM 5. Figure 3.12b presents the mean and the standard error of θ_{mean} for all the VMs. *Similar to the experiments in section 3.2, query performance varies significantly, both across the guest VMs and between repeated runs on the same guest VM.*

3.3.2 Database population from the snapshot

In this set of experiments, we take the database dump on each VM using the *mysqldump* backup utility. Next, we drop the existing database on each VM and repopulate using the dump file. As depicted in Figure 3.12c, query τ_{se} is less than 3 msec between the repeated runs on each VM. Also, query τ difference between any two VMs is less than 50 msec. Hence, *similar to the laboratory environment, database population from the dump significantly reduces the performance variations on Amazon EC2 platform as well.*

Summary: Benchmark complex query exhibits similar performance characteristics both on the laboratory test bed and the Amazon EC2 cloud platform.

3.4 Performance evaluation of benchmark simple query

In this set of experiments, we profile the benchmark simple query presented in Figure 3.3. Innodb buffer pool cache is turned off by setting the value of γ to 0. We increase λ in increments of 20 starting at 20 req/sec until the server refuses any more connections. Database is populated for $S_{default}$.

As depicted in Figures 3.13a and 3.13c, difference in τ for the same value of λ is negligible between any two VMs. Also, the difference between τ for λ=20 and τ for λ=120 is less than 1 msec for all the VMs. Similarly, query τ_{se} is less than 1 msec across all the experiments on each VM. Hence, for a simple query, performance trend is similar on both dom-0 and dom-u. However, request handling capacity of a dom-0 (220 req/sec) is almost double in comparison with the request handling capacity of a dom-u (120 req/sec).

Summary: Benchmark simple query exhibits negligible performance variations up to a high λ value of 120 req/sec in dom-u.

3.5 Performance evaluation of benchmark write statement

In this set of experiments, we profile the benchmark *write* statement presented in Figure 3.5. Innodb buffer pool cache is turned off by setting the value of γ to 0. Database is populated for $S_{default}$. In dom-0, we increase λ in increments of 10 starting at 10 req/sec until θ in the VM is less than 100%. In dom-u, we increase λ in increments of 20 starting at 20 req/sec until the server refuses to accept any more connections.

As depicted in Figures 3.14a and 3.14c, write request handling capacity of a dom-u is more than twice that of a dom-0. Statement τ increases linearly with the increase in request rate on a dom-u. Whereas, on a dom-0, statement τ increases non-linearly with the increase in request rate.

Summary: Benchmark write statement performance exhibits negligible performance variations up to a request rate of 80 req/sec in dom-u

3.6 Chapter Summary

In this chapter, we demonstrate novel performance variations across repeated runs of a dom-u and between dom-u's created equal on homogeneous hardware. We systematically investigate and identify disk I/O as the origin of variations. In addition, we show that it is not the Xen disk I/O subsystem, rather the MySQL storage engine which is responsible for the observed performance variations. Also, we present and evaluate a black box approach to reduce variations based on the *mysqldump* backup utility. Experimental results show that database population from a snapshot reduces perceived variations significantly and increases request handling capacity of a dom-u. Hence, *mysqldump* utility can be exploited to improve the performance of a guest VM on the cloud platform. However, as a downside, database remains offline during the re-population phase. We also evaluate the impact of different cache allocations on the database performance.

Chapter 4

A Comparative Analysis of Materialized Views Selection and Concurrency Control Mechanisms in NoSQL Databases

Published in IEEE Cluster Conference 2017.

Increasing resource demands require relational databases to scale. While relational databases are well suited for vertical scaling, specialized hardware can be expensive. Conversely, emerging NewSQL and NoSQL data stores are designed to scale horizontally. NewSQL databases provide ACID transaction support; however, joins are limited to the partition keys, resulting in restricted query expressiveness. On the other hand, NoSQL databases are designed to scale out linearly on commodity hardware; however, they are limited by slow join performance. Hence, we consider if the NoSQL join performance can be improved while ensuring ACID semantics and **without** drastically sacrificing write performance, disk utilization and query expressiveness.

This chapter presents the Synergy system that leverages schema and workload driven mechanism to identify materialized views and a specialized concurrency control system on top of a NoSQL database to enable scalable data management with familiar relational conventions. Synergy trades slight write performance degradation and increased disk utilization for faster join performance (compared to standard NoSQL databases) and improved query expressiveness (compared to NewSQL databases). Experimental results using the TPC-W benchmark show that, for a database populated with 1M customers, the Synergy system exhibits a maximum performance improvement of 80.5% as compared to other evaluated systems.

The rest of this chapter is organized as follows. In Section 4.1 we present the background information. Then, in Section 4.2, we motivate the need for a novel view selection

Table 4.1: Qualitative comparison of NoSQL, NewSQL and Synergy systems.

	Scalability	Query Expressiveness	Transaction Support	Disk Utilization
NoSQL (HBase)	Linear scale out	SQL	ACID with Snapshot Transaction Isolation	Moderate
NewSQL (VoltDB)	Linear scale out	SQL with joins limited to partition keys	ACID with Serializable Transaction Isolation	Low
Synergy	Linear scale out	SQL with MVs limited to Key/Foreign-Key joins	ACID with Read Committed Transaction Isolation	High

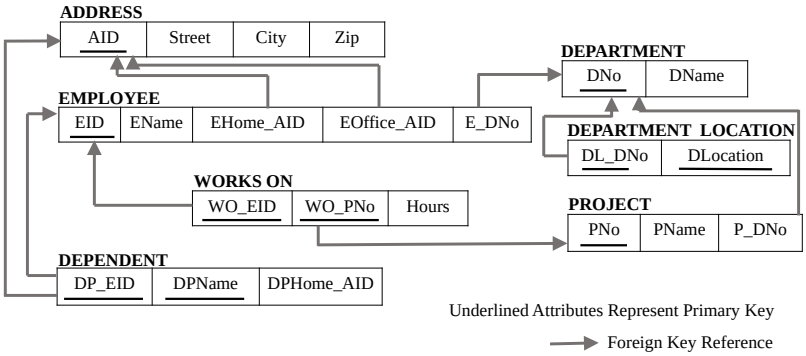


Figure 4.1: Relations in the Company Schema.

mechanism in a key-value store. Section 4.3 presents the Synergy system overview. Section 4.4 presents the candidate views generation mechanism in the Synergy system. Next, in Section 4.5 we describe the views selection procedure in the Synergy system. The views maintenance mechanism for the Synergy system is presented in Section 4.6. Thereafter, Section 4.7 presents the Synergy system architecture. In Section 6.3, we experimentally evaluate Synergy system. Finally, we conclude in Section 4.9.

4.1 Background

We first review the concepts of a relation, index and schema which are common to both SQL and NoSQL data models. Then, we present a model for the database workload. Finally, we describe the mechanism to perform a baseline transformation of a relational database to a NoSQL database. Table 4.1 shows the qualitative comparison of evaluated systems.

4.1.1 Relation, Index and Schema Models

Relation– A relation R is modeled as a set of attributes. The primary key of R denoted as PK(R), is a tuple of attributes that uniquely identify each record in R. The foreign key of R denoted as FK(R), is a set of attributes that reference another relation T. A relation can have multiple foreign keys, hence let F(R) denotes the set of foreign keys of R.

Index– In this work we utilize covered indexes that store the required data in the index itself. An index X on a relation R denoted as X(R), is modeled as a set of attributes (s.t. X(R) $\subset$ R). Let X^{tuple}(R) denotes a tuple of attributes that the index is indexed upon (s.t. X^{tuple}(R) $\subset$ X(R)). The key of an index is a union of attributes in tuples X^{tuple}(R) and PK(R), in that order. Since a relation can have multiple indexes, let I(R) denotes the set of indexes on R.

Schema– Using the previous definitions of a relation and an index, a database schema S is modeled as a set of relations and the corresponding index sets, S = $\{R_1, I(R_1), R_2, I(R_2),..., R_n, I(R_n)\}$, where n represents the number of relations in the schema. We use an example Company database for the purpose of exposition. Figure 4.1 depicts the relations in the Company database schema.

4.1.2 Modeling Workload

A database workload W = $\{w_1,..., w_m\}$ is modeled as a set of SQL statements, where m is the number of statements.

4.1.3 Baseline Database Transformation

In this section, we describe the mechanism to perform a **baseline transformation** from a relational to a NoSQL database.

Baseline Schema Transformation – A relation R becomes a relation R′ in NoSQL schema with the same set of attributes as R. The row key of R′ is a delimited concatenation

of the value of attributes in PK(R). Similarly, an index X(R) on a relation R becomes a relation X(R′) in NoSQL schema with the same set of attributes as X(R). The row key of X(R′) is a delimited concatenation of the value of attributes in the key of X(R). Note that in NoSQL, both for a relation and an index, we assign all attributes to a single column family.

Baseline Workload Transformation – Each read statement from the relational workload is added to the NoSQL workload. Each write statement for a relation R that specifies each key attribute in the WHERE clause is added to the NoSQL workload.

4.2 Challenges and Design Choices

Joins are expensive in a NoSQL database due to the distribution of data items across different cluster nodes. It is well understood that MVs improve join performance by precomputing and storing results in the database [51, 52, 53]. This observation is verified with TPC-W micro-benchmark which shows that scanning a MV is significantly faster than the join performance (see Section 4.8.2 for experiment details). Thus, we consider how to incorporate MVs into a NoSQL store, while ensuring consistency.

4.2.0.1 Implication of Materialized Views

NoSQL databases are generally limited to key-based single row operations [4, 3, 55]. Hence, to ensure the ACID semantics in the presence of MVs, view maintenance and concurrency controls are required to ensure consistency between the MVs and base tables. The design choices for concurrency control mechanisms include multi-versioning, locking and timestamp ordering. While multi-versioning may seem like a nature fit given HBase and other NoSQL system's temporal key component (i.e., cell values are composed of a row-key, column family, column and time stamp) [3, 55], experimental results show that getting and checking additional rows' timestamps decreases performance. Therefore, this result motivates a lock-based concurrency control mechanism to attain the read committed isolation level.

4.2.0.2 Lock Number and Granularity

Row level locks and database locks represent the two ends of the locking mechanism spectrum. Database locks degrade system throughput since only a single transaction can access the database at a time. Similarly, acquiring row level locks on individual base tables can be expensive in the presence of MVs in a NoSQL database, since the system may need to acquire a large number of locks for complex queries. Experimental results show that for a modest number of 100 locks, the time to acquire and release locks is 1.3x the response time of the most expensive write transaction in the proposed system (see Section 4.8.3 and Section 4.8.4.4). This observation motivates minimizing the number of locks required per transaction.

4.2.0.3 View Selection Challenges

The types of MVs that are allowed impact the data store performance in varying ways. Purely workload based MVs selection mechanisms [54] (schema relationships are oblivious) can result in optimal read performance by allowing for the materialization of a maximum number of joins in the workload (i.e., views constructed with many-to-many joins or non-foreign key joins). While this approach is well suited for OLAP workloads, it can degrade write performance and increase disk utilization and transaction management costs for the OLTP workloads, especially in a distributed database. In contrast a schema aware–workload driven MVs selection mechanism limits the type of views allowed, resulting in sub-optimal read performance. However, this approach prevents high storage costs and shifts of the bottleneck from read to the write performance. Given the design goal to hold a single lock per transaction across base tables and MVs, this observation motivates us to not allow views with many-to-many joins or joins that do not have key relationships.

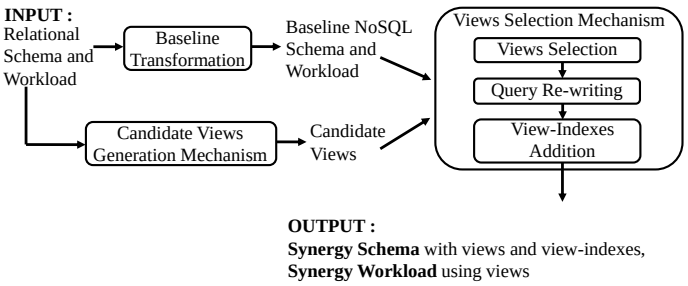


Figure 4.2: Database transformation workflow in Synergy system.

4.2.1 Design Decisions

For the Synergy system, we make the following design decisions based off of our analysis of the TPC-W benchmark, which contains many key/foreign-key equi-joins. First, we develop a concurrency control mechanism that leverages the schema's relational hierarchy, grabs one lock per transaction and provides the read committed isolation level. Second, in cooperation with our concurrency control mechanism, the system only materializes key/foreign-key equi-joins, does not materialize joins across many-many relationships, and each base relation may only be assigned to a single relational hierarchy for materialization (so that a single lock must be acquired per transaction). *We believe the synergistic design decisions between the concurrency control and view selection mechanism provides for a novel architecture and substantially differentiates this work from previous works on materialized view selection.*

4.3 System Overview

In this section we provide an overview of the **Synergy system**, as depicted in Figure 4.2. The objective of our system is to design a scalable and high performance NoSQL database while ensuring the ACID semantics.

We first perform a baseline transformation of the input relational database to a NoSQL database using the mechanism described in Section 4.1.3. Due to the slow join performance in the baseline transformed database system, we decide to use MVs. We use the **candidate**

views generation mechanism to create a list of potential views to materialize based on the database's hierarchical structure. Next, we use a **workload driven view selection mechanism** to select views from the candidate set. Then, we re-write the workload using selected views as needed. To ensure high read performance, we supplement the schema with additional view-indexes. To ensure ACID semantics in the presence of views, we implement a concurrency control layer on top of HBase (as described in Section 4.7), which is able to grab a single lock per transaction, while providing the read committed transaction isolation level.

System Limitations – The Synergy system only materializes key/foreign-key equi-joins. In addition, Synergy system is restricted to single SQL statement transactions. In agreement with our design decision of single lock per transaction, write statements that do not specify all key attributes and affect multiple base table rows are not supported. The Synergy system does not enforce foreign key constraints. The transaction isolation level in the Synergy system is limited to read committed. In addition, the Synergy system can only be used with NoSQL data stores that trade availability for strong consistency in presence of network partition (CP model from the CAP theorem [92]).

4.4 Generating Candidate Views

In this section we present a mechanism to create candidate views for the materialization of equi joins in the workload. We observe that the joins are slow in a NoSQL database (see Section 4.2). Hence, materializing the joins in the workload as views can improve the query performance. We harness the schema's structure to identify the candidate views, in particular the key/foreign-key relationships. We begin by presenting formal definitions for schema relationships and views.

We assume that the input schema S is normalized and free from both simple and transitive circular references, to limit the scope of this work. We model the relationships in S as a directed graph G=(H,E). The vertices in G represent the relations in S and edges encode

the key/foreign-key relationship between relations. An edge exists between relations R_i and R_j, if they are related as described by:

Definition 1 (Schema Relationships) *The relationship between relations R_i and R_j, denoted as $R_i \leftarrow R_j$, exists iff $FK_k(R_i)$ references $PK(R_j)$, where $FK_k(R_i) \in F(R_i)$*

Figure 4.3(a) depicts the schema graph corresponding to the relations in the Company database schema in Figure 4.1. Next, we define an edge and a path in the schema graph.

Definition 2 (Edge in Schema Graph) *A directed edge e_i in a schema graph from a relation R_i to a relation R_j is represented as a (PK,FK) tuple where PK is the primary key of R_i and FK is the foreign key of R_j.*

Definition 3 (Path in Schema Graph) *A path between relations R_i and R_j in a schema graph is modeled as an alternating sequence of relations and directed edges between the relations, $[R_i,e_i,...,e_{j-1},R_j]$. The alternating sequence begins and ends in a relation.*

Database schemas have a hierarchical structure; hence, we can choose a set of relations in the schema graph as roots to create rooted trees. Next, we define a rooted tree.

Definition 4 (Rooted Tree) *A rooted tree T is a directed graph composed of a subset of nodes and edges from the schema graph in which there exists a root node, and unique paths from the root node to each non-root node.*

We use rooted trees to identify the candidate views. Next, we define a candidate view.

Definition 5 (Candidate View) *A candidate view V is a path in a rooted tree. A view is stored physically as a relation. The attributes of V is a set union of attributes of relations in V and the key of V denoted as PK(V) is the key of the last relation in the view. Also, a view-index has the same definition and semantic as a table index.*

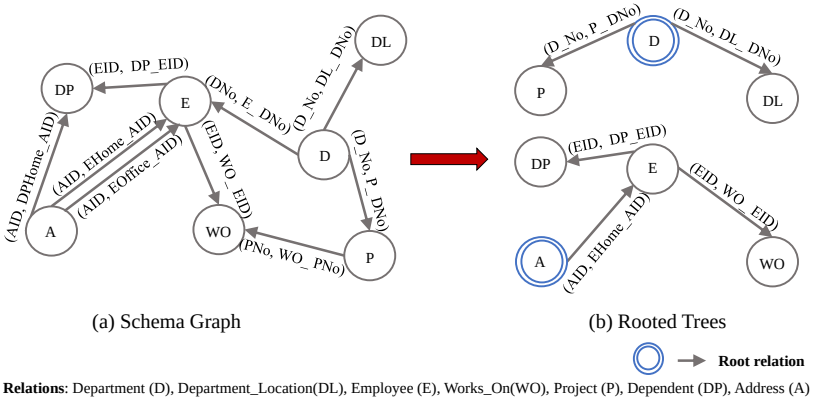


Figure 4.3: Input and output of the candidate views generation mechanism for the Company database with roots set $Q_{company} = \{Address, Department\}$.

4.4.1 Roots Selection

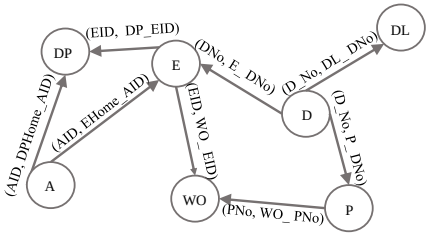
Each view has a single root. The set of roots Q for a schema S is a subset of relations in S. Q can either be provided by the database designer or it can be learned in an automated manner. In this work, we assume that the database designer provides Q. Note that the automated selection of roots is a separate problem and can be addressed independently.

4.4.2 Candidate Views Generation Mechanism

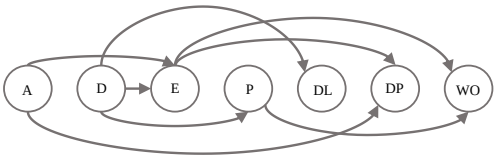
The goals of the candidate views generation mechanism are as follows:

- **Assign each non-root relation in the schema graph to at most one root.** This enables us to hold a single lock on the root relation's row key while ensuring ACID semantics.

