UNIVERSITY OF CALIFORNIA,
IRVINE

On the Performance Evaluation of Big Data Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Pouria Pirzadeh

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Guoqing (Harry) Xu
Doctor Till Westmann

2015

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

CURRICULUM VITAE **ix**

ABSTRACT OF THE DISSERTATION **x**

1 Introduction **1**
- 1.1 Contributions of this work . . . 4
- 1.2 Organization of this dissertation . . . 5

2 Performance Evaluation of Key Value Stores **6**
- 2.1 Overview . . . 6
- 2.2 Motivation . . . 7
- 2.3 Related Work . . . 10
  - 2.3.1 Benchmarking cloud serving systems . . . 11
  - 2.3.2 Range queries in distributed environments . . . 12
- 2.4 Key Value Stores . . . 13
  - 2.4.1 Range query in HBase and Cassandra . . . 15
- 2.5 Range Query in Hash Partitioning Key Value Stores . . . 16
  - 2.5.1 Index-based Technique . . . 17
  - 2.5.2 No-Index Technique . . . 21
  - 2.5.3 Hybrid Technique . . . 22
- 2.6 Framework . . . 23
  - 2.6.1 Data and workload generator . . . 23
  - 2.6.2 Workload executor and Key Value store clients . . . 24
  - 2.6.3 Evaluation process . . . 25
- 2.7 Experiments . . . 26
  - 2.7.1 Experimental setup . . . 26
  - 2.7.2 Effect of node capacity in VIX and RIX . . . 29
  - 2.7.3 Range Selectivity . . . 30
  - 2.7.4 Mixed query workloads . . . 31

2.7.5 Impact of hybrid schemes on other operations . . . 32
2.7.6 Lookup and update scalability . . . 36
2.7.7 Range query scalability . . . 38
2.7.8 Non-uniform data test . . . 42
2.8 Discussions . . . 45
2.8.1 Summary of the performance results . . . 45
2.8.2 Range index support by hash-partitioning Key Value stores . . . 47
2.9 Conclusion and Future Work . . . 49

3 Performance Evaluation of Big Data Management System Functionality 50
3.1 Overview . . . 50
3.2 Motivation . . . 51
3.3 Related Work . . . 53
3.4 Systems Overview . . . 55
3.4.1 System-X . . . 55
3.4.2 Apache Hive . . . 55
3.4.3 MongoDB . . . 56
3.4.4 AsterixDB . . . 57
3.5 Data and Workload Description . . . 57
3.5.1 Database . . . 58
3.5.2 Read-Only Queries . . . 65
3.5.3 Data Modification Operations . . . 80
3.6 Experiments . . . 81
3.6.1 Setup . . . 82
3.6.2 Read-Only Workload Results . . . 84
3.6.3 Data Modification Workload Results . . . 91
3.7 Discussion . . . 97
3.8 Conclusion . . . 102

4 Performance Evaluation of Big Data Analytics Platforms 104
4.1 Overview . . . 104
4.2 Motivation . . . 105
4.3 Related Work . . . 107
4.4 Systems Overview . . . 108
4.4.1 Apache Spark . . . 108
4.4.2 Apache Tez . . . 109
4.4.3 Storage Formats . . . 110
4.5 TPC-H Benchmark . . . 113
4.5.1 TPC-H Database . . . 113
4.5.2 Queries . . . 113
4.5.3 TPC-H Auxiliary Index Structures . . . 115
4.6 Experiments . . . 116
4.6.1 Experimental Setup . . . 116
4.6.2 Experimental Results . . . 119
4.7 Selected queries . . . 129

4.7.1 Query 1 . . . 129
4.7.2 Query 10 . . . 130
4.7.3 Query 19 . . . 133
4.7.4 Query 22 . . . 136
4.8 Discussion . . . 140
4.9 Conclusion . . . 142

5 Conclusions and Future Work **143**
5.1 Conclusion . . . 143
5.2 Future work . . . 145

Bibliography **147**

A TPC-H Queries in AQL **154**

LIST OF FIGURES

Page

2.1 BLink Tree on top of Voldemort 21
2.2 System Architecture 23
2.3 Effect of node capacity on insert 30
2.4 Effect of node capacity on lookup and update time (VIX scheme) 30
2.5 Selectivity test - Average time for a range query 31
2.6 Mixed range queries workloads 33
2.7 Lookup time for different systems and schemes 34
2.8 Update time for different systems and schemes 35
2.9 Insert time for different systems and schemes 35
2.10 Lookup time in different systems 36
2.11 Lookup throughput in different systems 37
2.12 Update time in different systems 37
2.13 Update throughput in different systems 37
2.14 Multi client test - Short range queries response time 39
2.15 Multi client test - Short range queries throughput 40
2.16 Multi client test - Medium range queries response time 40
2.17 Multi client test - Medium range queries throughput 40
2.18 Multi client test - Long range queries response time 41
2.19 Multi client test - Long range queries throughput 41
2.20 Multi client test - Impact of fanout and client threads - Medium range queries 41
2.21 Non-uniform data test - Response time for different range query lengths . . . 43
2.22 Non-uniform data test - Short range queries response time 43
2.23 Non-uniform data test - Short range queries throughput 44
2.24 Non-uniform data test - Medium range queries response time 44
2.25 Non-uniform data test - Medium range queries throughput 44
2.26 Non-uniform data test - Long range queries response time 45
2.27 Non-uniform data test - Long range queries throughput 45

3.1 Nested BigFUN schema 60
3.2 Normalized BigFUN schema 63
3.3 BigFUN operations 67

4.1 AsterixDB scale-up on external datasets 123
4.2 AsterixDB scale-up on internal datasets 124
4.3 Spark SQL scale-up on the text format 125

4.4 Spark SQL scale-up on the Parquet format 126
4.5 Hive scale-up on MR and the text format 127
4.6 Hive scale-up on MR and the ORC format 127
4.7 Hive scale-up on Tez and the text format 127
4.8 Hive scale-up on Tez and the ORC format 128
4.9 System-X scale-up 128
4.10 Query 10 plan - System-X on 9 partitions 132
4.11 Query 10 plan - System-X on 27 partitions 133
4.12 Query 19 plan - Spark SQL 135
4.13 Query 19 plan - AsterixDB 135
4.14 Query 22 plan - Spark 139
4.15 Query 22 plan - System-X 139
4.16 Query 22 plan - AsterixDB 140

LIST OF TABLES

Page

2.1 DQ-Gen parameters 24
2.2 Different Query Processing Schemes on Voldemort (V-API stands for Voldemort built-in API and IX-API stands for our implemented Index API) . . . 29
2.3 Mixed range queries response time for Figure 2.6 (in ms) 33
2.4 Operations time for figures 2.7 to 2.9 (in ms) 35

3.1 BigFUN operations description 66
3.2 Total database size (in GB) 82
3.3 Nested schema - Secondary index structures 85
3.4 Normalized schema - Index structures 85
3.5 Read-only queries - Average response time (in sec) - Part 1 92
3.5 Read-only queries - Average response time (in sec) - Part 2 93
3.5 Read-only queries - Average response time (in sec) - Part 3 94
3.6 Data modification - Avg response time (in ms) 97

4.1 TPC-H schema, Primary keys 115
4.2 TPC-H tables size (in GB) - SF 150 120
4.3 TPC-H tables size (in GB) - SF 300 120
4.4 TPC-H tables size (in GB) - SF 450 120
4.5 TPC-H tables loading time (in sec) 121
4.6 TPC-H queries response time (in sec) - SF 150 124
4.7 TPC-H queries response time (in sec) - SF 300 125
4.8 TPC-H queries response time (in sec) - SF 450 126

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my academic advisor Prof. Michael J. Carey for his trust and support during the past five years. It is not often that one finds an advisor with such a deep knowledge and insightful vision. With his unique character, patience, and strong research dedication, he has been a source of inspiration throughout the course of my graduate studies. It has been him who taught me how to find the right problem and the correct approach to solve it and most importantly leaving no stone unturned by paying enough attention to all the details. I have had the privilege to work with a smart, critical thinker and I am deeply indebted to Prof. Carey for his guidance.

I would like to thank Dr. Till Westmann. I have been amazingly fortunate to benefit from his advice. He has been like a second advisor to me, ready with brilliant ideas, and encouraging words. From him, I learned how to do disciplined research. His support has been an invaluable source of confidence for me throughout my studies.

I would like to thank Prof. Chen Li and Prof. Harry Guoqing Xu for being on the dissertation committee. Prof. Li introduced me to a wider range of scenarios that could benefit from the work done as part of my dissertation.

I would like to thank Dr. Hakan Hacigumus and Dr. Junichi Tatemura for the wonderful mentoring during my internship at NEC Labs America. That internship indeed was the start of my journey in the Big Data world and it directly contributed to this dissertation.

I would like to thank Oracle Labs for their generous support over a period of time in my graduate studies. They gave me the opportunity to work with new technologies and gain insight into them which had an impact on the contents of this dissertation.

I would like to thank certain people affiliated with the System-X vendor for their valuable help on parts of this work.

I would like to thank all other AsterixDB team members and alumnus, specifically Dr. Raman Grover and Dr. Yingyi Bu, for working hard together to build the systems, make them usable, and create an opportunity for everyone's contributions here and there.

The work reported in this thesis has been supported by a UC Discovery grant, by NSF IIS awards 0910989 and 1059436, and by NSF CNS award 1305430. In addition, the AsterixDB project has benefited from generous industrial support from Amazon, eBay, Facebook, Google, HTC, InfoSys, Microsoft, Oracle Labs, and Yahoo.

I would like to thank my dear friend Mohammad Khorramzadeh for all the fun we had during happy moments and his support during difficult times.

Finally, I would like to thank my incredible mother, amazing father and wonderful brother. Without their love and continuous encouragement, I could not make this thesis.

CURRICULUM VITAE

Pouria Pirzadeh

EDUCATION

Doctor of Philosophy in Computer Science **2015**
University of California Irvine *Irvine, CA*

Masters of Science in Computer Science **2009**
University of California Irvine *Irvine, CA*

Bachelor of Science in Computer Engineering **2006**
Sharif University of Technology *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Student Researcher **2009–2015**
University of California, Irvine *Irvine, California*

Research Assistant **2013–2015**
Oracle Labs *Belmont, California*

Research Assistant **2010**
NEC Labs America *Cupertino, California*

ABSTRACT OF THE DISSERTATION

On the Performance Evaluation of Big Data Systems

By

Pouria Pirzadeh

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Michael J. Carey, Chair

Big Data is turning to be a key basis for the competition and growth among various businesses. The emerging need to store and process huge volumes of data has resulted in the appearance of different Big Data serving systems with fundamental differences. Big Data benchmarking is a means to assist users to pick the correct system to fulfill their applications' needs. It can also help the developers of these systems to make the correct decisions in building and extending them. While there have been several major efforts in benchmarking Big Data systems, there are still a number of challenges and unmet needs in this area. This dissertation is aimed at contributing to the performance evaluation of Big Data systems from two major aspects. First, it uses both new and existing benchmarks to do deep performance analysis of two major classes of modern Big Data systems, namely NoSQL request-serving systems and Big Data analytics platforms. As its second contribution, this dissertation looks at Big Data benchmarking from a new angle by comparing Big Data systems with respect to their features and functionality. This effort is specifically important in the context of increasing interest in using a unified system which is rich in functionality to serve different types of workloads, especially as the Big Data applications evolve and become more complex.

Chapter 1

Introduction

The Big Data wave has changed the way that businesses operate and people live. We are observing an explosion of data in various domains and it is no longer tempting, but necessary, to store and analyze this data in order to help organizations make correct decisions and move forward. The *volume*, *variety*, and *velocity* of this data and rapid growth of large-scale, data-intensive applications have created an urgent need to build platforms to store, manage, process, and visualize the Big Data efficiently and in a scalable manner. On one end of the modern Big Data systems spectrum, we have NoSQL systems for serving fast and concurrent (simple) requests, and on the other end we see Big Data analytics platforms for batch-oriented and computationally heavy types of workloads.