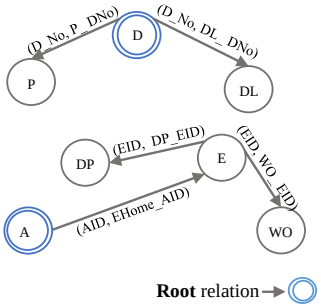
- **Select a single path between the root and each non-root relation assigned to it.** If there are multiple paths between any pair of relations, then the relationship can be one-many or many-many. However, recall that we define many-many relationship as a join materialization boundary. Therefore, to ensure a one-many relationship, we should select a single path.



(a) Schema DAG



(b) A topological ordering of schema DAG



(c) Rooted Graphs

Figure 4.4: Intermediate results of the candidate views generation mechanism for the Company database with roots set $Q_{company} = \{Address, Department\}$. Relations: Department (D), Department_Location (DL), Employee (E), Works_On (WO), Project (P), Dependent (DP), Address (A)

4.4.2.1 Mechanism Overview

In this section, we present the overview of the mechanism to generate the candidate views. We first **transform the input schema graph into a directed acyclic graph** (DAG) to ensure at most one direct path between any pair of relations in the graph. Thereafter, we **identify a topological ordering of the relations** in the schema DAG. Next, we use the topological order to iteratively examine and **assign each non-root relation to a root** by selecting a path from the root to the non-root relation. Following the assignment of schema

relations to roots, a rooted graph is created for each root relation. Finally, we **transform each rooted graph into a rooted tree** to ensure a single path between the root and each non-root relation. The output of the mechanism is a set of rooted trees and each unique path in a rooted tree represents a candidate view.

4.4.2.2 Mechanism Description

In this section, we describe the candidate views generation mechanism in detail using our continuing example of the Company database. We use $Q_{company} = \{Address, Department\}$ as the roots set. In addition, we use a synthetic workload for the purpose of exposition with $W_{Company} = \{w_1, w_2, w_3\}$,

W_1: Get address details of an employee

```
SELECT * FROM Employee as e, Address as a
WHERE a.AID = e.EHome_AID and e.EID = ?
```

W_2: Get all the employees and their hours who work in a department.

```
SELECT *
FROM Department as d, Employee as e, Works_On as wo
WHERE d.DNo = e.E_DNo and e.EID = wo.WO_EID
and d.DNo = ?
```

W_3: Get all the employees who work a certain number of hours.

```
SELECT * FROM Employee as e, Works_On as wo
WHERE e.EID = wo.WO_EID and wo.Hours = ?
```

Heuristic– During the different steps of the mechanism, we use a heuristic based approach to select a candidate from a set. We choose the **number of overlapping joins** as a simple workload aware heuristic to assign a weight to each candidate. Note that other heuristics can be used seamlessly with the mechanism.

Input: Schema graph G, workload W and the roots set Q.

Output: Set of rooted trees.

Steps:

1. **Transform input graph to DAG**: In the first step we transform the input schema graph G into a DAG. We achieve this by selecting and keeping at most one edge between any pair of nodes in the schema graph.

   We use our heuristic to assign a weight to each candidate edge. Then, we select the edge with maximum weight and remove the rest. For example, we remove the (AID, EOffice_AID) edge from the schema graph in Figure 4.3(a) to generate the schema DAG depicted in Figure 4.4a.

2. **Topologically order relations in the DAG**: Next, we identify a linear ordering of the relations in DAG such that for every directed edge from relation R_i to R_j, R_i comes before R_j in the ordering. Figure 4.4b represents a topological ordering of the schema DAG presented in Figure 4.4a.

3. **Assign relations to roots**: Next, in the topological order, we examine each non-root relation in the schema DAG and decide upon its assignment to a root by executing the following steps:

   (a) **Identify paths:** We identify paths in the DAG from each root relation to the non-root relation.

   (b) **Select a path:** Next, we utilize our heuristic to assign a weight to each path. Then, we iterate over the paths in the sorted order by weight until we find a path that includes a single root relation and none of the relations on the path are assigned to a root other than the root present in the path.

   (c) **Add path:** Then, we add the selected path to the rooted graph created for the root in the path.

   Figure 4.4c depicts the rooted graphs generated for the Company database.

4. **Transform rooted graphs to rooted trees**: Next, we transform the rooted graphs created in step 3 into rooted trees. We first identify a topological ordering of the non-root relations in the rooted graph. We repeat the next step while we have relations left in the topological ordering.

   (a) **Select a Path**: Using the rooted graph we identify paths between the root relation and the last relation in the topological ordering. Next, we assign a weight to each path using our heuristic. Then, we select the path with maximum weight and add it to the rooted tree. Thereafter, we remove all non-root relations in the path from the topological ordering and continue.

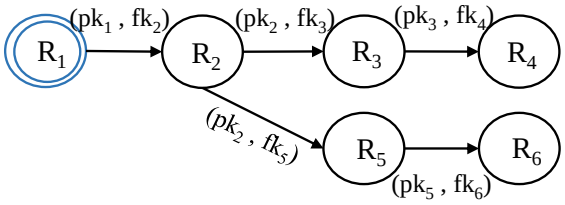
Note that in step 3, we examine non-root relations of a schema DAG in the forward topological order to give each non-root relation a chance to be assigned to any root that has a path to it. Conversely, in step 4, we examine non-root relations of a rooted graph in the reverse topological order to keep the paths that will allow materialization of maximum number of joins in the workload. Following the candidate views generation mechanism, a rooted tree is generated for each root in Q. Figure 4.3(b) depicts the set of rooted trees generated for the Company database.

4.4.2.3 Discussion

The proposed candidate views generation mechanism is a heuristic based approach; hence, does not guarantee materialization of optimal number of joins in the workload. In addition, the usability of generated candidate views for join materialization is dependent on roots selection.

4.5 Views Selection Mechanism

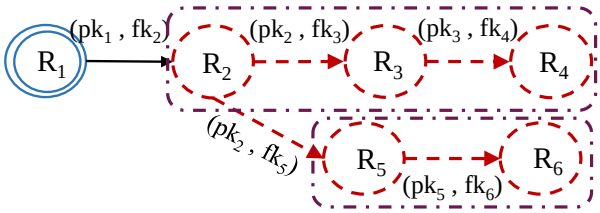
In this section we describe our procedures for views selection from the candidate set and re-writing queries using selected views. Similar to [54], we use a workload driven



(a) Example rooted tree

SELECT *
FROM R_2 , R_3 , R_4 , R_5 , R_6
WHERE $R_2.pk_2 = R_3.fk_3$
and $R_3.pk_3 = R_4.fk_4$
and $R_2.pk_2 = R_5.fk_5$
and $R_5.pk_5 = R_6.fk_6$

(b) Example equi join query



View	Name
$[R_2,(pk_2 , fk_3),R_3,(pk_3 , fk_4),R_4]$	$R_2 - R_3 - R_4$
$[R_5,(pk_5 , fk_6),R_6]$	$R_5 - R_6$

(c) Views selected

SELECT *
FROM R_2–R_3–R_4 , R_5–R_6
WHERE R_2–R_3–R_4 .$pk_2 = R_5$–$R_6.fk_5$

(d) Query re-write using selected views

Figure 4.5: Illustration of view selection and query re-writing procedure for an example equi join query using an example rooted tree.

views selection mechanism. We also illustrate our method for supplementing the schema with additional indexes to ensure query performance.

4.5.1 Views Selection

The high resource requirement and the expensive nature of joins in a NoSQL database (see Section 4.2) provides us with the motivation to materialize as many joins in the workload as possible to ensure low request response times and high system throughput. We use a workload driven approach to select views. We iteratively examine each equi join query in the workload and select views for it. Next, we describe our procedure to select views for a given query.

Views selection for a Query–We harness the rooted trees and the query syntax to select views for a query. To illustrate the procedure, we use the example rooted tree and the example query depicted in Figures 4.5(a) and 4.5(b) respectively. We begin the procedure with un-marked rooted trees. Then, we use the join conditions in the query to mark the relevant edges and participating relations in the rooted trees. Figure 4.5(c) depicts the marked edges and relations in the example rooted tree. Next, we examine each rooted tree to identify the views to be selected for the query.

For a given rooted tree, we iteratively choose a path until no new path can be chosen. During each iteration, path selection is done using two rules: 1) all the nodes and edges in the path are marked, and 2) the path starts in a marked node that has no incoming marked edge and ends in either a leaf node or a node that has no outgoing marked edge. Then, we select the chosen path as a view. Next, we un-mark the participating relations of the path and outgoing edges of the participating relations, in the rooted tree. Thereafter, we continue with the next iteration. Figure 4.5(c) depicts the views selected for the example query.

Final View Set– After processing the entire workload, we add the set of all selected views to the schema.

Limitations– 1) We select views only for the equi join queries in the workload. 2) Searching the space of all syntactically relevant views is not feasible in practice [54]; hence, similar to [54], our views selection procedure is heuristic based and does not necessarily select the optimal set of views. 3) A views selection procedure that can take advantage of view sharing opportunities across different queries is part of our future work. 4) Currently we do not pass a storage constraint to our views selection algorithm; however, it can be easily adapted to use storage constraint in presence of a cost based query optimizer.

4.5.2 Query Re-writing

Following views selection, we re-write queries using selected views. We iteratively examine each equi join query in the workload and re-write it using the views selected for it. To re-write a query, we replace the constituent relations of a view with the view. In addition, we remove the join conditions for which both participating relations belong to a single view. Figure 4.5(d) depicts the example query re-written using selected views.

4.5.3 Additional View Indexes

Unfortunately, in certain scenarios query execution times can be high despite the use of views. Consider a case in which the query using the view has a filter on an attribute other than the attribute that the view is indexed upon. Then, to prepare the query response, we have to scan the entire view. This can be expensive, depending on the size of the view. Hence, to improve the performance of workload queries that use views, we supplement the schema with additional indexes.

For each view, we examine each conjunctive query that uses this view and decide whether to add a view-index or not. If the query only has filters on one or more view attributes that neither the view nor any of its indexes are indexed upon, then we add a view-index indexed upon a filter attribute to the schema. Note that in this work we do not recommend indexes on base tables and assume that the input schema has necessary base table indexes.

4.6 View Maintenance Mechanism

In this section, we describe the mechanism for view maintenance as the underlying base tables are updated. For each type of write statement we present: 1) an applicability test to determine if a base table update applies to a view and 2) a tuple construction procedure to prepare tuples for the view update upon a base table update.

4.6.1 Insert Statement

4.6.1.1 Applicability Test

A base table *insert* for a relation R_i applies to a view V_i iff R_i is the last relation in V_i's sequence of relations.

4.6.1.2 Tuple Construction

Insertion into a view upon a base table insert may require reading tuples from the base tables to construct the view tuple. For a base table insert that applies to a view with k relations, we need to read related tuples from $k-1$ base tables to construct the view tuple. We utilize the key/foreign-key relationships between view relations to sequentially read the base table tuples, starting with relation R_{k-1} and ending in relation R_1. Then, we construct the view tuple using previously read tuples and the insert statement. Notice that the time to create a view tuple increases linearly with the number of relations in the view and is independent of the cardinality ratios between the relations.

4.6.2 Delete Statement

4.6.2.1 Applicability Test

A base table *delete* for a relation R_i applies to a view V_i iff R_i is the last relation in V_i. Note that we do not perform cascading deletes.

4.6.2.2 Key Construction

To delete a view tuple upon a base table *delete*, we use the base table key provided with the delete statement. However, to delete the view index tuple, we need to first construct the index key to issue a delete upon. Hence, we first read the tuple from the view using the base table key in the delete statement. Then, we use the attributes in the read tuple to

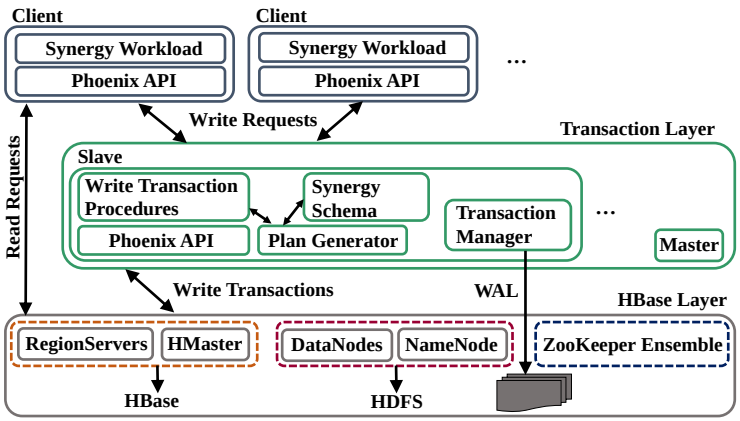


Figure 4.6: Synergy System Architecture Overview.

construct the index key and issue the delete. Notice that the time to construct a view index key is constant.

4.6.3 Update Statement

4.6.3.1 Applicability Test

A base table *update* for a relation R_i applies to a view V_i iff R_i is in V_i's sequence of relations.

4.6.3.2 Tuple Construction

Unfortunately, updating the view upon a base table update can be expensive if the view is not indexed on the key of the update statement, since we need to either join the base tables or scan the entire view for the tuple construction. To efficiently prepare view updates, we supplement the schema with additional indexes based on the workload. Due to space concerns, we omit the details.

4.7 System Architecture

In this section we describe the Synergy system architecture. The Synergy system comprises of **HBase layer**, **clients** and the **Transaction layer** as depicted in Figure 4.6.

HBase layer– The Synergy system harnesses HBase layer as the distributed data storage substrate. The HBase layer comprises of HBase, HDFS and ZooKeeper components. We refer the reader to [4] for the role and description of each component shown in Figure 4.6.

Clients– The clients utilize Phoenix API to execute read and write statements in the workload. A client sends a read request directly to the HBase layer. On the contrary, a write request is sent to the Transaction layer, followed by a synchronous wait for a response.

Transaction Layer– The Synergy system employs the Transaction layer for implementing ACID transaction support on top of the HBase layer. The Transaction layer is a distributed, scalable and fault tolerant layer that comprises of a *Master* node and one or more *Slave* nodes. The *Slave* nodes receive and process *write requests* from clients. Each slave node has a transaction manager that implements a write ahead log (WAL) for recovery and durability. The WAL is stored in HDFS. Upon receiving a request, the transaction manager first assigns a transaction *id* to the statement and then appends the statement in WAL along with the assigned *id*. Then, a transaction procedure utilizes Phoenix API to execute the transaction. Finally, a response is sent back to the client. The *Master* node is responsible for detecting slave node failures and starting a new slave node to take over and replay the WAL of a failed slave node.

4.7.1 Lock Implementation

Logical Locking– Recall that we restrict the write workload to statements that specify all key attributes (see Section 4.3) and decide to employ hierarchical locking as the concurrency control mechanism (see Section 4.2). Hence, to update a row for a relation in a rooted tree, we acquire the lock on the key of the associated row in the root relation. In addition, since each relation is part of at most one rooted tree, we hold a single lock per write operation.

Physical Locking– We implement our locking mechanism through lock tables stored

in HBase. We create one lock table per root relation. The lock table key has same set of attributes as the root relation's key and it includes a single boolean column that identifies if lock is in use or not. A lock table entry is created when a tuple is inserted into the root table.

Discussion– We implement light weight hierarchical locking mechanism in Synergy by holding a single lock per write operation. As a downside of hierarchical locking, all rows associated with the root key along all the paths are locked which can affect throughput with concurrent requests trying to grab the lock on the same root key. Note that lock management is not the primary contribution of our work. Other transaction management systems like Themis [93], Tephra [38], Omid [94] etc. could also be used.

4.7.2 Write Transaction Procedures

Synergy utilizes transaction procedures to atomically update the base table, views and corresponding indexes upon a base table update. For **insert** and **delete** statements, the transaction procedure first acquires the lock on the root key. Then, the base table, applicable views and corresponding indexes are updated using the tuple/key construction procedures described in Section 4.6. Finally, the lock is released. Note that each transaction inserts/deletes a single row in/from the base table, applicable views and corresponding indexes.

A base table update may require multi-row updates on the materialized view. Now, while a view is being updated upon a base table update, conflict with the concurrent writes is prevented by the locking mechanism; however, a concurrent read may read dirty data. Hence, to facilitate the detection of a dirty read, we mark the data in views and view-indexes before update and un-mark after update. If dirty data is read in a transaction, then the read is restarted. The **update** transaction is a 6-step procedure: 1) We first acquire a lock on the root key. 2) Then, we read all the rows that need to be updated. 3) Next, we mark all the rows that need to be updated. 4) Then, we issue a sequence of updates. 5)

Next, we un-mark all the updated rows. 6) Finally, we release the lock.

The plan generator component (see Figure 4.6) in the Synergy transaction layer auto generates the execution plan for each write transaction.

4.7.3 Transaction Isolation Level

The Synergy system is restricted to single statement transactions. In addition, Synergy does not support queries in which a relation is used more than once due to potential dirty reads. The Synergy system provides **ACID semantics** with **read committed** transaction isolation level, which is also the default transaction isolation level for PostgreSQL [95].

A single row is inserted/deleted into/from the base table, applicable views and corresponding indexes upon a base table **insert/delete**. In addition, to answer a query either a table is used directly or a view involving the table is used but not both. Hence, a reader either reads the entire row or the row is absent from the read result set. This enables read committed behavior for insert and delete statements.

Recall that the system marks the rows to be updated in a view as dirty before issuing updates, and if a concurrent *scan* reads a dirty row, the scan is restarted. Hence we modify the scan behavior to check for marked rows in the scanned result-set and **re-scan** if a marked row is present. This ensures that **update** statement preserves the read committed semantics.

Note that the **read committed** semantics are preserved during a **failure scenario**, since the base table lock is held until the system recovers from the failure.

4.8 Experimental Evaluation

In this section we first describe our experiment environment. Next, we use a TPC-W micro-benchmark to evaluate the join performance in HBase. Thereafter, we profile the performance overhead of two phase row locking in HBase. Finally, we evaluate the

Figure 4.7: Micro benchmark schema graph.

performance of Synergy system and compare it with four other systems using the full TPC-W benchmark.

4.8.1 Experiment Environment

4.8.1.1 Testbed

Amazon EC2 represents our experiment environment. We create an eight node cluster using m4.4xlarge virtual machine (VM) instances. Each instance is configured with 16 vCPU's, 64GB RAM and 120 GB SSD elastic block storage (EBS), running Ubuntu 14.04.