Various approaches exist for building Big Data platforms, which result in these systems being different from one another in terms of the architecture, data representation model, query support, storage, and level of guarantee that they provide for consistency and transactions. These differences are mainly due to the inherent needs of Big Data applications for massive (horizontal) scaling, high availability, and high performance. Traditional database systems have little or no ability to serve such applications with the level of demands that they

have, and new Big Data systems often decide to sacrifice or relax the full ACID (Atomicity, Consistency, Isolation and Durability) properties that have been provided by RDBMSs for a long time in favor of applications' demands. Moving from strict to "eventual" consistency (in Cassandra [56] or DynamoDB [75]) to achieve high availability is an example of such a decision.

In this situation, a major challenge in the Big Data world is comparing the available systems to each other. It is critical to have mature Big Data benchmarks to systematically evaluate Big Data systems from different aspects as a means of helping both customers (in choosing proper system(s) for their use cases) and vendors (in driving their efforts in building, hardening and solidifying their technologies).

While the large body of existing works (from both the industry and academia) in benchmarking traditional relational databases seems relevant and inspiring for the Big Data benchmarking, it is not sufficient to fulfill the needs of the Big Data world. The Big Data era has brought new challenges in benchmarking:

- **Diversity in data:** As the variety and velocity of the generated data items increase, data records in Big Data applications tend to be more and more heterogeneous and conform less to any strict schema. While many Big Data systems decide to opt for storing and processing unstructured data, there are systems that pick "semi-structured" data models to benefit from the amount of structure (if any) that exists in data items when storing them and processing queries. Moreover, each data item itself tends to have a more complex structure by including attributes of richer types such as temporal, spatial or collections and also attributes with nesting. A mature Big Data benchmark is expected to reflect these properties in its data and has to offer reasonable metrics to evaluate and compare Big Data systems on this aspect.

- **Diversity in workloads:** Considering the variety in major Big Data applications such

as social networks, e-commerce, or Internet of things (IoT), we are observing dominant workloads with different characteristics (such as simple, short read and write operations in the OLTP-style workloads, or batched, heavy analytical operations in the OLAP-style workloads). Making reasonable assumptions about the workload characteristics and covering this diversity in workloads is a necessary requirement for any Big Data benchmark so that it can usefully serve by different customers and vendors.

- **Diversity in performance metrics:** Different Big Data applications have different requirements for their performance. While traditionally and from the RDBMS technologies era, response time and throughput have been the major metrics to quantify the performance of a data store, the evolution of Big Data systems has added new dimensions to the comparison of their performance such as energy efficiency and cost effectiveness [74]. As a result, a comprehensive Big Data benchmarking effort needs to evaluate and compare its target Big Data systems from such perspectives as well.

Major efforts in Big Data benchmarking started with works such as [86] and [61]. To date, most of the works in this area have a narrow focus and can be mainly categorized as "functional benchmarks" (such as Terasort) which focus on a well-defined but very specific function, or "data-genre benchmarks" (such as Graph 500) for specific genres of data, or "micro-benchmarks" (such as the one in [86]) which focus on a small set of application-specific operations [46]. Given the current state of these works and the above challenges, however there are still unmet needs [52] that need to be fulfilled by a new generation of richer benchmarks developed specifically to explore and evaluate various aspects of Big Data systems and applications.

1.1 Contributions of this work

This dissertation focuses on the issues and challenges in the performance evaluation of Big Data systems. The major contributions of this work can be listed as:

1) A detailed study of the performance of representative Big Data systems picked from different parts of the Big Data platform spectrum. For this purpose, we consider major categories of workloads in Big Data applications and we use existing and new benchmarks to study and compare various Big Data systems to each other.

2) The design and implementation of novel and extensible micro-benchmarks which address important characteristics of data and application-specific workloads for conducting accurate benchmarking of different Big Data systems.

3) Converging the tasks of performance evaluation and functionality evaluation of Big Data systems in order to explore a "one size fits a bunch" conjecture which argues that one system with a larger feature set can perform well vs. multiple systems stitched together, each opting for providing a limited functionality with a narrow set of features. This point is important in the context of the arising interests in using a unified system (rather than a number of specialized systems) to serve different storage and processing needs (e.g., [36]).

4) Deriving practical guidelines and rules to assist in the process of designing, implementing and expanding Big Data systems. This step is achieved through careful and systematic analysis of the obtained results and identifying major performance bottlenecks and the trade-offs that may exist between picking a specific architecture and design versus specific operations.

5) Uncovering functional and performance-related issues in Big Data systems. A major part of this work has been done in the context of AsterixDB project. AsterixDB is a (new) full-function open source Big Data management system for ingesting, storing and querying semi-structured data. An important side-effect of this work has been identifying a number

of issues in different layers and components of AsterixDB which triggered efforts to address and fix them.

1.2 Organization of this dissertation

The remainder of this thesis is organized as follows. Chapters 2 to 4 are three core chapters that look at the performance evaluation of Big Data systems from different perspectives. Chapter 2 looks at the performance evaluation of NoSQL systems using different schemes and against mixed workloads of requests in Key Value stores with an emphasis on range queries. Chapter 3 aims at studying and comparing alternative Big Data systems with respect to the set of features that they support and the level of performance that they offer for them. Chapter 4 studies the performance of Big Data analytics frameworks for complex analytics operations which are computationally heavy and potentially I/O intensive. Finally, Chapter 5 concludes the thesis.

Chapter 2

Performance Evaluation of Key Value Stores

2.1 Overview

This chapter of the thesis focuses on the first major type of workload in the Big Data world: operations that are included in "cloud OLTP" applications whose requests normally consist of small online and concurrent read and write requests to the data. Recently there has been a considerable increase in the number of available Key Value stores for supporting data storage and applications in the cloud environment. While all these stores aim to offer highly available and scalable services in the cloud, they are significantly different from each other in terms of their architectures and the types of applications that they try to support. Such a variety makes it difficult for end users to pick the proper system to store their data and process the workloads that they have. In this Chapter, we consider three widely-used such systems: Cassandra [56], HBase [76] and Voldemort [88]. We compare them in terms of their support for different types of query workloads; We focus mainly on their ability to support

range queries. HBase and Cassandra have built-in support for range queries. Voldemort does not support this type of queries via its available API. To address that, we present practical techniques on top of Voldemort to support range queries. Our performance evaluation is based on a micro-benchmark which includes mixed query workloads, in the sense that they contain a combination of short and long range queries, as well as other types of typical queries on Key Value stores such as lookup (Get() - full value retrieval according to a given key) and update (Put() - modifying the value associated with a given key). We show that there are trade-offs in the performance of the selected systems and schemes, and we explore the types of query workloads that can be processed efficiently.

2.2 Motivation

Cloud computing is expected to provide large-scale applications with elasticity of resources through an approach called *scaling-out*, which is about dynamically adapting to growing workloads by increasing the number of servers. However, data-intensive applications using traditional relational database management systems (RDBMS) are hard to scale out since in a large, distributed cluster environment as consistency and availability are hard to achieve together in a scalable manner for interactive OLTP applications that require short response times.

Key Value stores have been increasingly adopted as an alternative to RDBMSs in order to scale out data-intensive applications. Key Value stores achieve better scalability and availability for a simpler set of workloads with reduced consistency support. There have been a number of different Key Value stores developed to meet various requirements. Examples include Google's BigTable [59], HBase [76], Cassandra [56], Voldemort [88], Couchbase [40], MongoDB [82], and Amazon's DynamoDB [75]. However, the large variety in their architectures makes it difficult to compare Key Value stores in an apples-to-apples manner.

To help understand the performance implication of the architectural differences of Key Value stores, Cooper et al. proposed a benchmark, called YCSB (Yahoo! Cloud Serving Benchmark) [61], which consists of several types of workloads that capture common characteristics of typical cloud OLTP applications.

In this chapter, we contribute to this effort on performance studies of Key Value stores by covering range queries extensively. The original workloads in YCSB include range queries in a very limited fashion since range queries were not very common in the usage of Key Value stores. However, we envision an increasing need for integrated support of transactional and analytic processing in the same data store, which calls for more extensive use of range queries and other operations.

In a traditional enterprise environment, transaction processing and analytic processing are typically done in different stores (operational data stores and data warehouses, respectively). Although this separation is still valid in cloud environments, we see the following cases where transactional stores should take some part of analytic processing in addition:

- *Real-time analytics*: To achieve better customer satisfaction or better security, web application service providers need to react to the real-time situations among customers. An analytic process should be able to access the real-time data as well as archived data.
- *Decision support for end users*: Recent web applications support richer user interactions including recommendations and powerful navigation. Such applications can be seen as simple "decision support systems" for end users, although queries may not be as complex as those in enterprise decision support systems.
- *Platform-as-a-Service*: Key Value stores are not only used for large-scale applications that occupy a cluster of machines, but also for large-scale *platforms* that host a large number of smaller applications. Google App Engine [70] is one such platform. In such a case, building data warehouses for these small applications might be overkill.

As a result, supporting range queries on Key Value stores for cloud OLTP applications will complement traditional ETL-based approaches. Notice that this complementary approach does not necessarily call for complete integration of OLTP and OLAP as a *one size fits all* data store. For traditional, batch-oriented OLAP, employing separate data stores dedicated for data warehouse use can make more sense. Our research goal here is to investigate how far we can push range query workloads to Key Value stores (e.g., for better interactivity or data freshness) in such a complementary environment.

In this chapter, we are especially interested in interactive applications with range queries (rather than Map-Reduce applications over Key Value stores). This setting is similar to YCSB except that we introduce longer range queries.

In fact, the degree to which Key Value stores can take part in analytic processing will depend on the architectural decisions on individual Key Value store implementations. For instance, HBase, an open source implementation of Google's BigTable, is in fact used for analytic processing as well: it is often used as one of data sources of Map-Reduce jobs. In that sense, HBase (BigTable) is a Key Value store that integrates transactional and analytic processing. As YCSB's paper reported, however, HBase underperforms other Key Value stores in terms of read response time for simple key lookups. Thus, we should regard HBase (BigTable) as one design choice for particular trade-off requirements and explore other architectures as well.

To explore design choices extensively, we have studied the implementation of range query support on Key Value stores that employ hash-partitioning. Whereas Key Value stores that employ range-partitioning (e.g. BigTable) natively support range queries, hash-partitioning Key Value stores (e.g. Voldemort) do not support range query APIs. However, hash-partitioning Key Value stores have their own benefits (e.g., easier load distribution compared to range-partitioning). Engineers should not have to give up this option only because it does not support range queries natively. In a traditional RDBMS, choosing the partitioning scheme

is a part of the physical database design process. We want to preserve hash-partitioning Key Value stores as an option to support such tradeoffs.

Implementing efficient range queries on hash-partitioning Key Value stores is not a trivial task, although the techniques used are not new. One solution is to construct range indexes using the data structure supported by such a Key Value store. For instance, Brantner et. al implement a relational database on top of Amazon's S3 [38] by implementing the BLink tree structure [79] with a disk page in the structure being emulated using a Key Value object [51]. Another solution is to directly access to all the data partitions in the storage nodes and consolidate the results, just as a shared-nothing parallel RDBMS does. In either case, additional overhead is a key concern. Extensive performance evaluation is required to investigate the viability of range query support on hash-partitioning Key Value stores.

In this chapter, we evaluate the performance of various alternative schemes to support range-query workloads (which include lookup and update queries as well) on top of open source Key Value stores: HBase, Cassandra, Voldemort. In particular, we explore schemes to support range queries on Voldemort, a hash-partitioning Key Value store. Experiments illustrate the trade-off among different schemes, indicating that there is no absolute winner. We discuss decision criteria for design choices based on the implications of this performance study.

2.3 Related Work

We can categorize the related work which looks at the performance of range queries into two major groups:

1. Works on benchmarking cloud data serving systems.

2. Works on processing range queries in distributed environments.

We describe each group in more details below.