HBase, HDFS and Zookeeper: The HDFS NameNode, the HBase HMaster, and the ZooKeeper server processes run on one instance. We designate five instances as slaves, each running the HDFS DataNode and the HBase RegionServer processes. We use Hadoop v2.6.5, HBase v1.2.4.

Synergy and Phoenix: We dedicate one instance to host a Synergy transaction layer slave and the Phoenix-Tephra server. Synergy transaction layer master is hosted on the same node that hosts HBase and HDFS masters. We use Phoenix v4.8.2.

VoltDB: We create a five instance VoltDB (v6.8) cluster by hosting a VoltDB daemon on each instance that is also hosting the HDFS DataNode and the HBase RegionServer processes.

Client: We reserve one node as client to drive the workload for each system.

4.8.1.2 Performance Metric

The request response time represents our performance metric, denoted as τ. We measure τ in the client.

Q1: Get all the customers and their orders
Using base tables,
SELECT *
FROM Customer as c, Order as o
WHERE c.c_id = o.o_c_id
Using view,
SELECT * FROM Customer-Order

Q2: Get all the customers, their orders and the constituting order lines.
Using base tables,
SELECT *
FROM Customer as c, Order as o, Order line as ol
WHERE c.c_id = o.o_c_id and o.o_id = ol.ol_o_id
Using view,
SELECT * FROM Customer-Order-Order line

Figure 4.8: Micro-Benchmark Workload.

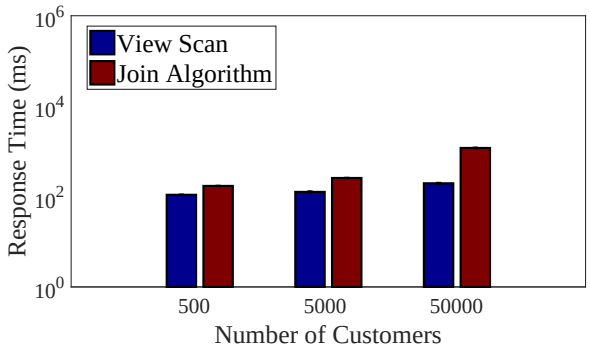
4.8.2 Micro Benchmark Evaluation

We use a TPC-W micro-benchmark to evaluate the join performance in HBase.

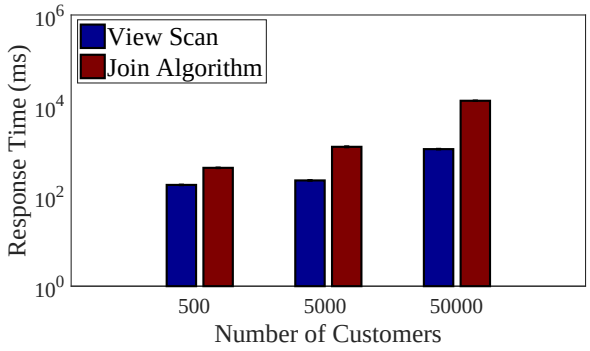
4.8.2.1 Schema and Workload

The micro benchmark schema comprises of three relations from the TPC-W benchmark: Customer, Order and Order_line. Customers can have one or more orders and each order can have one or more order lines. Figure 4.7 depicts the schema graph for the benchmark schema. Next, to evaluate the join performance, we create a synthetic workload comprising of two foreign key equi-join queries: Q1 (*Customer,Order*) and Q2 (*Customer,Order,Order_line*).

A join query can be evaluated using two different approaches: 1) using a join algorithm that combines the matching tuples from the specified tables and 2) scanning pre-computed and stored results from a materialized view. Hence, to compare the join algorithm performance with the view scan performance, we materialize the joins in the workload as views. Customer-Order and Customer-Order-Order_line represent the MVs corresponding to the



(a) Q1



(b) Q2

Figure 4.9: Micro benchmark results to show that performance of join algorithms is slow in HBase. Y axis is drawn at log scale.

join queries Q1 and Q2 respectively. Figure 4.8 presents the workload queries written using base tables and MVs.

4.8.2.2 Experiment Setup and Results

Each experiment is characterized by the database scale and the join query. We set the cardinality ratio between relations as 1:10. We scale the database by increasing the number of customers in multiples of 10, starting at 500. For each database scale, we major compact both base tables and views after database population. Section 4.8.2.1 presents the join queries in the workload. We repeat each experiment 10 times and report the mean and the standard error of response time. Figure 4.9 depicts the experiment results with Y axis drawn at log scale.

	Number of Locks		
	10	**100**	**1000**
Overhead (in ms)	342	571	2182

Figure 4.10: Experiment to show overhead associated with two phase row locking in HBase.

For the database populated with 50K customers, view scan is 6x and 11.7x faster than the join algorithm for queries Q1 and Q2 respectively. ***In conclusion, micro-benchmark results show that the join algorithm performance is slow in HBase, providing the motivation for join materialization.***

4.8.3 Locking Overhead Evaluation

In this experiment, our goal is to evaluate the performance overhead of acquiring and releasing row locks in HBase. We create a single lock table in the HBase layer with two attributes: *id* and *lock_status*. The *lock_status* is a boolean column that identifies whether lock is in use or not. We use *checkAndPut* HBase operation in the client node to acquire and release locks. We increase the number of locks in multiples of 10 starting at 10 and measure the overhead in client. We repeat each experiment 10 times and present the mean overhead time. Figure 4.10 shows the experiment results.

Locking overhead with 100 locks is 1.3x the response time of statement W13 in the Synergy system (see Section 4.8.4.4 and Figure 4.13); W13 represents the most expensive write transaction in the Synergy system. Also note that 100 represents a modest number of locks for a write transaction, since multiple tables with varying cardinalities may be joined together as a view. ***In conclusion, overhead associated with the acquisition and release of row locks represents a major transaction performance bottleneck in HBase, motivating the use of a single lock per transaction.***

Mechanisms	Systems				
	VoltDB	Synergy	MVCC-A	MVCC-UA	Baseline
Materialized Views Selection	None	Schema Relationships Aware	Schema Relationships Aware	Schema Relationships Un-Aware	None
Concurrency Control	Single Threaded Partition Processing	Hierarchical Locking	MVCC	MVCC	MVCC

Figure 4.11: Materialized views selection mechanism and concurrency control mechanism used in each evaluated system.

4.8.4 TPC-W Benchmark Evaluation

4.8.4.1 Benchmark

TPC-W [86] is a transactional web benchmark. It has a two tier architecture including a web tier and a database tier. TPC-W workload includes 14 different types of web requests where each request is modeled as a servlet. Each servlet is in turn composed of one or more SQL statements. We analyze the TPC-W servlets to extract all the SQL statements that can be invoked at the database tier. Extracted set of SQL statements represents our workload.

We exclude a DELETE statement (DELETE FROM *shopping_cart_line* WHERE *scl_sc_id* = ?) from the workload that may affect multiple base table rows. Phoenix currently does not provide an implementation of the *soundex* algorithm; hence, we exclude two join queries from the workload that use *soundex* algorithm.

The database size (DB_{size}) can be modulated by varying two parameters: the number of customers (NUM_CUST) and the number of items (NUM_ITEMS). We set NUM_ITEMS to 10 * NUM_CUST. In addition, we change the cardinality between the Customer and the Orders table from .9 to 10. We populate the database with 1 million customers. For each system that utilizes HBase as the storage layer, we major compact base tables, indexes and MVs after the database population.

4.8.4.2 Systems Evaluated

Synergy: We use Q_{TPC-W} = {*Author, Customer, Country*} as the roots set to generate views in the Synergy system. We create base tables, selected views and corresponding indexes in HBase. In addition, we create lock tables for each root in Q_{TPC-W} . We disable the Phoenix-Tephra transaction support.

MVCC-UA: To compare our views generation and selection mechanism with [54], we deploy SQL Server 2012 on a single EC2 VM instance. Next, we populate the TPC-W database with 1 million customers and run the TPC-W benchmark queries. Then, we use the SQL Server's database engine tuning advisor to analyze the profiled workload and generate views. We create the generated views along with base tables and indexes in HBase and run the workload with Phoenix-Tephra transaction support (MVCC) enabled.

MVCC-A: In addition to the base tables and indexes, we create the views and the view-indexes generated by the Synergy system in HBase and run the workload with Phoenix-Tephra transaction support (MVCC) instead of the specialized transaction support used in Synergy.

Baseline: We only create base tables and corresponding indexes in HBase and run the workload with Phoenix-Tephra transaction support (MVCC).

VoltDB: A VoltDB table can either be partitioned or replicated. The partitioning column is specified by the user and partitioned tables can only be joined on equality of partitioning column. Now, a table can join with other tables using different columns in different queries of the workload; however, since each table can only be partitioned on a single column, only a subset of workload join queries may work for a partitioning scheme.

To profile the performance of maximum number of joins in the TPC-W benchmark we use **three different partitioning schemes in VoltDB**. However, note that in practice only one partitioning scheme could be used for a database. Also, note that only base tables and corresponding indexes are used in VoltDB.

Figure 4.11 summarizes the MVs creation and concurrency control mechanisms used

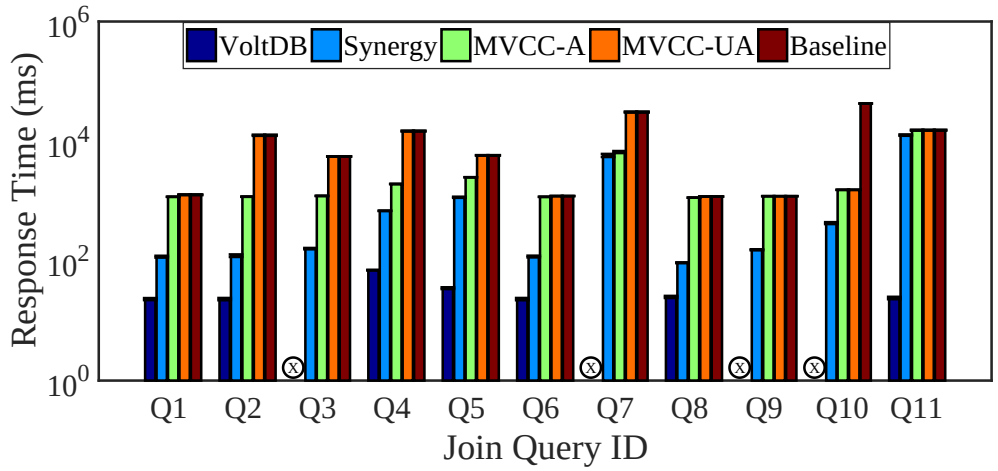


Figure 4.12: Evaluation and comparison of join performance across different systems using join queries in the TPC-W benchmark. Y axis is drawn at log scale. Join queries $\{Q_3, Q_7, Q_9, Q_{10}\}$ are not supported in VoltDB.

in each evaluated system.

4.8.4.3 Performance Evaluation of Joins in the TPC-W Benchmark

Experiment setup– In this set of experiments, we evaluate and compare the join performance across different systems using the join queries in the TPC-W benchmark. Recall that we used three different partitioning schemes in VoltDB to support maximum number of TPC-W joins, using any single partitioning scheme less than 50% of the TPC-W joins are supported.

We evaluate each query10 times and present the mean and the standard error of the recorded response times. See Table 4.4 for the specification of join queries in the TPC-W benchmark. Figure 4.12 presents the experiment results. Note that join queries $\{Q_3, Q_7, Q_9, Q_{10}\}$ are not supported in VoltDB.

Discussion– On an average the join queries in Synergy are 19.5x, 6.2x and 28.2x faster as compared to the MVCC-UA, MVCC-A and Baseline system respectively. The view selection mechanism in the Synergy system selects more MVs as compared to MVCC-UA, resulting in significantly larger join performance benefit. In MVCC-UA, the response time

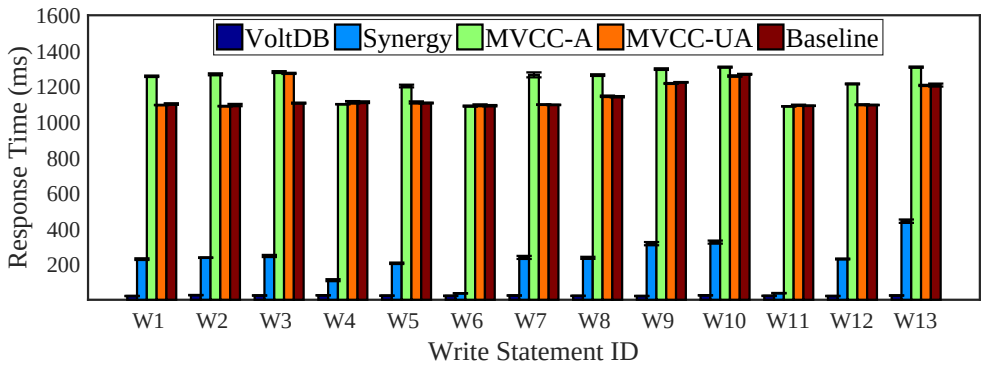


Figure 4.13: Performance Evaluation of the write statements in the TPC-W benchmark to exhibit the overhead of lock management and updating views in the Synergy system. Comparison of write statement performance across different systems.

of Q_{10} is significantly lower than the Baseline system since MVCC-UA utilizes a materialized view for query evaluation. The join performance in Synergy system with specialized concurrency control is marginally better than MVCC-A that uses MVCC. The join queries that used views in Synergy are on an average 11x slower than VoltDB (excluding queries that are not supported in VoltDB). ***In conclusion, join response times in the Synergy system with selected views are significantly lower as compared to MVCC-UA and Baseline system for the benchmark queries. In addition, although the Synergy join performance is slower than VoltDB, Synergy allows for significantly more expressive joins than VoltDB.***

4.8.4.4 Performance Evaluation of Write Statements in the TPC-W Benchmark

Experiment setup–In this set of experiments, we aim to evaluate the performance overhead of acquiring/releasing a lock and updating MVs in the Synergy system. In addition, we compare the write statement performance across different systems using the write statements in the TPC-W benchmark. See Table 4.5 for the specification of write statements in the TPC-W benchmark. We evaluate each statement 10 times and present the mean and the standard error of the recorded response times. Figure 4.13 presents our experiment results.

Discussion–On an average the write statements in Synergy are 9x, 8.6x and 8.6x less

expensive than MVCC-UA, MVCC-A and Baseline system respectively. In Synergy system, the execution time of statements W6 and W11 is significantly lower than the other write statements since the corresponding relation is not part of any views. Although Baseline system does not use any MVs and MVCC-UA utilizes only one materialized view, the statement response times in these systems are high since MVCC adds an overhead of 800-900 ms to each statement's execution time. On an average the write statements in Synergy are 9.4x more expensive than the VoltDB. ***In conclusion, experimental results show that the use of hierarchical locking in Synergy system significantly reduces the write transaction response times in presence of MVs.***

Table 4.2: Sum of RT of all the statements in the TPC-W benchmark to quantify trade off between read performance gain and write performance overhead of using MVs in each evaluated system. VoltDB is excluded since it does not support all queries in the benchmark.

	Evaluated Systems			
	Synergy	**MVCC-A**	**MVCC-UA**	**Baseline**
Mean Response Time (in seconds)	33.7	77.4	132.4	173.4
Standard Error	.03	.02	.06	.07

Table 4.3: Database sizes across different evaluated systems.

No. of Customers	**Database Size (in GB)**				
	VoltDB	**Synergy**	**MVCC-A**	**MVCC-UA**	**Baseline**
1M	31.8	92	91.8	45.73	43.8

4.8.4.5 Performance Comparison of All Evaluated Systems

Experiment setup–In this set of experiments, we evaluate the performance gain and the storage overhead of using MVs in the Synergy system and compare it with the other systems. Note that we exclude VoltDB since it does not support all join queries in the TPC-W benchmark. We evaluate the performance of systems using all the statements in the TPC-W benchmark.

During an experiment, we run each benchmark SQL statement and record its response time. Next, we compute the sum of response time of all statements. We run each experiment 10 times and present mean and standard error of the benchmark response time. Table 4.2 presents the experiment results. Table 4.3 summarizes the database sizes across different systems.

Discussion– Synergy system exhibits a performance improvement of 74.5%, 56.3% and 80.5% as compared to the MVCC-UA, MVCC-A and Baseline system respectively. Conversely, the database size in the Synergy system is 2x, 1x and 2.1x the database size in the MVCC-UA, MVCC-A and Baseline system respectively. Hence, Synergy system trades slight write performance degradation and increased disk utilization for faster join performance. ***In conclusion, the specialized concurrency control mechanism and the MVs generation mechanism in the Synergy system significantly improve the read performance without shifting the bottleneck to the write performance.***

Table 4.4: Specification of joins in the TPC-W Benchmark.

Join ID	Tables	Filters	Order By	Group By	Limit Clause
Q1	Item, Order_line	ol_o_id	None	None	None
Q2	Customer, Orders	c_uname	o_date, o_id	None	1
Q3	Customer, Address, Country	c_name	None	None	None
Q4	Author, Item	i_subject	i_title	None	50
Q5	Author, Item	i_subject	i_pub_date, i_title	None	50
Q6	Author, Item	i_id	None	None	None
Q7	Orders, Customer, Address as ship_addr, Address as bill_addr, Country as ship_co, Country as bill_co	o_id	None	None	None
Q8	Item, Shopping_cart_line	scl_sc_id	None	None	None
Q9	Item as I, Item as J	i_id	None	None	None
Q10	Author, Item, Order_line, Orders tmp table	i_subject	ol_qty	i_id	50
Q11	Order_line as ol, Order_line as ol2, Orders tmp table	ol.ol_i_id, ol2.ol_i_id <>	ol_qty	ol_i_id	5

Table 4.5: Specification of write statements in TPC-W Benchmark.

ID	Statement	ID	Statement	ID	Statement	ID	Statement
W1	Insert Orders	W2	Insert CC Xacts	W3	Insert Order line	W4	Insert Customer
W5	Insert Address	W6	Insert shopping cart	W7	Insert shopping cart line	W8	Delete shopping cart line
W9	Update Item1	W10	Update Item2	W11	Update shopping cart	W12	Update shoppping cart line
W13	Update customer						

4.9 Chapter Summary

In this chapter we present the Synergy system, a data store that leverages schema based–workload driven materialized views and a specialized concurrency control system on top of a NoSQL database that allows for scalable data management with familiar relational conventions. Synergy trades slight write performance degradation and increased disk utilization for faster join performance (compared to standard NoSQL databases) and improved query expressiveness (compared to NewSQL databases). Experiment results on a lab cluster using the TPC-W benchmark show the efficacy of our system.

Chapter 5

A comparative analysis of state-of-the-art SQL-on-Hadoop systems for interactive analytics

Published in IEEE BigData Conference 2017.

Hadoop is emerging as the primary data hub in enterprises, and SQL represents the de facto language for data analysis. This combination has led to the development of a variety of SQL-on-Hadoop systems in use today. While the various SQL-on-Hadoop systems target the same class of analytical workloads, their different architectures, design decisions and implementations impact query performance. In this chapter, we perform a comparative analysis of four state-of-the-art SQL-on-Hadoop systems (Impala, Drill, Spark SQL and Phoenix) using the Web Data Analytics micro benchmark and the TPC-H benchmark on the Amazon EC2 cloud platform. The TPC-H experiment results show that, although Impala outperforms other systems (4.41x – 6.65x) in the text format, trade-offs exists in the parquet format, with each system performing best on subsets of queries. A comprehensive analysis of execution profiles expands upon the performance results to provide insights into performance variations, performance bottlenecks and query execution characteristics.

The rest of this chapter is organized as follows. In Section 5.1 we present the background information. Then, in Section 5.2, we describe our study goals. Section 5.3 utilizes the WDA micro benchmark to evaluate and compare the performance of chosen systems. Next, in Section 5.4, we use the TPC-H benchmark to thoroughly evaluate query expressiveness, optimizer quality and query execution engine efficiency in the examined systems. Finally, we conclude in Section 5.5.

Table 5.1: Qualitative comparison of evaluated SQL-on-Hadoop systems.

System	Query Optimizer	Execution Model	Fault Tolerance	Execution Runtime	Schema Requirements
Impala	Cost Based Optimizer that attempts to minimize network transfer	Volcano Batch-at-a-Time, Pipelined Execution, Runtime Code Generation	Requires Query Restart	MPP Engine with Long Running Daemons	Upfront definition required
Spark SQL	Extensible Catalyst Optimizer with Cost and Rule based optimization	Pipelined Execution, Whole Stage Code Generation, Vectorized Processing	Lineage based RDD Transformations	Spark Engine	Reflection and Case classes based schema inference
Drill	Apache Calcite derived Cost Based Optimizer	Vectorized Processing, Pipelined Execution, Runtime Code Compilation	Requires Query Restart	MPP Engine with Long Running Daemons	Schemaless Query Support
P-HBase	Apache Calcite derived Cost Based Optimizer	Blocking Execution	Requires Query Restart	Client Coordinated Parallel HBase Scans Based	Upfront definition required

5.1 Background

In this section we elucidate on how we utilize the profiling information exposed by each evaluated system. Table 5.1 presents a qualitative comparison of the evaluated systems.

IMPALA and **DRILL.** The profiler provides an execution summary for the scan, join, aggregation, data-exchange and sort operators present in a query execution plan. Note, the scan operator includes the time to scan, filter, and project tuples from a table. Also, the data-exchange operator includes the time to transfer the data over the network; however, the data de/serialization time is not summarized by the profiler.

SPARK SQL. The profiler generates detailed statistics for each execution stage in the query DAG. For each stage, we summarize the task data to calculate average values for scheduling delay, GC time, shuffle-read time, shuffle-write time and executor-computing time. We map the query execution plan operators to the query DAG stages. Note, multiple operators (with pipelined execution), such as join and partial-aggregation, final-aggregation and sort, scan and partial-aggregation, may be mapped to a single DAG stage. The executor-computing time for a stage is credited as the processing time for the operator/s mapped to that stage. The Spark runtime performs I/O in the background while a task is computing, and shuffle-read represents the time for which a task is blocked reading the data over the network from another node [96]. Hence, the shuffle-read time for a stage is attributed as the processing time for the corresponding data-exchange operator/s in the query plan. Also, the shuffle-write time for a stage is assigned to the data serialization overhead.

In this study, we utilize the average operator time in each evaluated system for analysis. In addition, we could not use operator level execution time break-down in **PHOENIX** since currently it does not record execution statistics.

We acknowledge the variance in profiling information exposed by evaluated systems; however, despite the differences, we are able to gain significant insight into the performance characteristics of evaluated systems.

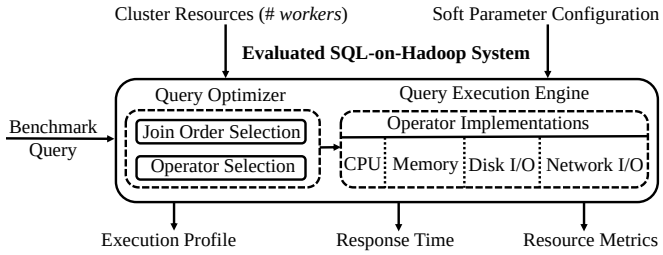


Figure 5.1: Query execution model in each evaluated system.

5.2 Experiment Goals

In this section we describe our study goals. The query performance in a SQL-on-Hadoop system is dependent both on the quality of the execution plan generated by the query optimizer and the efficient execution of the generated plan by the query engine, as shown in Figure 5.1. The optimizer generates a query execution plan by evaluating different join orders and selecting physical operators for the relational algebra operations in the chosen query plan. The query execution engine utilizes operator implementations to carry out the generated plan and produce the output RS. An operator may use one or more sub-systems (CPU, Disk, Network and Memory) in the execution engine to perform its task. The query RT represents our performance metric. We collect resource utilization metrics and query execution profiles in each system.

In this study, our goal is to evaluate and understand the characteristics of two main components of a SQL-on-Hadoop system (query optimizer and query execution engine) and their impact on the query performance. For the query optimizer, our objective is to characterize the execution plan generation, the join order selection and the operator selection in each evaluated system. To this end, we analyze and compare the generated execution plans in each system and understand their impact on the query performance. For the query execution engine, we aim to evaluate the efficiency of operator implementations in each system and identify the performance bottlenecks. To this end, we utilize the query execution profiles to extract the operator processing times. In each system, we aggregate the

processing times for each operator type to understand the contribution of each operator type to the query RT. In addition, we compare the total benchmark operator processing times between evaluated systems to identify if a sub-system in an execution engine is a performance bottleneck.

We perform experiments along three dimensions – *storage format, scale-up* and *size-up*. We examine the impact of text (row-wise) and parquet (columnar) storage formats on the query performance in each system. To understand the *size-up* characteristic in each system, we increase the data size in a cluster and examine the query performance changes. We evaluate the *scale-up* behavior in each system by proportionally increasing both the cluster and the data size. Note that we use the *scale-up* and *size-up* definitions specified in [97].

5.3 Micro Benchmark Experiments

In this section we utilize the WDA micro benchmark proposed in [70] to evaluate and compare the performance of Impala, Drill, Spark SQL and P-HBase systems.

5.3.1 Hardware Configuration

We harness Amazon EC2 cloud environment to setup experiment testbeds. Our testbeds comprise of "*worker*" VMs of r3.4xlarge instance type, a "*client*" VM of r3.4xlarge instance type and a "*master*" VM of d2.xlarge instance type. A r3.4xlarge VM instance is configured with 16 vCPUs, 122GB RAM, 320GB SSD instance storage and 120GB EBS storage and a d2.xlarge VM instance is configured with 4 vCPUs, 30.5GB RAM, 3x 2000GB HDD instance storage and 120GB EBS storage. Note, we choose r3.4xlarge instance for the *worker* nodes since evaluated systems are optimized for in-memory execution and the RAM size in a r3.4xlarge instance is large enough to hold the intermediate result-sets of all workload queries in-memory for the largest evaluated DB size.

5.3.2 Software Configuration

We deploy HDFS, HBase, Drill, Impala, Spark, Yarn, Hive and Zookeeper frameworks on each cluster. We host the control processes (e.g. HMaster, NameNode etc.) from each framework on the *master* VM. We also use *master* VM as the landing node for the data generation. Worker processes from each framework (e.g. DataNode, Drillbit etc.) are hosted on each *worker* VM in the cluster. We reserve the *client* VM to drive the workload. We deploy the Phoenix JDBC driver in the *client* VM and put relevant jars on the classpath of each Region Server. We use the Cloudera Distribution v5.8.0, Impala v2.6.0, Spark v2.0, Drill v1.8.0, HBase v1.2.0 and Phoenix v4.8.0.

Fine tuning of soft parameters is quintessential to getting the best performance out of a software system. To identify the values for performance knobs that ensure a fair comparison between evaluated systems, we rely on: 1) Settings used in prior studies (e.g. [74, 72]). 2) Best practice guidelines from industry experts and system developers (e.g. [98]). 3) Experimentation with different values that enable us to run all benchmark queries and achieve the best performance.

For HDFS, we enable short-circuit reads, set the block size to 256MB and set the replication to 3. We enable Yarn as the resource manager for Spark and set *spark.executor.cores* to 8, *spark.executor.memory* to 8GB and *spark.memory.offheap* to 16GB. We set the heap size to 20GB in each Region-Server. We assign 95GB to each worker process in Impala and Drill.

5.3.3 Experiment Setup

We evaluate systems one at a time. During the evaluation of a system, we stop the processes for other systems to prevent any interference. We disable HDFS caching in each system. We execute benchmark queries using a closed workload model, where a new query is issued after receiving the response for the previous query. We run each query five times

and report the average of the query RT for the last four runs with a warm OS cache. We use a JDBC driver to run the workload in P-HBase and shell scripts to issue queries in Spark SQL, Drill and Impala. Similar to [70], we write query output to HDFS in Spark SQL and to local file system in Impala, Drill and P-HBase. We use *collectl* linux utility to record resource utilization on each *worker* node with a 3 second interval. We export the query execution profile in all systems except P-HBase since it currently does not record execution statistics.

5.3.4 WDA Benchmark

The benchmark schema comprises of two relations (UserVisits and Rankings) that model the log files of HTTP server traffic. The benchmark workload comprises of simple tasks (selection query, aggregation query, and join query) related to HTML document processing. Note, similar to [73], we choose the aggregation task variant that groups records by the source IP address prefix. In addition, we remove the UDF task from the workload since it could not be implemented in each evaluated system. We use the data generator utility provided with the WDA benchmark to generate 20GB of UserVisits data and 1GB of Rankings data per *worker* node (same as in [70]). We refer reader to [70] for detailed benchmark description.

Table 5.2: Data preparation times (seconds) in evaluated systems for WDA benchmark.

# of Worker Nodes	**Insert HDFS**	**Impala**			**P-HBase**			**Drill and Spark SQL**
		Load Tables	**Compute Stats**	**Total**	**Load Tables**	**Major Compact**	**Total**	**Total**
2	280	10.93	283.97	574.9	6748	1590	8618	280
4	550	9.5	291.4	850.9	7027	1722	9299	550
8	1124	8.6	295.2	1427.8	9088	1559	11771	1124

5.3.5 Data Preparation

We evaluate each system with the data stored in the text format to ensure storage format parity across systems. In each system, we first load the text data from the landing node into

HDFS. Drill and Spark SQL are capable of directly querying the text data stored in HDFS. Next, we describe the subsequent data preparation steps taken in Impala and P-HBase,

IMPALA. We create the benchmark schema and load tables with the text data stored in HDFS. Next, we utilize the COMPUTE STATS command to collect statistics for each table.

P-HBASE. We create UserVisits and Rankings tables with visitDate and pageRank as the first attribute in the respective row-keys. As a result, similar to [70], UserVisits and Rankings tables are sorted on visitDate and pageRank columns respectively. We assign all columns in a table to a single column family. We utilize the *salting* feature provided by Phoenix to pre-split each table with two regions per region-server. *Salting* prevents region-server hotspotting and achieves uniform load distribution across region-servers in the cluster. We utilize a MR based bulk loader in Phoenix to load the text data stored in HDFS into HBase tables. Next, we run *major compaction* on each table to merge multiple HFiles of a region into a single HFile. Phoenix collects data statistics for each table during the major compaction process.

Table 5.2 presents the data preparation times in each evaluated system for 2, 4 and 8 *worker* nodes. The time to load data into HDFS from the landing node increases linearly with an increase in the cluster and the data size (recall that the data size increases with each *worker*). The time to load data into Impala tables is a small fraction of the total data preparation cost since Impala's text loading process simply moves the files to a Impala managed directory in HDFS. The statistics computation process in Impala and the major compaction process in P-HBase exhibit good *scale-up* characteristic as the execution times remain nearly constant for the different cluster sizes. However, the data loading times for the MR based bulk loader in P-HBase increase with an increase in the data size and the number of *worker* nodes in the cluster.

Table 5.3: Query RTs (in seconds) in evaluated systems using WDA benchmark for 2, 4 and 8 worker nodes in the cluster. RT in bold text denotes the fastest system for each query and cluster size combination. AM denotes the arithmetic mean. To compute the normalized AM: for each query, we normalize the query RTs in each system and for each cluster size by the query RT in Impala with 2 worker nodes.

WDA Query No.		**# of worker nodes – 2**				**# of worker nodes – 4**				**# of worker nodes – 8**			
		Impala	**Spark**	**Drill**	**P-HBase**	**Impala**	**Spark**	**Drill**	**P-HBase**	**Impala**	**Spark**	**Drill**	**P-HBase**
Q1	Execution	**0.28**	5.5	9	0.3	**0.28**	6	16.5	0.34	**0.28**	7	30	0.62
	Write Output	3.13	**0.5**	2	8.7	6.03	**0.9**	3.5	14.66	11.52	**2.1**	5	26.38
	Total	**3.41**	6	11	9	**6.31**	6.9	20	15	11.8	**9.1**	35	27
Q2	Total	**12.6**	68.6	33	240	**12.4**	68	31.8	228	**13.15**	69.1	68	231
Q3	Total	14.8	67.1	34	**14**	**15**	70.1	35.1	17	**15.12**	84.6	72	40
AM		10.27	47.23	26	87.67	11.24	48.33	28.97	86.67	13.36	54.27	58.33	99.33
Normalized AM		1	3.89	2.7	7.52	1.28	4.04	3.58	7.84	1.84	4.61	6.79	9.63

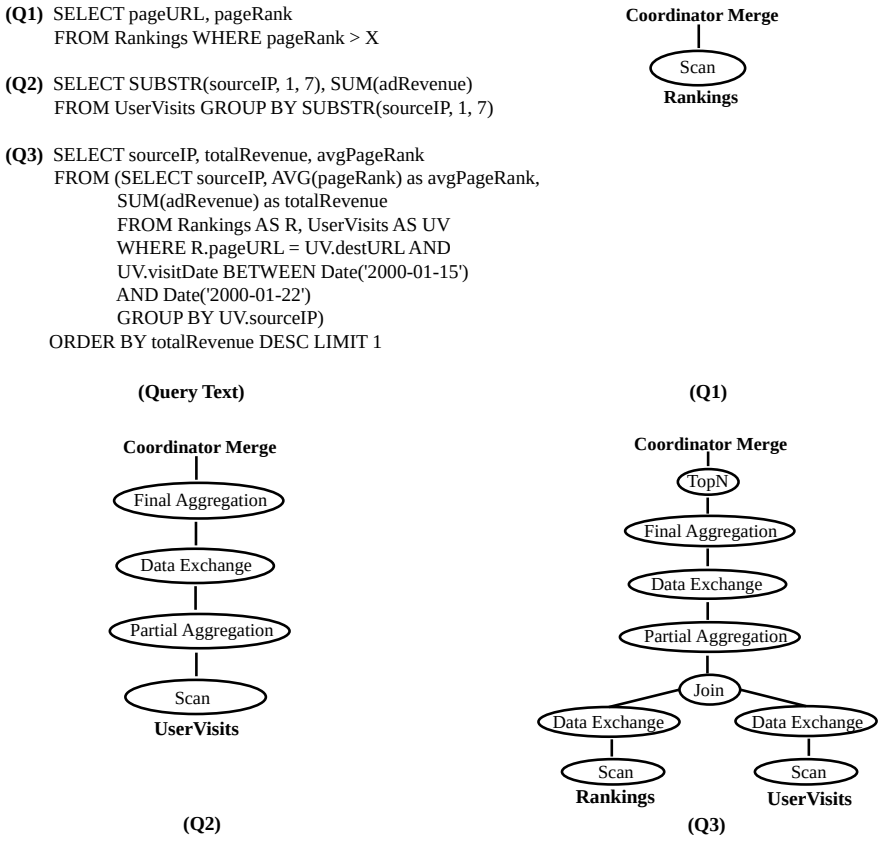


Figure 5.2: Query text and logical plans for the WDA benchmark queries.

5.3.6 Experiment Results

Table 5.3 presents the query RT for tasks in the WDA benchmark. Note, the standard error of the mean query RT is negligible in evaluated systems; hence, we exclude it from the presentation of the results. To evaluate the *scale-up* characteristic in each system, we perform experiments with 2, 4 and 8 *worker* nodes in the cluster. On an average, Impala is (5.2x - 7.5x), (2.7x - 3.7x) and (2.5x - 3.9x) faster as compared to P-HBase, Drill, and Spark SQL, respectively. Next, for each benchmark task, we analyze the execution profiles to understand its performance characteristics in each evaluated system. Figure 5.2 shows the text and the logical execution plans for the benchmark tasks.

5.3.6.1 Selection Task (Q1)

The selection task is a scan query with a lightweight filter on the Rankings table. It is designed to primarily measure the read and the write throughput in each evaluated system. Similar to [70], we set the filter parameter to 10, resulting in $\approx 33K$ records per *worker* node in the cluster.

DISCUSSION. Impala, Spark SQL and Drill perform a full scan of the Rankings table and apply the filter to generate the output RS. Impala is the fastest with sub-second scan operator time. P-HBase achieves significant scan efficiency by performing a range-scan on the Rankings table since it is sorted on the pageRank attribute. The scan operator time represents a major fraction (.95x) of the total query execution time in Spark SQL. Drill is the slowest and the scan operator is the primary contributor (.9x) to the total query execution time in Drill. Impala and Spark SQL exhibit the lowest (8%) and the highest (35%) mean CPU utilizations, respectively. The output RS materialization time shows that Spark SQL achieves the highest write throughput and P-HBase is the slowest.

SCALE-UP BEHAVIOR. The constant query execution time across different cluster sizes shows linear *scale-up* behavior in Impala. Although the relative query execution time in Drill, Spark SQL and P-HBase increases with the increase in the cluster size, Drill exhibits the maximum increase ($\approx 80\%$). Further examination of the query execution profile in Drill shows that, although the processing time of the scan operator remains nearly constant, the scan operator wait time increases as the cluster is scaled up, resulting in the observed increase in the query execution time. In each system, the output RS materialization time increases proportionately with the increase in the cluster size since nearly $33K$ records are generated for each *worker* node in the cluster.