2.3.1 Benchmarking cloud serving systems

Cooper et. al., in [61] have presented Yahoo! cloud serving benchmark (YCSB), an extensible framework for performance comparison of cloud data serving systems. YCSB consists of different tiers, each for benchmarking a major aspect of cloud systems. [61] also provides experimental results on the performance and scalability of PNUTS [60], Cassandra, HBase, and sharded MySQL.[1] One important distinction of our work and [61] is our focus on the range queries. Although the set of core workloads in YCSB contains a workload on short ranges (consisting of range scans that start from a random key, and retrieve up to 100 consecutive records), the type and context of range queries that we wanted to consider were broader. We have used range queries with a wide range of different selectivities, and also mixed workloads of them, to cover different sets of real scenarios. Thus, we decided to create our own data and query generator. Moreover, beside using the built-in range query support in Cassandra and HBase, we have developed and tested different techniques for supporting such queries in Voldemort (which originally was not among the considered Key Value stores in YCSB). Since both our system and YCSB emulate interactive (OLTP) workloads, one can naturally combine our results, with YCSB results: ours can be served as an extension to YCSB's results on range queries. We have included other operations (lookup, update and insertion) to make this coupling more meaningful. Possible future work is to integrate them as an extended YCSB implementation. Pavlo et. al., in [86] have done a comparison of Map-Reduce and parallel databases on a collection of tasks. Their work is mainly focused on batch processing types of jobs (such as OLAP applications). Among other works on benchmarking cloud serving systems [97] provides results on performance evaluation of a set of cloud-based data management systems (HBase, Cassandra, HadoopDB[31], and Hive) to

[1] YCSB is available as an open source package, and it has recently extended its set of clients with MongoDB and Voldemort clients, as well.

analyze different implementation approaches for various application environments. Shi et.al., in [97] mainly use two benchmarks, one focusing on data reads and writes, and the other on structured queries, while each benchmark is associated with several tasks. [100] presents *ecStore*, which is an elastic cloud storage system that is able to support range queries using a persistent B-Tree, beside its other features. Considering cloud data serving systems and their benchmarking in a more general aspect, Cattell in [57] presents a survey on comparing SQL and cloud-based data stores, grouped based on their data models into: Key Value stores, Document stores, Extensible record stores and relational databases. The comparison in [57] is done on a number of different dimensions, while the major focus is on the scalability of data stores for OLTP like applications. [48] discusses major guidelines and requirements for benchmarking and analyzing cloud services. Describing the reasons that make benchmarks for the traditional databases deficient for the newly emerging cloud-based data stores and services, the authors in [48] present the ideas to design new benchmarks, which capture main characteristics of these systems such as scalability and fault-tolerance, to obtain more meaningful and applicable results for real case scenarios.

2.3.2 Range queries in distributed environments

There is an extensive set of works on evaluating range queries in the distributed and P2P systems, such as publish/subscribe systems (pub/sub) or distributed hash tables (DHTs) [87][39][73][92]. One important issue that is widely studied in these works is the tradeoff between the efficiency of range query processing and load balancing. Different solutions have been offered for this problem, mostly based on replication, data migration, using clever hash functions to preserve locality, and distributed indices [44][67][78][90]. Ganesan et. al., in [68] have used traditional database techniques and proposed two systems, $SCRAP$ and $MURK$, based on space filling curves and space partitioning, that can support range queries. Schütt et. al., in [94] presented Chord#, which is a P2P protocol that is capable of supporting

range queries, using key-order preserving functions. In addition, [95] presents a routing scheme based on Chord#, named *SONAR*, to support multi-dimensional data and range queries. Cloud computing needs to address these concerns as well, and there is an overlap between the solutions, proposed in these two areas. For example, we can consider distributed index structures (as we used BLink tree on top of Voldemort) to process range queries. [33] proposes a fault tolerant and scalable distributed B-tree, with extended operations, based on distributed transactions. Lomet in [80] presents dPi-tree, which is a distributed and scalable index structure, capable of processing range searching in parallel.

2.4 Key Value Stores

There are many instances of Key Value stores available for managing the data and queries in Big Data applications. While these systems have significant differences with each other in terms of their architecture, they are all attempting at addressing important requirements that arise in cloud computing such as scaling-out, availability and high operational throughput. Because of the natural trade-off that exists in achieving these end goals, it is practically impossible to have a unique Key Value store which is optimal for all different types of applications and query workloads. In this chapter, we select three popular and widely used Key Value stores as the systems to explore: HBase [76], Cassandra [56] and Voldemort [88]. These systems considerably differ with each other in terms of their architecture, design decisions, and types of the applications they can serve the best. The following is a brief overview of them.

HBase: HBase [76] is an open source version of Google's Big Table [59], built on top of Hadoop file system (HDFS) [41]. An HBase cluster consists of a master node which manages the data store and one or more region server workers that are responsible to store Key Value pairs and process read and write requests. Key Value pairs (rows) are stored in byte-

lexicographical sorted order across distributed regions. In its data model, HBase uses the concept of *column-families*, which is a set of columns within a row that are typically accessed together. All the values of a specific column-family are stored sequentially together on the disk. Such a data layout results in fast sequential scans on the consecutive rows, and also on the adjacent columns within a column family in a row (partial column access). HBase relies on HDFS for durability and employs multi-versioning for Key Value pairs. The update operation is essentially implemented as appending data to files on HDFS (HBase uses an LSM-based [84] storage). Updates are once written into memory buffers, which are flushed periodically to the disk. As a result, writes are faster than reads which need (multiple) random I/Os for combining updates, corresponding to the row(s) of interest.

Cassandra: As a Key Value store, Cassandra [56] is mainly designed and implemented based on the data model in Big Table [59] and the architecture of Amazon's DynamoDB [75], a distributed Key Value store that uses hash-based partitioning for distributing the data across nodes. Using LSM-based storage and similar to HBase, Cassandra is also optimized for fast write operations. At the same time, Cassandra takes Dynamo's approach to durability and availability: it uses replication across multiple nodes along with hinted hand-off technique, while the cluster is configured as a *ring of nodes*. Cassandra also uses an *eventual consistency* model, in which the consistency level can be selected by the client. Unlike HBase, there is no master node in Cassandra, and a *gossip* mechanism is used to propagate the current state of the cluster nodes. Cassandra supports pluggable data partitioning scheme and lets the application developers choose an appropriate component (called partitioner). The version we used provides a range-partitioner in order to support range queries.