5.3.6.2 Aggregation Task (Q2)

The aggregation task groups records by the seven-character prefix of the sourceIP attribute in the UserVisits table and produces ≈ 1000 groups regardless of the number of

workers in the cluster. It is designed to evaluate the performance of each system for parallel analytics on a single table. Note, output RS materialization time is negligible; hence, we only report the total query RT.

DISCUSSION. Each evaluated system scans the UserVisits table and performs partial-aggregation followed by the final-aggregation to generate the output RS. Impala is at least 5x faster than the other evaluated systems. Impala uses the streaming-aggregation (SA) operator and it is the primary contributor ($\approx$ .85x) to the query RT. Drill utilizes the hash-aggregation (HA) operator and although the query RT is high in Drill, the total aggregation operator processing time is less than 2s (as compared to $\approx$ 11s in Impala) across all cluster sizes. The scan operator is the primary contributor (.65x - .9x) to the query RT in Drill. P-HBase exhibits the highest query RT since it scans approximately 40% more data (UserVisits size is $\approx$ 1.4x in P-HBase, due to the HBase HFile format that appends key, column family, column and timestamp to each cell value) than other systems and its execution engine lacks the run time code generation feature. In Spark SQL, the scan and the partial-aggregation operators are mapped to a single DAG stage that contributes nearly 98% to the query RT. Impala achieves lowest mean CPU utilization (20%) as compared to Drill (35%), Spark SQL (60%) and P-HBase (55%).

SCALE-UP BEHAVIOR. Impala, Spark SQL and P-HBase exhibit near linear *scale-up* characteristic as the query RT remains almost constant with the increase in the cluster size. On the contrary, the query RT in Drill more than doubles as the cluster size is increased from 4 to 8 *worker* nodes. Further analysis of execution profile shows that, similar to the selection task, increases in the scan operator wait time is primarily responsible for this increase in the query RT. Note, although the query RT in Drill increases as the cluster is scaled up, the aggregation operator time remains nearly constant.

5.3.6.3 Join Task (Q3)

The join task consumes two input tables (Rankings and UserVisits) and combines records with values matching on the join attributes (pageURL and destURL). It is designed to examine the efficiency of each sub-system (CPU, disk, etc.) in the evaluated query execution engines.

DISCUSSION. Drill, Impala, and Spark SQL scan and hash-partition Rankings and UserVisits tables on the join keys. The matching records from the partitioned tables are then joined, aggregated and sorted to generate the output RS. Impala utilizes the hash-join (HJ) operator and although the join operator processing time is high in Impala ($\approx$ 7s), query RT is dominated by the scan operator time ($\approx$ 80% of query RT) for the UserVisits table. Similarly, the scan operator time for the UserVisits table is the primary contributor to the query RT in both Spark SQL (.8x – .9x) and Drill (at least .75x). Drill uses the HJ operator and despite the high query RT, Drill exhibits the lowest join operator processing time (less than 4.5s) among all evaluated systems and across all cluster sizes. P-HBase performs a range-scan of the UserVisits table to prepare and broadcast the hash table to each region server. Since the UserVisits table is sorted on the filter attribute visitDate, P-HBase is able to perform the range-scan and gain significant scan efficiency.

SCALE-UP BEHAVIOR. The query RT in P-HBase increases as the cluster is scaled up since the time to broadcast the hash table of one join input from the client to the region servers increases. Spark SQL utilizes the sort-merge-join (SMJ) operator and the join operator processing time shows increase as the cluster is scaled up. Similar to the aggregation task, the query RT in Drill more than doubles as the cluster size is increased from 4 to 8 *workers* due to an increase in the scan operator wait time. In addition, the join operator time in Drill shows marginal increase (2.7s – 4.2s) as the cluster is scaled up. Impala exhibits near linear *scale-up* behavior with almost constant query RTs across different cluster sizes.

5.4 TPC-H Benchmark Experiments

Next, we utilize the TPC-H benchmark to evaluate and compare the performance of Impala, Drill and Spark SQL systems. We use TPC-H since its workload comprises of a wide range of complex analytical tasks that thoroughly test query expressiveness, optimizer quality and query execution engine efficiency in the examined systems. We evaluate each system with the data stored in, both the text and the parquet storage formats. The parquet format enables analytical systems to achieve improved disk I/O efficiency by allowing reads to skip unnecessary columns, and improved storage efficiency through compression and encoding schemes. To evaluate the *size-up* characteristic of each examined system, we perform experiments for three scale-factors (SFs) : 125, 250, and 500. Note that SF denotes the database size (in GB) in the text format. We exclude P-HBase from the TPC-H experiments since more than 90% of the benchmark queries require evaluation of one or more joins to compute the output RS. However, the P-HBase execution engine architecture with client coordinated shuffle is not apt for join heavy workloads and results in orders of magnitude slower query performance as compared to the other evaluated systems. We use the same experiment setup for each evaluated system as described in Section 6.3.3.

Table 5.4: Data preparation times (seconds) in systems for the TPC-H benchmark

TPC-H Scale Factor	**Insert HDFS**	**Impala**						**Spark SQL and Drill**		
		Text			**Parquet**			**Text**	**Parquet**	
		Load Tables	**Compute Stats**	**Total**	**Load Tables**	**Compute Stats**	**Total**	**Total**	**Convert**	**Total**
125	957	31.4	118.4	1106.8	120.2	127.7	1236.3	957	134.8	1091.8
250	1900	32.8	214.6	2147.4	205.9	247.2	2385.9	1900	212.3	2112.3
500	3898	33.4	415	4346.4	354.1	441.9	4727.4	3898	328	4226

5.4.1 Hardware and Software Configuration

Our experiment testbed comprises of 20 *worker* VMs, 1 *client* VM and 1 *master* VM (see Section 6.3.1 for VM instance descriptions). We use the same software configuration for each evaluated system as described in Section 6.3.2.

5.4.2 Data Preparation

For the text format, we use the same data preparation steps in each evaluated system as described in Section 5.3.5. For the parquet format, we take different steps in the Impala and the Spark SQL systems. In Spark SQL, for each TPC-H table, we use a script to first read the text files stored in HDFS into a *rdd*, then convert the *rdd* into a *data frame* and finally save the *data frame* back into HDFS in the parquet format. Created parquet files are then queried in, both Drill and Spark SQL systems. In Impala, we first create the schema for parquet tables and then load parquet tables using the text tables. Next, we utilize the COMPUTE STATS command to collect statistics for each parquet table. We use *Snappy* compression with parquet format in each evaluated system. Table 5.4 shows the data preparation times for each evaluated system at TPC-H scale factors 125, 250 and 500. The DB size in the parquet format at scale factors 125, 250 and 500 is 39.9GB, 79.8GB and 168.1GB respectively. The data preparation in each system for both the text and the parquet formats increases proportionately with the increase in the data size, exhibiting good *size-up* property.

5.4.3 Experiment Results

Table 5.5 presents the RT of TPC-H queries in each evaluated system for the text and the parquet storage formats at scale factors 125, 250, and 500. Again, the standard error of the mean query RT is minimal in evaluated systems; hence, we exclude it from the presentation of the results. Table 5.5 also shows the arithmetic mean (AM) of the RT of all benchmark queries for each storage format, SF, and evaluated system combination.

Only 15 TPC-H queries could be evaluated in each system. AM–Q{2,11,13,16,19,21,22} represents the AM of the RT of all benchmark queries except Q2, Q11, Q13, Q16, Q19, Q21, Q22. The query optimizer in Impala failed to plan for Q11. Drill exhibits minimal query expressiveness with six failed queries in the two storage formats. Queries Q2, Q19,

Table 5.5: Query RTs (seconds) in evaluated systems using the TPC-H benchmark at 125, 250 and 500 scale factors. RT in bold text denotes the fastest system for each query, scale factor and file format combination. To compute the normalized AM–Q{2,11,13,16,19,21,22}: for each query, we normalize the query RTs in each system, at all scale factors and for each storage format by the query RT in Impala, for the parquet storage format, at scale factor 125.

TPC-H Query No.	TPC–H Scale Factor – 125						TPC–H Scale Factor – 250						TPC–H Scale Factor – 500					
	Impala		Spark SQL		Drill		Impala		Spark SQL		Drill		Impala		Spark SQL		Drill	
	Text	Parquet	Text	Parquet	Text	Parquet	Text	Parquet	Text	Parquet	Text	Parquet	Text	Parquet	Text	Parquet	Text	Parquet
Q1	37.68	37.79	**24.8**	**4.86**	33.33	18.06	71.09	73	**46.23**	**6.56**	63.81	26.9	142.43	138.6	**78.46**	**22.56**	132.33	50.7
Q2	**3.59**	**2.58**	43.4	16.03	Failed	Failed	**5.56**	**3.43**	72.93	23.86	Failed	Failed	**8.73**	**4.1**	152.5	33.2	Failed	Failed
Q3	**9.08**	7.34	28.7	**7.3**	52.16	16.5	**15.66**	12.66	54.5	**11.5**	92.46	23.7	**28.33**	24.2	112.93	**22.13**	189.46	40.46
Q4	**7.94**	**7.31**	44.2	25.56	54.76	18.36	**13.76**	**12.66**	79.8	46.46	94.63	29.1	**28.5**	**16.66**	171.4	94	197	44.9
Q5	**13.3**	**10.44**	46.1	25.16	50.23	13.8	**23.9**	26.63	97.13	35.1	95.83	**18.33**	**44.63**	52.2	151.06	62	198.73	**37.16**
Q6	**3**	**1.81**	21.63	2.26	29.93	5.26	**5.86**	**1.84**	36.76	2.7	57.66	6.73	**11.16**	**2.8**	74.03	3.9	125.73	12.86
Q7	**13.47**	**13.29**	75.81	22.4	52.1	17.96	**22.2**	**21.76**	144.33	61.56	100.4	32.83	**42.16**	**44.6**	244.43	65.06	199.5	66.73
Q8	**5.68**	**3.49**	47.33	18.86	42.23	5.53	**10.6**	**6.7**	67	41.5	87.2	11.46	**21.73**	**11.53**	128.4	78.73	208.56	22.93
Q9	**14.79**	**12.29**	54.26	30.36	46.7	12.4	**23.53**	**17.73**	95.7	66.73	112.3	28.1	**40.13**	**31.06**	181.13	125	231.33	61.7
Q10	**7.27**	**5.68**	32.83	8.56	40.73	12.86	**11.6**	**9.13**	62.26	10.76	79	19.06	**20.73**	**15.5**	110.13	32.56	168.86	40.46
Q11	Failed	Failed	33.23	21.33	**11.93**	**1.66**	Failed	Failed	51.46	28.76	**19.76**	**1.8**	Failed	Failed	77.96	37.63	**32.66**	**3.66**
Q12	**6.58**	4.97	26.76	**4.76**	37.43	18.1	**11.23**	7.96	48.1	**6.5**	76.7	28.3	**21.03**	12.93	93.56	**10.43**	157.6	58.3
Q13	**9.01**	10.75	16.43	**9.26**	Failed	9.36	**17.86**	20.46	22.96	15.23	Failed	**9.4**	34.93	36	**34.86**	25.73	Failed	**22.9**
Q14	**4.44**	3.88	23.26	**3.73**	30.2	4.5	**7.5**	**4.4**	39.9	4.96	61.53	8.83	**12.93**	8.3	72.9	**7.23**	134.76	18.33
Q15	**11.26**	8.4	49.5	**4.96**	30.6	6	**18.4**	9.92	77.96	**7.36**	61.99	13.26	**31.88**	16.21	147.06	**12.03**	129.06	22.26
Q16	**6.37**	**6.32**	68.93	81	Failed	8.46	**6.16**	**8.36**	136.76	152	Failed	17.06	**12.75**	**12.96**	286.93	371.96	Failed	25.3
Q17	**29.65**	31.19	53.8	17.06	95.26	**14.96**	**61.23**	64.73	99.93	41.93	182.26	**24.13**	**125.76**	96.74	202.06	87.06	361.63	**43.76**
Q18	**17.8**	17.82	63.6	**15.13**	98.76	25.93	**36.24**	30.2	110.83	**27.23**	226.36	48.16	**72.93**	**47.03**	201.9	51.5	478.06	124.83
Q19	49.69	52.22	**22.96**	**4.46**	Failed	Failed	105.73	114.53	**40.63**	**5.36**	Failed	Failed	211.13	209.13	**74.46**	**8.8**	Failed	Failed
Q20	**8.54**	**4.53**	38.26	21.43	45.13	9.86	**15.24**	**7.46**	57.26	19.1	91.03	18.4	**29.1**	**13.16**	110.86	26.73	171.96	27.63
Q21	**24.19**	**23.15**	131.7	83.63	Failed	Failed	**46.8**	**44.83**	265.66	172.8	Failed	Failed	**95.83**	**89**	583.5	364.43	Failed	Failed
Q22	**3.36**	**2.86**	22.86	13.3	Failed	Failed	**5.16**	**5.13**	30.7	17.26	Failed	Failed	**8.7**	**8.33**	48.23	35.66	Failed	Failed
AM	–	–	44.11	20.06	–	–	–	–	79.04	36.6	–	–	–	–	151.76	71.74	–	–
AM–Q{2,11,13, 16,19,21,22}	12.86	11.73	41.37	14.15	50.15	12.76	23.64	21.29	72.84	26.58	100.46	21.23	45.82	36.97	134.77	47.75	209.07	42.51
Normalized AM–Q {2,11,13,16,19,21,22}	1.26	1	5.83	1.87	6.75	1.68	2.25	1.65	9.95	3.15	13.4	2.79	4.27	2.86	18.86	5.65	28.41	5.8
Text (AM) over Parquet (AM)	1.26		3.11		4.01		1.36		3.15		4.8		1.49		3.33		4.89	

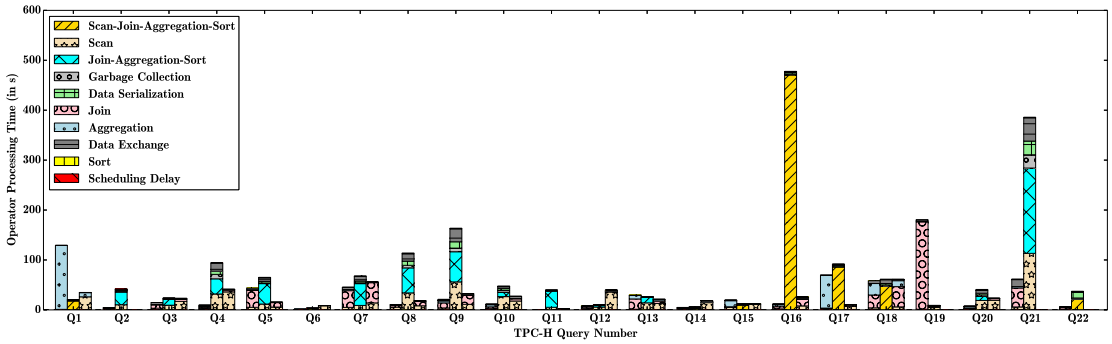


Figure 5.3: The breakdown of query RT into aggregated processing time for each operator type in evaluated systems. The TPC-H scale factor is 500 and the storage format is parquet. For each query, the left bar, the middle bar and the right bar represent Impala, Spark SQL and Drill systems, respectively.

Q21 and Q22 failed for both storage formats in Drill with a server side "Drill Remote Exception", whereas queries Q13 and Q16 failed with the same error for the text storage format only.

We use the normalized AM–Q{2,11,13,16,19,21,22} (see Table 5.5) to carry out an overall performance comparison of evaluated systems for the text and the parquet storage formats at scale factors 125, 250, and 500. In the text format, Impala is the fastest (4.41x – 6.65x) and Drill is the slowest (1.15x – 6.65x), across all evaluated scale factors. In the parquet format, although Drill is nearly 1.1x faster than Spark SQL at smaller scale factors (SF 125 and SF 250), Spark SQL marginally outperforms (1.02x) Drill for the largest evaluated scale factor (SF 500). Impala is the fastest (1.68x – 2.0x) system in the parquet format, across all evaluated scale factors. In contrast with the text format, parquet format results exhibit interesting query performance trade-offs with each system outperforming the other two systems for a subset of TPC-H queries.

In the subsequent sections, we analyze the query execution profiles to gain an insight into the optimizer characteristics and the execution engine efficiency in each evaluated system.

5.4.3.1 Execution Time Breakdown

In this section we present the breakdown of query RT into aggregated execution time for each operator type to understand execution characteristics in evaluated systems. We perform this analysis for the largest evaluated scale factor (SF 500) in the parquet format (see Section 5.4.3.3 for parquet vs. text comparison). Figure 5.3 depicts the execution time breakdown for the TPC-H benchmark queries in evaluated systems.

IMPALA. Impala primarily utilizes HJ and SA operators to perform join and aggregation operations, respectively. The query RT is dominated by scan, join and aggregation operator times in Impala. On an average, the join and aggregation operator times are 35% and 25% of the query RT, respectively. Use of a single CPU core to perform the join and the aggregation operations (identified in previous work [72] as well) combined with the choice of SA operator to perform the grouping aggregation, results in sub-optimal CPU and memory resource usage and is the primary performance bottleneck in Impala. Although, on an average, the scan operator time is 18% of the query RT, Impala exhibits the most efficient disk I/O sub-system among all evaluated systems. The average data-exchange operator time is 6% of the query RT, demonstrating efficient network I/O subsystem. Query 19 RT is relatively high in Impala since predicates are evaluated during the join operation instead of being pushed down to the scan operation.

DRILL. Drill mainly uses HA and SA operators to perform grouping and non-grouping aggregation operations, respectively. In addition, HJ represents the primary join operator in Drill. The scan operator contributes the maximum (on an average 42%) to the query RT in Drill. The total scan operator time in Drill is nearly 4.5x as compared to Impala for all benchmark queries that completed in both systems. Although, on an average, the join operator time in Drill is 21% of the query RT, the HJ operator choice combined with an efficient operator implementation results in lowest total join operator time for all benchmark queries among all evaluated systems. Drill exhibits high scheduling overhead with average time being 13% of the query RT; however, the data-exchange operator shows notable

efficiency with average time being 4% of the query RT.

SPARK SQL. Recall that multiple query plan operators may be mapped to a single DAG stage in Spark SQL (see Section 5.1). Hence, for queries that perform: 1) partial-aggregation and join, and/or 2) final-aggregation and sort operations in a single stage, we present the sum of join, aggregation, and sort operator times, denoted as JAS. Also, for queries that perform scan and partial-aggregation operations in a single stage, we present the sum of scan and JAS operation times.

Spark SQL largely utilizes SMJ and HA operators to perform join and aggregation operations, respectively. On an average, the scan and the JAS operations contribute 42% and 46% to the query RT, respectively (based on the 15 TPC-H queries that perform scan and JAS operations in separate stages). The joins are expensive in Spark SQL due to use of SMJ operator that performs a costly sort operation on both join inputs before combining the matching records. On an average, the GC time is 7% of the query RT and scan operation represents the principal source of GC overhead. Although the average data-exchange operator time is 7% of the query RT, the network data transfer performance in Spark SQL is at least 3x slower as compared to other evaluated systems based on the total benchmark data-exchange time.

5.4.3.2 Correlated Sub-query Execution in Drill

In this section we discuss correlated sub-query execution characteristic in Drill through an example TPC-H query (Q4). In the case of correlated sub-queries with one or more filters on the outer table, Drill optimizer generates a query execution plan that first performs a join of the outer and inner table to filter the inner table rows for which the join key does not match with the join key of the filtered outer table. The filtered inner table rows are then joined with the outer table rows to generate the output RS. We also compare the Drill query execution with the query execution in Impala and Spark SQL systems. Figure 5.4 depicts the query text, execution plans and the profiled operator times in evaluated systems for the

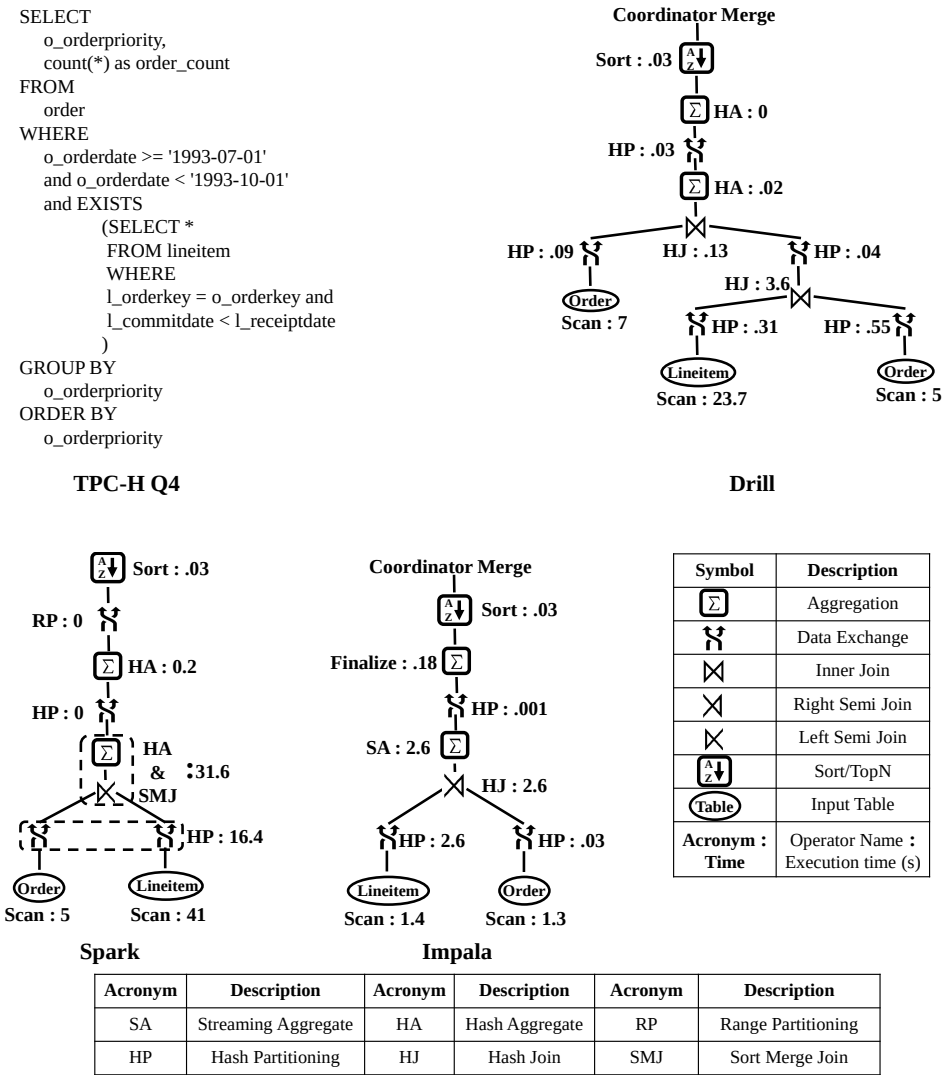


Figure 5.4: Query text and execution plans with profiled operator times in evaluated system for TPC-H query 4. The storage format is parquet and the SF is 500

TPC-H query 4. Note that the storage format is parquet and scale factor is 500.

DRILL. The Order table is scanned (o_orderkey, o_orderdate), filtered (o_orderdate >= '1993-07-01' and o_orderdate <'1993-10-01') and hash-partitioned on the o_orderkey. Similarly, Lineitem table is scanned (l_orderkey, l_commitdate, l_receiptdate), filtered (l_commitdate <l_receiptdate) and hash-partitioned on the l_orderkey. Next, tuples from the Order and the Lineitem partitions are inner joined using the HJ operator and the intermediate RS is hash-partitioned on the o_orderkey. This join operation reduces the number of Lineitem rows that

are shuffled across the cluster nodes. Next, Order table is scanned (o_orderkey, o_orderdate, o_orderpriority), filtered (o_orderdate >= '1993-07-01' and o_orderdate <'1993-10-01') and hash-partitioned on the o_orderkey for the second time. Then, tuples from the intermediate RS and Order partitions are inner joined using the HJ operator. Subsequently, the results are partially hash-aggregated and hash-partitioned on the grouping attribute (o_orderpriority) to enable final hash-aggregation. Finally, sorted results are merged in the coordinator node.

The first join between the Lineitem and the Order table reduces the data that are partitioned across the cluster nodes. However, as shown in Figure 5.4, scan operation is the primary performance bottleneck in Drill. In addition, using the same plan with text data worsens the performance since all columns in the Order table are scanned twice during query execution.

SPARK SQL. The Lineitem and the Order tables are scanned and hash-partitioned on the join keys (o_orderkey, l_orderkey). The tuples from the Order and Lineitem table partitions are then left-semi joined using the SMJ operator. The results are then partially hash-aggregated and hash-partitioned on the grouping attribute (o_orderpriority) to enable final hash-aggregation. The aggregated results are then range partitioned and sorted to generate the output RS. Although only three columns need to be scanned from both tables in the parquet format, due to a bug in the plan generation for queries with *exists* clause, all columns are scanned in both tables. As a result, scan operation is very costly for both tables. In addition, since relevant columns are projected after the join operation, the data-exchange operation and the sort operation in SMJ (required disk spill) are expensive as well.

IMPALA. Similar to Spark SQL, Impala scans and hash-partitions the Lineitem and the Order tables on the join keys (o_orderkey, l_orderkey). The tuples from the Lineitem and the Order table partitions are then right-semi joined using the HJ operator. The join results are then partially hash-aggregated and hash-partitioned on the grouping attribute (o_orderpriority) to enable final hash-aggregation. The aggregated results are then sorted

and merged in the coordinator to produce the output RS. A simple and effective query execution plan combined with an efficient disk I/O subsystem enables Impala to outperform other systems by at least a factor of 2.

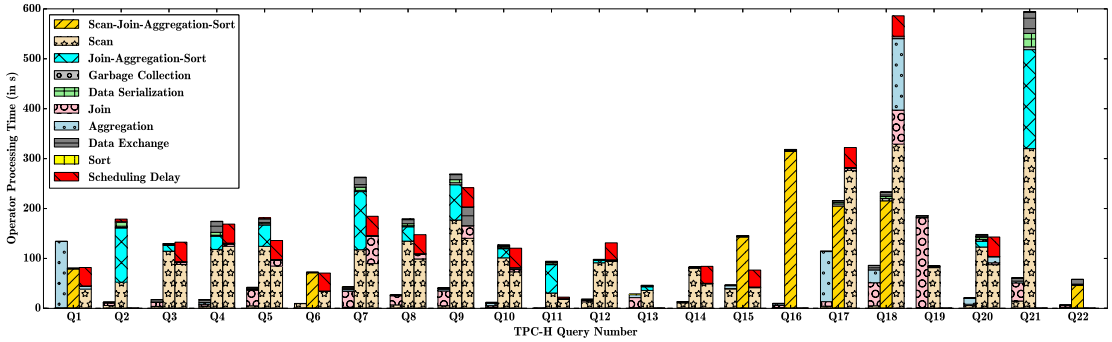


Figure 5.5: The breakdown of query RT into aggregated processing time for each operator type in evaluated systems. The TPC-H scale factor is 500 and the storage format is text. For each query, the left bar, the middle bar and the right bar represent Impala, Spark SQL and Drill systems, respectively.

5.4.3.3 Parquet versus Text Performance

In this section we evaluate the query performance differences between the text and parquet storage formats in each evaluated system. The last row in Table 5.5 shows the ratio of overall text to parquet performance in each system for TPC-H scale factors 125, 250 and 500. These numbers were computed using the normalized AM–Q{2,11,13,16,19,21,22} values. Drill and Impala exhibit the maximum (4.01x – 4.89x) and minimum (1.26x – 1.49x) performance speed-up, respectively from the text to the parquet format. Also, as we increase the data size, the speed-up factor between the two formats increases in each system. Next, we use results from the largest TPC-H scale factor (SF 500) to understand the reasons for the performance difference between the two storage formats in evaluated systems. For each system, figures 5.3 and 5.5 show the per operator total time spent by a query in the parquet and text storage formats, respectively.

IMPALA. Impala explicitly collects statistics on the data and generates the same query execution plan in both formats. On an average, the query scan operator time in the text

format is 3.2x the parquet format. The input and output data sizes for a join operator remain same in both storage formats; however, on an average, the HJ operator time in the text format is 3x as compared to the parquet format. Excluding query 17 for which the partial-aggregation operator time increases by 60% from the parquet to the text format, the total aggregation operator time for all benchmark queries shows nominal difference (less than 5%) between the two formats. The sort and the data-exchange operator times exhibit minimal variance between two storage formats. Hence, scan and HJ operators are the primary contributors to the increase in query RT in the text format as compared to the parquet format.

DRILL. Drill generates the same query execution plan for both storage formats. The join, aggregation, sort and data-exchange operators exhibit nominal difference in the processing time between the two storage formats. The scan operator is the principal performance bottleneck in the text format in Drill, since on an average, the scan operator time in the text format is 12x as compared to the parquet format.

SPARK SQL. Spark SQL generates query execution plans with the same join order in both formats. To query the text data, the Spark SQL optimizer harnesses SMJ operator to perform all the joins in the execution plan. However, for the parquet data, small tables (region and nation) are exchanged using the broadcast mechanism and the HJ operator is utilized to join tuples from the small table and the other join input. The joins performed using the same operator (SMJ) show nominal difference between the two storage formats. On an average, the scan operator time in the text format is 8.7x as compared to the parquet format. Note, remaining operations (data-exchange, GC, etc.) show insignificant difference in processing time between the two storage formats.

Table 5.6: Size-up property evaluation in each system. SF denotes scale factor.

Storage Format	Impala		Spark SQL		Drill	
	SF250 / SF125	SF500 / SF250	SF250 / SF125	SF500 / SF250	SF250 / SF125	SF500 / SF250
Text	1.78	1.89	1.76	1.89	1.99	2.1
Parquet	1.65	1.73	1.68	1.79	1.66	2.07

5.4.3.4 Size-up Characteristic Evaluation

In this section we assess the *size-up* behavior in the evaluated systems as we increase the TPC-H scale factor in multiples of 2 between 125 and 500. Table 5.6 presents the ratio of overall performance at consecutive scale factors in evaluated systems for both storage formats. These numbers were computed using the normalized AM–Q{2,11,13,16,19,21,22} values.

IMPALA. Impala exhibits sub-linear size-up behavior for both storage formats. On an average, the join, aggregation and data-exchange operator times double as the database size is doubled. However, the scan operator time exhibits sub-linear increase, resulting in the sub-linear size-up behavior in Impala.

DRILL. With the increase in the database size, the optimizer's join procedure selection in Drill favors hash-partitioned HJ as compared to the broadcast HJ. Hence, in the parquet format, Drill chooses more broadcast HJs for scale factor 125 (DB size – 39.9 GB) as compared to the scale factor 250 (DB size – 79.8 GB). Since broadcast HJs exhibit higher execution times in comparison with the hash-partitioned HJs in Drill, sub-linear size-up behavior is observed as the scale factor is doubled from 125 to 250 in the parquet format.

SPARK SQL. Spark SQL shows sub-linear size-up behavior for both storage formats. In the text format, although the JAS operation time reduces marginally as the database size is doubled, the reduction in scan operation time is primarily responsible for the sub-linear size-up behavior. In the parquet format, the reduction in both scan and JAS operation times is accountable for the sub-linear size-up behavior.

5.5 Discussion and Chapter Summary

In this section we summarize the strengths and weaknesses of evaluated systems and the lessons learned from this study.

QUERY OPTIMIZER. The query optimizers in Impala and Spark SQL generate simple

and efficient execution plans by evaluating and selecting from a variety of join strategies including semi-join, anti-join, etc. However, we note that the cardinality estimates can be significantly off in Impala, resulting in expensive join order selection in some cases. The cost-based query optimization is still in its nascent stages in Phoenix; hence, users need to: 1) define the join evaluation order, and 2) choose the join algorithm to be used. The query optimizer in Drill can generate complex and inefficient execution plans, especially for the correlated sub-queries.

QUERY EXECUTION ENGINE. The Impala execution engine has the most efficient and stable disk I/O sub-system among all evaluated systems, as demonstrated by the lowest scan operator times for both storage formats. Although the Drill execution engine (with the columnar data model for in-memory processing) is optimized for the on-disk columnar data storage format (parquet), the scan operator is the principal contributor to the query RT in the parquet format. In addition, the scan operator becomes a performance bottleneck in Drill for the data stored in the text format with an unstable behavior characterized by the high scan operator wait times. In comparison with other evaluated systems, Phoenix has a notably larger data footprint due to the HBase HFile storage format, resulting in expensive full table scans.

The join and aggregation operator implementations in the Drill query engine harness all available CPU cores and achieve the shortest processing times among all evaluated systems. In contrast, the use of a single CPU core to perform the join and aggregation operations in Impala, results in sub-optimal resource utilization; however, ongoing efforts to enable multi-core execution for all operators [99] should lead to performance improvement. Joins are costly in Spark SQL due to the choice of SMJ as the primary join operator, which requires an expensive sort operation prior to the join operation.

The data-exchange operator contributes nominally to the query RT in Impala, Spark SQL, and Drill. However, the client coordinated data-exchange operation is the primary performance bottleneck in Phoenix, making it ill-suited for join-heavy workloads that shuf-

fle large amounts of data.

The results from this study can be utilized in two ways: (i) to assist practitioners choose a SQL-on-Hadoop system based on their workloads and SLA requirements, and (ii) to provide the data architects more insight into the performance impacts of evaluated SQL-on-Hadoop systems.

Chapter 6

Fusion: Implementation and evaluation of block range indexes in a SQL-on-Object-Storage system

Cloud computing is emerging as a novel paradigm for large scale distributed computing, driven by the economies of scale. Cloud platforms like Amazon EC2 [28], Microsoft Azure [79], Google Cloud [100] etc., enable users to elastically acquire and release resources from a virtually infinite resource pool. Concomitant with the emergence of cloud computing is the data deluge that the enterprises are experiencing in storing the data generated from a variety of sources including social media, transactional systems, sensors, etc. Now, the combination of the need to glean actionable insights from the big data and to tap the potential of cloud hosting platforms, has led to the development of both new storage systems (e.g. HDFS [21], MapR-FS [101], Amazon S3 [25], Microsoft Azure Blob Storage [26], etc.) and query engines (e.g. Impala [23], Drill [24], Spark SQL [20]).

Instance storage, network attached Block storage, and Object storage represent different storage substrate choices for modern query engines (e.g. Impala, Spark SQL, etc.) on the cloud platforms. Each storage substrate choice represents a trade off between cost, persistence and performance. Hadoop storage system is designed to leverage the shared nothing architecture where each node in the cluster acts both as a compute node and a storage node. A Hadoop cluster in the cloud environment can utilize either Instance storage or network attached Block storage (e.g. Amazon EBS) as the storage substrate. A Hadoop cluster achieves optimal performance with the Instance storage by leveraging short circuit local reads. Although Instance storage is ideal for ad hoc data analysis, complete data loss after a cluster shutdown makes it less suitable for analyzing large and growing data, due to the high data population times. A Block storage service like Amazon EBS is non-ephemeral and enables users to elastically attach/detach network drives to the VM instances

in the cluster; however, EBS is expensive. In contrast, object storage systems like S3, Azure blob storage, etc., are cheaper than the network attached Block storage systems and provide better performance per dollar [35]. In addition, the use of a object storage system like S3 as the storage backend de-couples compute from storage, enabling independent scaling of either clusters. However, despite S3's better performance per dollar as compared to EBS, S3 may not be suitable for interactive analytics due to the data transfer latency (between EC2 and S3) and low read throughput. Hence, new mechanisms are required to enable interactive analytics with a object storage system as the storage backend of query engines.

Traditionally, the read performance of a query engine can be improved by increasing data read throughput and skipping irrelevant data. The data read throughput can be increased by employing caching, parallelizing reads and storing the data in a compressed format. Conventionally, columnar storage and indexing structures are used to skip reading the irrelevant data.

In this work, we focus on exploring indexing structures that can effectively skip irrelevant data stored in object storage systems like S3. Although B-tree, B+-tree, and R-tree indexes have been shown to be very effective in traditional data warehousing systems, high maintenance overhead, large storage footprint, and poor random I/O performance of object storage systems like S3, makes them less attractive for the query engines that use object storage systems as the storage backend. On the contrary, Block Range Indexes (BRIN) can be a better fit for skipping irrelevant data stored in object stores, since they exhibit low maintenance overhead, are lightweight (orders of magnitude smaller in size than traditional indexes [27]) and can be very effective for naturally ordered data (e.g. sensor data).

In this work our goals are two fold: 1) Empirically evaluate the performance impact of block range indexes in a object storage system using a standard benchmark. 2) Empirically evaluate the performance trade offs of different block range index implementations.