Voldemort: Project Voldemort [88] is an open source implementation of Amazon's DynamoDB. Choosing a simpler data model, compared to HBase and Cassandra, Voldemort tries to provide low-latency with high availability accesses to the data. Because of the hash-partitioning policy, there is no built-in support for range queries in Voldemort. Such a

decision is mainly made for achieving better load balancing. Moreover, Voldemort is flexible in terms of supporting different pluggable storage engines, such as Berkeley DB [47] and MySQL [83].

While we focus on the current status of range queries support in Key Value stores, we have selected the above three systems because of the fact that they are being used extensively in a number of known companies, such as Facebook and LinkedIn that serve a large number of clients. Moreover we believe that because of the major differences between their design and implementation, they can be considered as a representative set for the state-of-the-art in Key Value stores. An important design decision of our interest that should be considered here in more details is the Key Value pairs partitioner. The choice of the partitioner along with the way data is stored on the disk within the persistent storage determine the ability of a Key Value store to support range queries. Most of the hash-partitioning systems such as Voldemort do not support range queries directly. On the other hand, order-preserving partitioners enables such support, according to the way they distribute the data across the nodes. Considering these three systems, HBase only supports the order-preserving partitioner, as the rows should be stored in a sorted manner. Voldemort is only using the hash-based partitioner to achieve its load balancing goals. Cassandra is able to use either of the partitioners, based on the user's preference.

2.4.1 Range query in HBase and Cassandra

HBase and Cassandra support range queries via their proprietary APIs, which are different in the way they model these queries: Whereas HBase's Scan operation takes an interval identified by an inclusive start key and exclusive end key as a range query's parameter, Cassandra takes a pair of a start key and a count number, i.e., the (maximum) number of consecutive keys that are scanned.

In this chapter, we are interested in range queries with an interval, which is typical for analytic processing. Thus, we emulate the interval-based range queries on Cassandra by repeating count-based scan operations.

The current implementation of HBase's Scan operation retrieves Key Value pairs sequentially without parallelism, which is reasonable for traditional use cases:

1. An interactive application uses this sequential scan directly for very short range queries.
2. A batch-oriented application uses this sequential scan, combined with Map-Reduce framework, such that a long range is partitioned into smaller ones, each of which gets scanned by a mapper.

In this chapter, we are interested in interactive applications that issue longer range queries where the batch-oriented nature of Map-Reduce is not suitable. Accordingly we implemented our custom solution to improve the efficiency of range query processing in HBase by dividing the whole range query interval into a number of non-overlapping sub-intervals, and the query is processed concurrently across them.

2.5 Range Query in Hash Partitioning Key Value Stores

As described in the previous section, Voldemort does not support range queries. Thus, one needs to develop and use additional techniques and/or data structures beside the original architecture, to add that support. In this chapter, we use two different techniques, on top of Voldemort for this purpose: one based on indexing and the other based on direct access to the data nodes.

2.5.1 Index-based Technique

Our first technique is to build a range index on top of Key Value store where index nodes (or "disk pages") are implemented using Key Value pairs (e.g., [51]). A unique key is assigned to each index node, and a node is accessed by this key through the original API of the Key Value store. Similar to [51], we implement BLink tree [79], which is a distributed variant of B-Tree.

Concurrency control

The BLink tree, compared to the regular B-Tree, provides higher degrees of concurrency for different types of reading and writing operations. It has a simpler locking scheme compared to a typical B-Tree implementation, as it does not use read locks (and so it lets read operations not get blocked during their tree traversal). Moreover, any write operation holds locks on a constant number of nodes, at any given time. Specifically in addition to the leaf level nodes, in BLink tree each internal node (except the rightmost node at each level) has a link pointer to the next node at the same level. This extra link is provided as an additional way to access a specific node in the tree (other than the unique path that exists from root to each node), from its left sibling node. Upon splitting a node, which results in two new nodes, the link pointer of the first node points to the second node, while the link pointer in the second node is identical to the link pointer of the original node (prior to split). The link pointer is a quick fix to let the concurrent reads go through, once they access a node involved in the split process. In such a case, if a read operation is looking for a key that is greater than the highest key in a node, the read process simply needs to access the right node, by following the link pointer. This way, the read operation does not get blocked at the cost of probably an extra I/O. The insertion process is similar to the read process, as it starts going down the tree from the root to find the proper leaf node to put data into, but it also makes sure

to remember the rightmost node, it had visited, at each level. This way, the insert process has enough information to fix the tree if adding the new value results in (cascading) splits in the tree. An important point is that upon split, the link pointers get fixed and set first (prior to the parent pointers) to let the concurrent reads go through as described above.

Write operations

The remaining issue on concurrency is to manage multiple write operations. Instead of introducing any central coordinators, we want to let clients coordinate in a decentralized manner. The following is the underlying mechanism for our concurrency control of write operations:

- *Atomic update of a single Key Value pair*: Voldemort supports atomic updates on a single Key Value pair in a non-blocking manner, which is similar to a compare-and-swap (CAS) operation of modern CPUs. A write operation on a Key Value pair is associated with a version with which the object was read. An attempt to write a Key Value pair with an obsolete version will fail. When data is inserted into an index node, we always use this mechanism to serialize updates (read-and-write) operations on the same Key Value pair.

- *Locking a single Key Value pair*: An index node with over capacity must be split. Since it involves operations over multiple Key Value pairs, the atomic update support, described above, is not sufficient. We introduce a lock that is embedded as a flag in each index node, which itself is a Key Value pair. Any write operation is blocked when a lock is set. Notice that, however, read operation will not be affected by this lock.

Eventual Consistency

Unfortunately, Voldemort supports a CAS operation only when the replication factor is 1 (i.e., no replication). When data is replicated, conflicting write operations may result in concurrent versions (which is modeled as vector versions [75]). Handling this eventual consistency requires a way to reconcile such concurrent versions. It is future work and beyond the scope of what we present in this chapter to build a BLink tree on eventually consistent data stores. Here we only discuss possible approaches and challenges to support eventual concurrency.

- *Lock table for split operations.* Instead of embedding a lock flag, we can introduce a separate set of Key Value pairs that represents a mapping from an index node's ID to its lock status. We can employ different replication factors between the lock table (without replication) and the index (with replication) to enable CAS operations on the lock table. The lock is used to avoid conflicting split operations.