Table 6.1: Example Block range index for *id* column of Customers table.

min	max	block number
1	20	1
24	48	2
36	100	3

6.1 Block Range Index

In this section, we first review block range indexes. Then, we explore different implementation and storage choices for block range indexes. A table is stored as a set of blocks on a storage medium. For a table T, a block range index on a column *c* aggregates and materializes a set of values for each block in T. Table 6.1 shows an example block range index (*min*, *max*, *block number*), storing minimum and maximum aggregates for the *id* column of the Customers table, which comprises of 3 blocks on the physical storage. Note that the intervals in the index can overlap (e.g. intervals (24 – 48) and (36 – 100) in Table 6.1 overlap). The object storage systems like S3 and Azure Blob Store do not expose the size or number of blocks that a stored object is comprised of. Hence, object store users need to proactively partition a table into chunks and compute aggregates for these chunks to leverage the block range indexes.

A block range index on a column *c* can be utilized by the query engine to skip blocks of a table during query processing in two cases:

- EQUALITY filter on *c* in workload query (*c* = a) – In this case BRIN returns all chunk ids for which value (a) is between *min* and *max* values. As an example, for a workload query – SELECT * FROM Customers WHERE id = 5, the equality filter is *id* = 5; hence, BRIN returns $\{1\}$.

- RANGE filter on *c* in workload query (*c* in range(a,b)) – In this case BRIN returns all chunks ids for which range (a,b) overlaps with (*min*,*max*) interval. As an example, for a workload query – SELECT * FROM Customers WHERE id $\geq$ 25 and id $\leq$ 70,

the range filter is *id* in range(25,70); hence, BRIN returns {2,3}.

Hence, a *equality* filter in the workload query requires a response for a *point(a)* query from the block range index. A *range* filter in the workload query requires a response for a *range(a,b)* query from the block range index. During query processing, the query engine reads only those table chunks whose ids are returned by the block range index. Note that block range indexes can work with the data stored in both the text and columnar (e.g. Parquet) storage formats.

6.1.1 BRIN Implementation and Storage Choices

In this section we look at different choices in implementing and storing a block range index. A block range index can be implemented using a key value store or a interval tree data structure. A key value store based distributed BRIN implementation is scalable where as an interval tree based BRIN implementation can provide low query latency.

6.1.1.1 Key Value Store Based Implementation

We store a block range index as a table in an ordered key value store, such that each index interval (*min*, *max*) is stored as a row and each individual aggregate value in a row (e.g. *min*) is stored in a separate column. In addition, each row also stores the *chunk id* associated with the interval in an individual column. An ordered key value store like HBase is designed to perform fast key-based Get's and Range scans. Hence, we consider how to harness a key value store to efficiently perform *point(a)* and *range(a,b)* queries, that represent the standard workload for a block range index.

Point(a) Query – For a point query, we return all table rows whose interval (*min*, *max*) satisfies the predicate,

$$min \leq a \quad \cap \quad max \geq a$$

To efficiently execute this query in a key value store, we choose *min* as the first part of the table key. As a result, query becomes a range scan (*min* in range (* , a)) and the resulting rows are filtered on the fly (using filter *max* $\geq$ a).

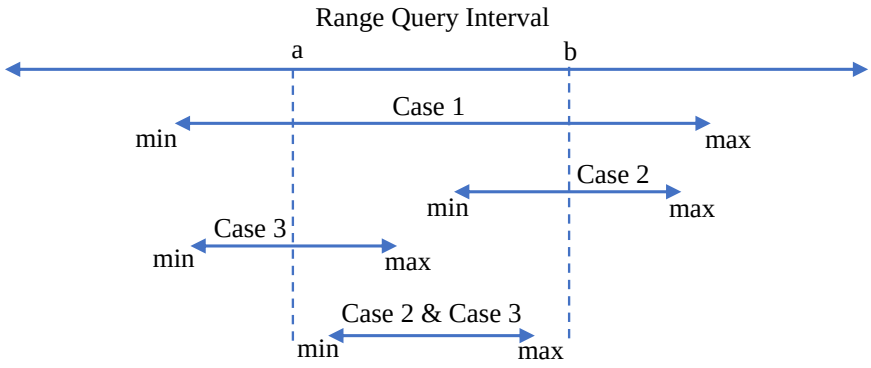


Figure 6.1: Range query sub cases in a block range index.

Range(a,b) Query – For range queries, we store two copies of an index, each in a separate table. For the first table we choose *min* as first part of the table key. Similarly, for the second table we choose *max* as first part of the table key. For a range query, we return all table rows whose interval (*min*, *max*) overlaps with the given interval (a, b). More specifically, we break it down into three sub cases (Figure 6.1) and return the union of rows returned by each sub case.

Case 1.

$$min \leq a \quad \cap \quad max \geq b$$

To efficiently execute this query in a key value store, we use the index that has *min* as the first part of the table key. As a result, query becomes a range scan (*min* in range (* , a)) and the resulting rows are filtered on the fly (using filter *max* $\geq$ b).

Case 2.

$$min \geq a \quad \cap \quad min \leq b$$

To efficiently execute this query in a key value store, we use the index that has *min* as the first part of the table key. As a result, query becomes a range scan (*min* in range (a , b)).

Case 3.

$$max \geq a \quad \cap \quad max \leq b$$

To efficiently execute this query in a key value store, we use the index that has *max* as the first part of the table key. As a result, query becomes a range scan (*max* in range (a , b)).

6.1.1.2 Interval Tree Based Implementation

We use the augmented self-balancing binary search tree to implement a interval tree [102] that resides on a single compute node. Note, exploring mechanisms to implement a distributed interval tree using a scalable storage engine like key value store, object store, etc., is part of our future work. Since the storage associated with a compute cluster is ephemeral, the interval tree needs to be serialized to a persistent storage medium to prevent re-computation. The different storage choices include RDBMS, key value store, object store, etc. Although RDBMS represents a feasible choice to persist the tree, a large enterprise can have several different applications using the system; hence, the tree storage choice needs to be scalable.

To persist the tree, we choose to serialize it to two different storage substrates: 1) as a blob in a object store, and 2) as a key-value pair in a key value store. The different implementation choices for a interval tree that resides on a single node in the compute cluster are 1) disk based, and 2) entirely memory resident. For the sake of simplicity, we implement the interval tree to reside in memory for the entire existence of a compute cluster.

6.2 System Overview

Figure 6.2 shows the overview of the Fusion SQL-on-Object-Storage system. Traditionally, users interact with the web/mobile applications and perform transactions. The

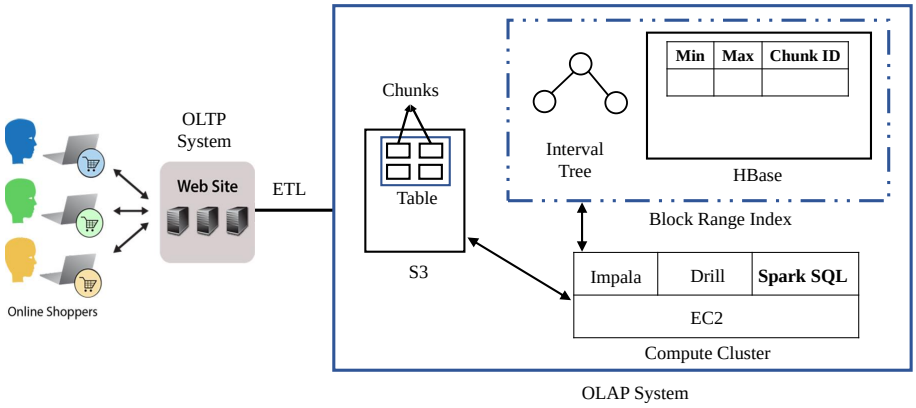


Figure 6.2: Fusion system overview.

transaction data from the OLTP system/s is then loaded into an OLAP system using the ETL process. We harness S3 as the storage engine for the OLAP system. The tables are horizontally partitioned into fixed sized chunks and the chunks for a table are stored in an individual S3 directory. We leverage Amazon EC2 cloud platform to create the compute cluster and deploy the Spark SQL query engine in it. Note, other distributed query engines like Impala, Drill, HAWQ, etc., could also be used. We utilize the Spark SQL query engine to compute ranges/intervals for each chosen block range index. We currently use a simple workload driven approach to choose block range indexes for creation. Specifically, we create an index for each column that is part of a point and/or range filter in the workload.

As described in section 6.1.1, we leverage interval trees and key-value stores as two different mechanisms to implement a block range index. We harness HBase NoSQL data store to implement the block range index. We use S3 and HBase as two different storage substrates to persist the interval tree. When a new compute cluster is created, we read and deserialize the index tree stored on a storage substrate (S3/HBase) and construct the interval tree that resides in the memory on a single node in the compute cluster. Before destroying a compute cluster, we serialize the tree back to the persistent storage (S3/HBase).

Query Processing. Figure 6.3 shows the different steps involved in the processing of a workload query in the Fusion system. User issues a workload query for execution to the Fusion system (step 1). To identify the relevant table chunks to query, Fusion system

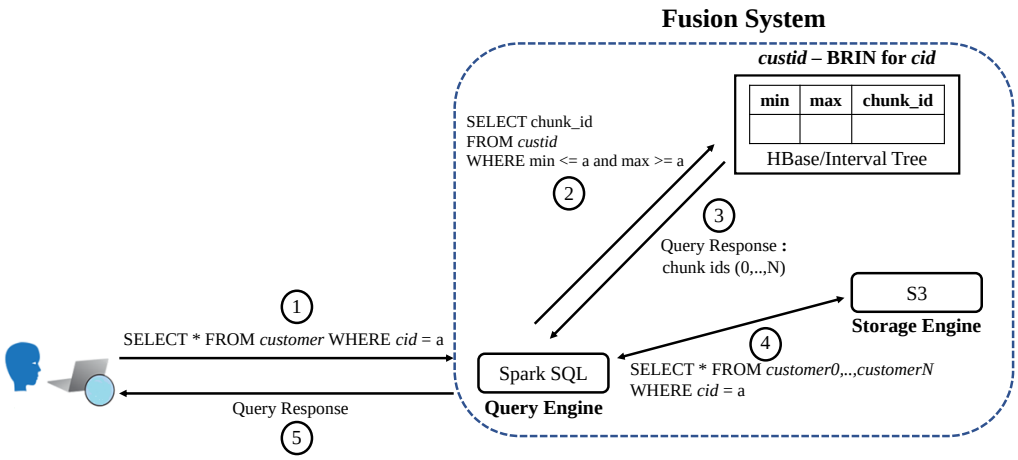


Figure 6.3: Query processing in Fusion system.

utilizes the filter value from the workload query to prepare and issue a query to the relevant BRIN index (step 2). Next, BRIN index response provides the ids of table chunks relevant to the workload query (step 3). Then, Fusion system re-writes the workload query using the chunk ids provided by the BRIN index (step 4). Finally, the Spark SQL query engine is used to execute the workload query and send the response back to the user (step 5). Note, in the absence of a usable filter or a relevant BRIN index, the Fusion system executes the workload query by reading all table chunks.

6.3 Experimental Evaluation

In this section we first describe our experiment environment. Next, we use the TPC-W benchmark to evaluate the performance of Fusion system and compare it with a Baseline system that does not use block range indexes.

6.3.1 Hardware Configuration

We harness Amazon EC2 cloud environment to setup the compute cluster (Spark) and the key value store cluster (HBase). Our testbeds comprise of "*worker*" VMs of m4.2xlarge instance type, a "*client*" VM of m4.2xlarge instance type and a "*master*" VM of d2.xlarge

instance type. A m4.2xlarge VM instance is configured with 8 vCPUs, 32GB RAM, and 120GB EBS storage and a d2.xlarge VM instance is configured with 4 vCPUs, 30.5GB RAM, 3x 2000GB HDD instance storage and 120GB EBS storage. We use three *worker* VMs in the compute cluster and two *worker* VMs in the key value store cluster.

6.3.2 Software Configuration

We deploy Spark, Yarn, Hive, and Zookeeper frameworks on the compute cluster. Similarly, we deploy HDFS, HBase, Phoenix, and Zookeeper frameworks on the key value store cluster. We host the control processes (e.g. HMaster) from each framework on the *master* VM. We also use *master* VM as the landing node for the data generation in the compute cluster. Worker processes from each framework (e.g. DataNode) are hosted on each *worker* VM in the cluster. We reserve the *client* VM to drive the workload. We deploy the Phoenix JDBC driver in the *client* VM and put relevant jars on the classpath of each Region Server. We use the Cloudera Distribution v5.8.0, Spark v2.0, HBase v1.2.0 and Phoenix v4.8.0.

Fine tuning of soft parameters is quintessential to getting the best performance out of a software system. To identify the values for performance knobs that ensure a fair comparison between evaluated systems, we rely on: 1) Settings used in prior studies (e.g. [74, 72]). 2) Best practice guidelines from industry experts and system developers (e.g. [98]). 3) Experimentation with different values that enable us to run all benchmark queries and achieve the best performance.

We enable Yarn as the resource manager for Spark and set *spark.executor.cores* to 8, *spark.executor.memory* to 8GB and *spark.memory.offheap* to 16GB.

6.3.3 Experiment Setup

We evaluate systems one at a time. During the evaluation of a system, we stop the processes for other systems to prevent any interference. We execute benchmark queries using a closed workload model, where a new query is issued after receiving the response

for the previous query. We run each query three times and report the average of the query RT for the last two runs with a warm OS cache. We use Phoenix as a SQL skin to query the block range indexes implemented using HBase. We use a JDBC driver to issue queries in Phoenix and shell scripts to issue queries in Spark SQL. We use *collectl* linux utility to record resource utilization on each *worker* node with a 3 second interval. We export the query execution profile in Spark SQL.

6.3.4 TPC-W Benchmark

We use the TPC-W [86] benchmark to empirically evaluate the implemented Fusion system. To perform experiments, we modulate the database size by varying two parameters in the population script: NUM_EBS and NUM_ITEMS. We exclude write statements from the workload that do not specify each key attribute in the where clause. This is done to prevent modifying more than one S3 chunk in response to a update statement in the transaction log of the OLTP system.

6.3.5 Experiment Results

In this section, we first evaluate the Fusion system and compare it with a Baseline system using the TPC-W benchmark. Next, we examine the impact of chunk size on the performance of the Fusion system. Then, we evaluate the performance impact of increasing the database size in the Fusion system. Finally, we examine the performance of TPC-W write statements in the implemented Fusion system.

6.3.5.1 Fusion versus Baseline

We evaluate the Fusion system and compare it with a Baseline system. Fusion system creates nine block range indexes based off of the analysis of the TPC-W workload. On the contrary, Baseline system does not utilize block range indexes. We populate TPC-W database with 10 GB of data and 16 MB table partitions in the Fusion system. Now, prior to

the index creation, we need to compute the intervals/ranges for each index using the table chunks.

Table 6.2: Total time to compute index intervals for 10GB TPC-W database with 16MB table partitions.

Number of Indexes	Total Index Intervals Computation Time (in s)
9	780

Index Intervals Computation. We utilize Spark SQL to compute intervals for each index. We compute intervals for indexes sequentially, one index at a time. In addition, for a index, we compute intervals sequentially, one chunk at a time. Table 6.2, shows the time to compute intervals for all nine indexes. We acknowledge that parallelism could be harnessed to reduce the time to compute index intervals.

Table 6.3: Total time to create indexes in the Fusion system.

Index Creation Time (in s)		
HBase	Interval Tree	
	S3 Storage	HBase Storage
283.4	6.25	1.48

Index Creation. Table 6.3 shows the time to create HBase and interval tree based block range indexes. We use a MR based data uploader (provided by Phoenix) to populate HBase indexes. The high task start up costs of MR jobs leads to significantly higher index creation time in HBase as compared to the interval trees. Interval tree based index creation time includes the time to serialize/deserialize index to/from the persistent storage (HBase/S3) medium. Although S3 storage is nearly 4x slower than the HBase storage for the interval tree based indexes, the use of HBase requires a EBS based cluster setup and maintenance, which can be expensive. Note, in contrast with HBase indexes that are query-ready across different compute cluster creations/shutdowns, interval tree based indexes need to be re-populated after each cluster creation/start.

Query Performance. Table 6.4 shows the performance comparison of Fusion and Baseline systems for all TPC-W queries that could use block range indexes. The total query

Table 6.4: Fusion versus Baseline: Performance of TPC-W queries that could use block range indexes.

Number of Queries	Fusion			Baseline
	Total Time to Query Index (in s)		Query Execution Time (in s)	Query Execution Time (in s)
	HBase	Interval Tree		
16	0.54	.01s	211.4	275

Table 6.5: Fusion versus Baseline: Overall performance of TPC-W benchmark queries.

Number of Queries	Total Query Execution Time (in s)	
	Fusion	Baseline
20	230.8	295.5

execution time in Fusion system is nearly 24% lower than the Baseline system. The time to query both HBase and interval tree based block range indexes is negligible as compared to the query execution time.

Queries that could not use any indexes in the Fusion system (with partitioned data) exhibit performance similar to the Baseline system (with un-partitioned data). Table 6.5 shows the overall performance of TPC-W queries in Fusion and Baseline systems.

6.3.5.2 Impact of Data Partition Size in Fusion System

To evaluate the performance impact of table partition size in the Fusion system, we experiment with two different table chunk sizes: 4MB and 16MB (see Section 6.3.5.1). Similar to Section 6.3.5.1, the TPC-W database is populated with 10GB of data.

Table 6.6: Index intervals computation in Fusion system for 10GB TPC-W database with 4MB and 16MB partitions.

Number of Indexes	Total Index Intervals Computation Time (in s)	
	16 MB	4 MB
9	780	4122

Index Intervals Computation. Table 6.6, shows the time to compute index intervals for both 4MB and 16MB chunk sizes. Reducing the chunk size from 16MB to 4MB increases the number of intervals that need to be processed; however it also reduces the data to process per interval. The total index intervals computation time for 4MB chunk size is

nearly 5x as compared to the 16MB chunk size. This behavior is attributed to higher job overhead and inefficient execution in Spark SQL for smaller (4MB) chunk size.

Note, **index creation** times exhibit a nominal increase for 4MB chunks as compared to the 16MB chunks; hence, we exclude these results from presentation.

Table 6.7: Total query execution time in Fusion system for 10GB TPC-W database with 4MB and 16MB partitions.

Number of Queries	Fusion - Total Query Execution Time (in s)	
	16 MB	4 MB
20	230.8	260.5

Query Performance. The basic idea behind smaller data chunk size is to enable more aggressive data skipping, leading to a reduction in the overall query response time. However, as shown in Table 6.7, for the TPC-W benchmark, the total query execution time in the Fusion system exhibits an increase for the 4MB chunk size as compared to the 16MB chunk size. Further investigation shows that, although several queries that use a block range index on a non-key column, read significantly less table data for the 4MB chunks than the 16MB chunks, the query response times do not exhibit a significant decrease due to the high processing overhead for smaller data chunks (4MB) in Spark SQL.

6.3.5.3 Fusion versus Baseline: Impact of Database Size

We evaluate the impact of database size on the query performance in both Fusion and Baseline systems. We experiment with two different database sizes: 10GB (see Section 6.3.5.1) and 50GB. Similar to Section 6.3.5.1, the data partition size is 16MB in the Fusion system.

Table 6.8: Fusion versus Baseline: Overall performance of TPC-W benchmark queries for 10GB and 50GB TPC-W databases.

Number of Queries	Total Query Execution Time (in s)			
	DB Size - 10GB		DB Size - 50GB	
	Fusion	Baseline	Fusion	Baseline
20	230.8	295.5	907.3	1170.4

Query Performance. Table 6.8 shows the the overall performance of TPC-W queries in Fusion and Baseline systems for 10GB and 50GB database sizes. Although the benchmark query execution time reduces in both systems with the increase in the database size, the total query execution time in Fusion system is nearly 22% lower than the Baseline system across both database sizes.

6.3.5.4 Write Statement Performance

Fusion System. For insert statements, we compute the time to insert a chunk in a S3 table and not a single row, since the ETL process can bundle multiple insert statements for a table from the OLTP transaction log into a chunk and then insert this chunk into the S3 table. This prevents creation of single row chunks in a S3 table.

Table 6.9: Write statement performance in Fusion system for 4MB and 16MB chunks.

Statement Execution Time (in s)			
Chunk Size - 4 MB		Chunk Size - 16 MB	
Insert	Update/Delete	Insert	Update/Delete
.34	1.75	1.54	3.22

For an update/delete statement, we first identify the chunk that needs to be updated. This is accomplished by first querying the relevant block range index to identify the candidate chunk(s) and then using the Spark SQL computing engine to narrow the candidate set down to the chunk that contains the record to be updated. Next, we read the identified chunk from S3 into memory, update the record and write the updated chunk back into S3. Although the time to read, update and write a chunk back into S3 is proportional to the chunk size, identifying the chunk to be updated may require querying all table chunks in the worst case. Therefore, the execution time for an update/delete statement can be arbitrary. Since the TPC-W data is sorted on the primary key, the query to the block range index for each update/delete statement in the benchmark always returns a single chunk. As a result, the execution time for each benchmark update/delete statement is nearly the same. Hence, we present the execution time for only one benchmark update/delete statement. Table 6.9

shows the write statement performance in the Fusion system.

Baseline System. To execute a insert/update/delete statement on a table, a new table with changes needs to be written in full, since the S3 objects are immutable. Hence, the time to modify a table in the Baseline system is proportional to the size of the table. However, it is impractical in practice to read and re-write big tables; hence, we exclude write performance evaluation in Baseline system from our experiments.

6.4 Chapter Summary

In this chapter, we present the implementation and evaluation of Fusion system that harnesses block range indexes as a mechanism to improve the query performance of SQL-on-Object-Storage systems. We empirically evaluate the creation and querying overhead associated with two different block range index implementations. Experiment results show that the use of S3 as a persistent storage medium for small interval tree based indexes, can achieve both acceptable performance and cost efficiency. Experiment with different table chunk sizes demonstrates that although smaller chunk size may allow for more aggressive data skipping, the query performance may actually regress due to the high processing overhead associated with smaller chunks in query engines like Spark SQL. The Fusion system with immutable table chunks is not suitable for update heavy workloads as demonstrated by the experimental evaluation of benchmark write statements.

Chapter 7

Future Work

In this dissertation, we have presented various mechanisms to enable scalable data management for each enterprise workload class. However, through the development of these mechanisms, we have identified several avenues to explore and build upon the current work. Next, we discuss some of these ideas as part of our future work. In addition, we discuss additional experiments than can be performed to improve the comprehensiveness of this dissertation.

7.0.1 Mechanisms

In chapter 3, we present a black box approach based on the database re-population, to reduce the observed performance variations and develop robust profiles. However, we do not take a position on how often the database should be re-populated. Hence, an alerting mechanism could be developed that monitors the performance and generates an alert prior to a significant performance regression. In addition, re-populating the complete database may not be practical for large databases. Therefore, fine-grained re-population strategies (e.g. rebuilding specific indexes) could be explored. The proposed black box solution provides symptomatic relief; however, a white box approach that identifies the actual source of variations, could be more useful in developing the mechanisms that prevent variations from manifestation.

In chapter 4, the proposed Synergy system utilizes a heuristic based approach to generate the candidate views for materialization. However, to generate more robust candidates, the generation mechanism could be adapted to use a cost based optimizer instead of a heuristic. The views selection mechanism in the Synergy system, selects views for each

workload query independently. However, this leads to the loss of opportunity to share views across different queries and save the storage space. Hence, we plan to supplement the Synergy system with a new views selection mechanism, that can leverage the opportunity to share views across different workload queries. In addition, we plan to merge our code for the proposed system with the Phoenix code base and make the Synergy system available to all Phoenix users.

In chapter 5, we profile the performance of four SQL-on-Hadoop systems along several dimensions including, file formats, scale-up, size-up, database size etc. The profiled results could be utilized to build performance models. The performance models can in turn be leveraged to answer WHAT-IF questions related to the capacity planning process for each evaluated system in the cloud environment.

In chapter 6, Fusion system utilizes in-memory interval trees to implement the block range indexes. However, in-memory interval trees are centralized and could be too big to hold in memory for large tables. Hence, we are currently working on devising a mechanism to create distributed interval trees using key value stores. In addition, the block range indexes selection mechanism in the Fusion system is oblivious to the distribution of data in the table columns. Hence, we plan to supplement the Fusion system by computing the data distribution information on table columns and using this information to guide whether implementing a block range index could be useful in skipping data. Exploring other indexing structures that can be used with the proposed Fusion system (e.g. Bitmap indexes can be very useful with low-cardinality columns) is part of our future work.

7.0.2 Experiments

- Chapter 3 – We host MySQL database server in a cloud VM instance and evaluate its performance. It would be interesting to evaluate and confirm that the observed behavior is not specific to MySQL server and can be generalized to other RDBMSs (e.g. PostgreSQL).

- Chapter 4 – A throughput evaluation using different request rates (using a workload driver that generates concurrent requests) would emphasize the higher request handled capacity in the proposed Synergy system as compared to the other evaluated systems. In addition, Synergy system can be empirically evaluated using different roots sets to understand the impact of roots selection on both the quality of the selected views and the concurrency control overhead.

- Chapter 5 – We examine the size-up and scale-up properties for all evaluated systems, we plan to also evaluate the speed-up property to create a more comprehensive understanding of performance characteristics in each system.

- Chapter 6 – For the design of the Fusion system we argue that the B+-tree indices are not a good fit; however, in the future, we would like to implement and evaluate B+-tree based indices to empirically justify this argument. In addition, we evaluate Fusion system performance with two different chunk sizes (16MB and 4MB); however, profiling with several different chunk sizes would create more data points and enable us to better understand the impact of chunk size on the system performance.

Chapter 8

Conclusion

Persistent growth in the use of mobile devices, social media platforms, gaming consoles, etc., combined with the ever-increasing online user population, has resulted in a data explosion. Traditional data management solutions are inadequate to meet the storage and processing challenges posed by such large and complex data. Although several large scale data management solutions have been proposed recently for each enterprise workload class, system architects still face several challenges in choosing or designing the right solution for their workloads. To this end, we propose mechanisms to enable scalable data management for each enterprise workload class.

8.1 Summary of Contributions

For OLTP workloads, we have developed mechanisms for scalable transaction processing in relational and distributed databases. For relational databases, we present a simple mechanism to stabilize the performance of cloud hosted databases. For distributed databases, we present the design of a novel data store that utilizes materialized views and a specialized concurrency control mechanism to enable scalable transaction processing. Our detailed contributions are as follows:

- **Relational Databases.** Profiling represents a standard technique to build a database performance model. However, relational databases hosted on the virtualized cloud platforms exhibit performance variations. The observed performance variations raise the challenge of creating a consistent and repeatable profile. To this end, we present a black box approach based on the database population from a snapshot to reduce the perceived performance variations. The experimental evaluation shows that the

profile created for a database populated using a snapshot can be used for performance modeling up to a high (80%) CPU utilization value. We also validate our findings on the Amazon EC2 cloud platform.

- **Distributed Databases.** NoSQL databases are designed to scale out linearly on commodity hardware; however, they are limited by the slow join performance. To this end, we present Synergy system that leverages schema and workload driven mechanism to identify materialized views and a specialized concurrency control system on top of a NoSQL database to enable scalable data management with familiar relational conventions. Experimental results using the TPC-W benchmark show that, for a database populated with 1M customers, the Synergy system exhibits a maximum performance improvement of 80.5% as compared to other evaluated systems.

SQL-on-Hadoop and SQL-on-Object-Storage denote two different architectures for OLAP. For SQL-on-Hadoop systems, we study the impact of file formats, size-up and scale-up characteristics in four state-of-the-art SQL-on-Hadoop systems. For SQL-on-Object-Storage systems, we implement and evaluate the impact of block range indexes on the query performance. Our detailed contributions are as follows:

- **SQL-on-Hadoop.** We perform a comparative analysis of four state-of-the-art SQL-on-Hadoop systems (Impala, Drill, Spark SQL and Phoenix) using the Web Data Analytics micro benchmark and the TPC-H benchmark on the Amazon EC2 cloud platform. The TPC-H experiment results show that, although Impala outperforms other systems (4.41x – 6.65x) in the text format, trade-offs exists in the parquet format, with each system performing best on subsets of queries. A comprehensive analysis of execution profiles expands upon the performance results to provide insights into performance variations, performance bottlenecks and query execution characteristics.

- **SQL-on-Object-Storage.** Baseline SQL-on-Object-Storage systems are slow and

impractical to use. To this end, we present the design and implementation of Fusion system that harnesses block range indexes as a mechanism to improve the query performance in SQL-on-Object-Storage systems. We empirically evaluate the creation and querying overhead associated with two different block range index implementations in the Fusion system. Experiment results show that the use of S3 as a persistent storage medium for small interval tree based indexes, can achieve both acceptable performance and cost efficiency. In addition, the smaller chunk size may allow for more aggressive data skipping; however, the query performance may actually regress due to the high processing overhead associated with smaller chunks in query engines like Spark SQL.

BIBLIOGRAPHY

[1] How much data is generated every minute on social media? [online]. available: http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/.

[2] Apache Storm [online]. available: http://storm.apache.org/releases/current/powered-by.html.

[3] Fay Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 15–15, 2006.

[4] HBase. [online]. available: http://hbase.apache.org/.

[5] MongoDB. [online]. available: https://www.mongodb.org/.

[6] Apache Cassandra. [online]. available: http://cassandra.apache.org/.

[7] Foundation DB. [online]. available: https://foundationdb.com/.

[8] Robert Kallman et al. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, pages 1496–1499, 2008.

[9] James C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[10] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *VLDB '05*, 2005.

[11] MonetDB [online]. available: https://www.monetdb.org/home.

[12] Vertica [online]. available: https://www.vertica.com/.

[13] Teradata [online]. available: http://www.teradata.com/?langtype=1033.

[14] Redshift [online]. available: https://aws.amazon.com/redshift/.

[15] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 1990.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, 2004.

[17] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[18] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD '08*, 2008.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[20] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *SIGMOD '15*, 2015.

[21] Apache Hadoop [online]. available: http://hadoop.apache.org/.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03*, 2003.

[23] Marcel Kornacker et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR'15*, 2015.

[24] Apche drill [online]. available: https://drill.apache.org/.

[25] Amazon S3 - Simple Storage Service.

[26] Microsoft Azure Blob Storage. [online]. available: https://azure.microsoft.com/en-us/services/storage/blobs/.

[27] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD '16*, 2016.

[28] Amazon elastic compute cloud (amazon ec2). [online]. available: http://aws.amazon.com/ec2/.

[29] Amazon Relational Database Service [online]. available: https://aws.amazon.com/rds/.

[30] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD '13*, 2013.

[31] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2), September 2010.

[32] Apache Drill: Interactive Ad-Hoc Analysis at Scale [online]. available: http://online.liebertpub.com/doi/pdf/10.1089/big.2013.0011.

[33] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[34] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 1994.

[35] Top 5 Reasons for Choosing S3 over HDFS. [online]. available: https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html.

[36] Apache Kudu. [online]. available: http://getkudu.io/.

[37] Apache Phoenix. [online]. available: http://phoenix.apache.org/.

[38] Thepra. [online]. available: http://tephra.incubator.apache.org/.

[39] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM'10*.

[40] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In *CLOUDCOM '10*.

[41] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys '10*.

[42] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*

[43] M. Suhail Rehman and Majd F. Sakr. Initial findings for provisioning variation in cloud computing. In *CLOUDCOM '10*.

[44] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *SoCC '12*.

[45] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *ICAC '05*.

[46] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

[47] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *DIS*, pages 158–169, 1996.

[48] José A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3), September 1989.

[49] Parag Agrawal et al. Asynchronous view maintenance for vlsd databases. In *SIGMOD*, pages 179–192, 2009.

[50] Jingren Zhou, Per-Ake Larson, Jonathan Goldstein, and Luping Ding. Dynamic materialized views. In *ICDE*, pages 526–535, 2007.

[51] Per-Ake Larson and H. Z. Yang. Computing queries from derived relations. In *VLDB*, pages 259–269, 1985.

[52] H. Z. Yang and Per-Ake Larson. Query transformation for psj-queries. In *VLDB*, pages 245–254, 1987.

[53] Jonathan Goldstein and Per-Ake Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, pages 331–342, 2001.

[54] Sanjay Agrawal et al. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[55] Accumulo. [online]. available: https://accumulo.apache.org/.

[56] Sattam Alsubaiee et al. Asterixdb: A scalable, open source bdms. *Proc. VLDB Endow.*, pages 1905–1916, 2014.

[57] Sudipto Das et al. G-store: A scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[58] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *IEEE Transactions on Services Computing*, 5(4):525–539, Oct-Dec 2012.

[59] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Scalable join queries in cloud data stores. In *CCGrid*, pages 547–555, 2012.

[60] Hoang Tam Vo et al. Towards elastic transactional cloud storage with range query support. *Proc. VLDB Endow.*, pages 506–514, 2010.

[61] Sudipto Das et al. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, April 2013.

[62] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 1–15, 2010.

[63] Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[64] Jeff Shute et al. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, pages 1068–1079, 2013.

[65] VoltDB. [online]. available: https://voltdb.com/.

[66] Justin DeBrabant et al. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, pages 1942–1953, 2013.

[67] Bin Liu et al. Automatic entity-grouping for OLTP workloads. In *ICDE*, 2014.

[68] Carlo Curino et al. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, pages 48–57, 2010.

[69] Davi E. M. Arnaut, Rebeca Schroeder, and Carmem S. Hara. Phoenix: A relational storage component for the cloud. In *CLOUD*, pages 684–691, 2011.

[70] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09*, 2009.

[71] Avrilia Floratou, Nikhil Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. Can the elephants handle the nosql onslaught? *Proc. VLDB Endow.*, 5(12):1712–1723, August 2012.

[72] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proc. VLDB Endow.*, 7(12):1295–1306, August 2014.

[73] AmpLab Big Data Benchmark [online]. available: https://amplab.cs.berkeley.edu/benchmark/.

[74] Stefan van Wouw et al. An empirical performance evaluation of distributed sql query engines. In *ICPE'15*, 2015.

[75] Pouria Pirzadeh et al. Bigfun: A performance study of big data management system functionality. In *IEEE BIG DATA*, 2015.

[76] Juwei Shi et al. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *in Proc. VLDB Endow.*, 2015.

[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, 2012.

[78] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1), August 2009.

[79] Microsoft Azure Cloud Computing Platform and Services. [online]. available: https://azure.microsoft.com/en-us/.

[80] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC '14*, 2014.

[81] Alluxio [online]. available: https://www.alluxio.org/.

[82] Block Range Index [online]. available: https://www.postgresql.org/docs/9.5/static/brin-intro.html.

[83] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB '98*, 1998.

[84] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD '14*, 2014.

[85] Ioannis Patlakas, George Sfakianakis, Peter Triantafillou, and Nikos Ntarmos. Interval indexing and querying on key-value cloud stores. In *ICDE '13*, 2013.

[86] TPC-W benchmark. [online]. available: http://www.tpc.org/tpcw/.

[87] Prabhakar Chaganti. *Xen Virtualization: A fast and practical guide to supporting multiple operating systems with the Xen hypervisor*. Packt Publishing, 2007.

[88] Pengcheng Xiong et al. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, pages 87–98, 2011.

[89] Yuan Chen, Subu Iyer, Dejan Milojicic, and Akhil Sahai. A systematic and practical approach to generating policies from service level objectives. In *IM*, pages 89–96, 2009.

[90] Mysql: Explain syntax. [online]. available: http://dev.mysql.com/doc/refman/5.0/en/explain.html.

[91] Mysql: The innodb buffer pool. [online]. available: http://dev.mysql.com/doc/refman/5.5/en/innodb-buffer-pool.html.

[92] CAP theorem. [online]. available: https://en.wikipedia.org/wiki/cap_theorem.

[93] Themis. [online]. available: https://github.com/xiaomi/themis.

[94] Omid. [online]. available: https://github.com/yahoo/omid.

[95] PostgreSQL: Transaction Isolation. [online]. available: http://www.postgresql.org/docs/9.1/static/transaction-iso.html.

[96] Kay Ousterhout et al. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[97] D. J. DeWitt. The Wisconsin Benchmark: Past, present and future. In J. Gray, editor, The Benchmark Handbook. Morgan Kaufmann, 1993.

[98] Apache HBase Performance Tuning. [online]. available: https://www.slideshare.net/lhofhansl/h-base-tuninghbasecon2015ok.

[99] Multi-threaded query execution in Impala. [online]. available: https://issues.apache.org/jira/browse/impala-3902.

[100] Google Cloud Platform. [online]. available: https://cloud.google.com/.

[101] MapR-FS. [online]. available: https://mapr.com/products/mapr-fs/.

[102] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.