- *Reconciliation of concurrent writes.* The remaining cause of concurrent versions is a write operation on leaf nodes (i.e., inserting and deleting index entries). When a leaf node with concurrent versions is detected, concurrent insertions and deletions of index entries must be consolidated. Although this reconciliation logic looks straightforward, a challenge is to implement it *efficiently*, which is our future work.

In this chapter, we use Voldemort with no replication to avoid eventual consistency. For the same reason, we do not have BLink tree implementation on Cassandra, for which we cannot avoid eventual consistency.

Node caching

Notice that the concurrency gained from the extra links of the BLink tree enables the caching of internal nodes: using obsolete nodes that are cached only results in extra sibling node traversal. A cached node is invalidated when such sibling node traversal happens with its children. In our experiments, we employ internal node caching to avoid possible performance bottlenecks such as reading the root node repeatedly.

Leaf node schemes

There is one design choice in this indexing technique: whether to embed data objects (tuples) into index nodes (leaves). Alternatively, we can embed references (keys) to data objects into index nodes and store data objects as individual Key Value pairs in a separate store. We refer to these index schemes as *value-embedded index* (VIX) and *reference-embedded index* (RIX) respectively. In a traditional RDBMS, VIX is often chosen for primary key indexes since it can exploit disk I/O characteristics especially for fast table scans. In the case of Key Value stores, however, RIX can also be a viable option especially when the workload is primarily simple put/get operations and range queries take smaller fraction of the workload. The performance impact of VIX/RIX on put/get operations is discussed below.

Impact on put/get operations

Using the VIX technique, all the put and get operations on the Key Value pairs, which were previously done through the Voldemort API directly, needed to be done via the BLink tree. It means that even for a lookup or value update, we need to traverse the tree levels, based on the search procedure of BLink tree, to meet the proper key and its value (if existing). In contrast, for the RIX case, we can still use the Voldemort API to perform lookups and value

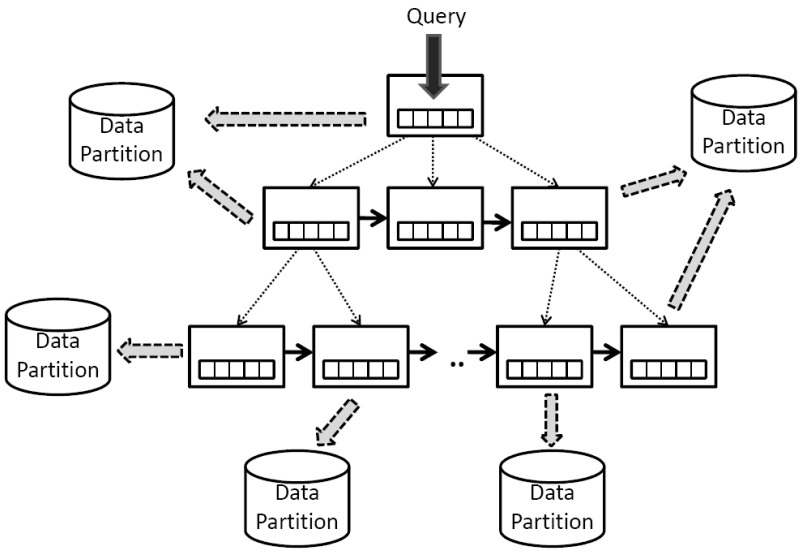

Figure 2.1: BLink Tree on top of Voldemort

updates, as the Key Value pairs are stored separately from the index structures.

Parallel query execution

For the RIX case, a range scan of leaf nodes only yields a set of references, with which data objects must be retrieved separately. We employ parallel retrieval of data objects to reduce the impact of extra latency. Index scan and object retrieval are done in a pipelined manner, and the object retrieval part is done by a pool of multiple threads.

2.5.2 No-Index Technique

An alternative solution to process range queries is to directly access all the data partitions in the storage nodes, run the range scan, and consolidate the results, just similar to the way that a shared-nothing parallel RDBMS does. In the case of a traditional parallel RDBMS, various operations (such as grouping and aggregation) are *pushed* to each and every node to execute a complex query efficiently. In this chapter we only consider simple range queries,

and what is pushed to a Key Value store node is very simple and limited (i.e., range selection over the keys). We refer to this technique as *No-Index technique (no-IX)*.

When we use this no-IX technique, we choose MySQL as the storage engine of Voldemort (we choose BerkeleyDB (BDB) otherwise), in order to implement node-local operations (range selection on each storage node). Using the MySQL interface, we implemented a range query API in addition to the original Voldemort API.

Therefore, in order to process a range query, one possibility is going through the following two-step procedure:

1. executing the query on each and every Voldemort node through its storage engine, in parallel, to obtain range query's "partial" results.
2. merging the partial results and returning the ultimate query answer.

Using the no-IX technique, all the *put* and *get* operations on the keys and values are simply done as before via the Voldemort API. But just for the range queries, we need to directly do the selection on the stored data in MySQL nodes.

2.5.3 Hybrid Technique

Later, in the experiments section, we show how the above two techniques (indexing and no-IX) can be compared with each other in different scenarios. But we should also point out a third option, which is actually a combination of the above two techniques. This option, which we call *Hybrid technique*, tries to benefit from both indexing and no-IX techniques. More specifically, in the hybrid technique, we try to maintain an index structure (VIX or RIX) beside the Voldemort store that hosts Key Value pairs. Considering a specific range scan, the ultimate technique for processing the query (using the index or going after no-IX)

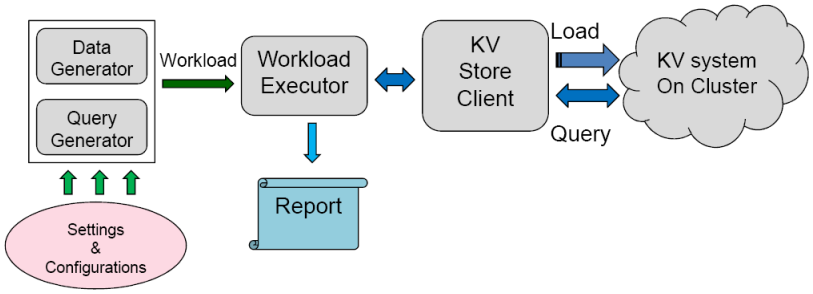

Figure 2.2: System Architecture

is actually chosen based on the "expected" selectivity of the query, i.e., expected number of Key Value pairs that need to be scanned for the query. While we show that we can improve the performance of range query processing using the hybrid approach in some scenarios (for example an update heavy workload); this approach has the extra cost of maintaining an index structure beside the Voldemort store.

2.6 Framework

In this section, we describe the architecture and features of our performance evaluation framework which consists of the following three main components: (1) *Data and workload generator*, (2) *Workload Executor*, and (3) *Key Value store client*. Figure 2.2 illustrates the architecture and interactions among the components.

2.6.1 Data and workload generator

The data and workload generator, which we call *DQ-Gen*, is mainly responsible for generating the raw Key Value pairs to be stored in datasets and the query workloads used for performance evaluation. They are generated based on a set of user-defined parameters. Table 2.1 lists the main parameters, which are needed to be set by the user. These parameters enable us to generate a wide range of different datasets and query workloads that are similar

Parameter	Description
Number of keys	Number of the Key Value pairs (loaded in store)
Keys distribution	Distribution of keys in [0, 1) space (uniform or Zipf)
Number of columns	Number of columns in the value associated to each key
Value size	Size of the value in each column (in bytes)
Query distribution	Distribution of query points (w.r.t. generated keys space)
Query selectivity	(for range queries) expected fraction of the key space to be retrieved by a query

Table 2.1: DQ-Gen parameters

and compatible to the actual scenarios. To generate the Zipfian distribution, our system utilizes the component in the YCSB [61] data generator tool, which employs the algorithm for generating a Zipfian-distributed sequence from Gray et al., [71]. In our case, we use a clustered distribution where the popular items are clustered together towards 0 (smaller values are more popular).

2.6.2 Workload executor and Key Value store clients

The workload executor is the component that interacts with the Key Value store client and guides the dataset loading and query workload execution. The most important criterion in the design and implementation of this component is its *extensibility*. Aiming at making the evaluation of various Key Value stores and techniques possible, workload executor defines an abstract API for the Key Value stores' clients, that need to interact with it. This API defines the main operations that should be supported by the client, so that the data loading and query execution becomes feasible for the workload executor. These main operations are:

- Insert: Adds a new Key Value pair to the dataset.
- Lookup: Fully retrieves an existing Key Value pair.

- Range Query: Fully retrieves a set of Key Value pairs whose keys fall within a given range.
- Update: Updates the stored value associated with an existing key.
- Delete: Removes an existing Key Value pair completely.

The exact implementation of each operation depends on the specific Key Value store that the user wants to evaluate.

2.6.3 Evaluation process

Once a specific Key Value store client is defined along with the input dataset and the query workload(s), the workload executor starts the evaluation as a *multi-client*, *multi-thread* process, based on the user defined settings. Within this process, data is first loaded (if needed) into the Key Value store. As the loading process normally takes a long time, this phase could be done as a batch process in the framework. The user has the ability to control and tune different parameters (such as batch sizes or number of loading threads) to make this process fast and efficient. At the end of the loading phase, the workload executor provides the user with a detailed report, containing metrics such as the average time to load one Key Value pair, the amount of load on each node, and the overall throughput of the Key Value store during the bulk load phase.

The next step for the evaluation is the query workload execution. In this phase, each query in the workload is executed via the Key Value store client in a multi-client and/or multi-thread manner. Performance metrics that are measured are the average, minimum and maximum response time for the queries and the overall throughput of the Key Value store during the query workload execution phase.

For this chapter, we implemented Key Value store clients for HBase, Cassandra, and Voldemort. For Voldemort, the clients consist of various implementations: Value-embedded Index (VIX), Reference-embedded Index (RIX), no-IX, and Hybrid. Details of the generated datasets and query workloads along with the obtained results are presented in the next section.

2.7 Experiments

In this section, we present the results of experiments conducted on our framework. We first give some details on our experimental setup, the versions of the systems that we used, and the datasets and query workloads that we generated.

2.7.1 Experimental setup

All the reported results are obtained on a six-server cluster. Each machine has an Intel Xeon E5620, 2.4 GHZ CPU with 4 cores, 16GB of memory and 1TB 7200 rpm disks used as the persistent storage partitions. Machines are connected to each other through a Gigabit Ethernet switch. For Cassandra, we used the stable 0.6.8 version and the HBase stable version was 0.20.6. The Voldemort version was 0.81. We used the periodic synchronization in Cassandra to achieve durability. We generated different datasets, with 1 million and 10 million rows, while each row had 5 columns. The total value size was up to 1KB, for each key. Keys were selected from the [0, 1) key space, with either uniform or non-uniform (Zipf) distribution. For the non-uniform key generation case, we used YCSB tool with the default Zipfian constant of 0.99. We have two major sets of experiments, conducted on these datasets: Uniform and non-uniform. For the uniform test case, we generated the following different sets of query workloads:

- Lookup-only and Update-only: Each consists of 5,000 keys, randomly selected form the corresponding dataset keys, where in the case of updates, each selected key is associated with a new value with the same size as the old value for that key.

- Range queries with selectivity of α: for a fixed value of α, varying between 0.0001 and 0.9 across different workloads, the query workload consists of 200 to 1000 range queries, where starting key of each query is selected randomly from the existing keys in the datasets, and the length of the range query is equal to the considered α.

- Mixed range queries: Consisting of 200 to 1,000 range queries with different selectivities. We have 3 types of mixed range queries: short, medium and long. For the case of mixed short range queries, query's selectivities vary between 0.0001 and 0.002. For the medium they vary between 0.002 and 0.1, and mixed long range queries have selectivities greater than 0.1.

For the non-uniform test case, in which we wanted to study the performance of techniques developed on top of Voldemort in more depth, we generated the following two query workloads:

- Range queries with length of ℓ: This workload consisted of 15 sets of queries, each containing 50 range queries of a fixed length ℓ, where ℓ varied between 0.001 to 0.9.

- Mixed range queries: This workload contained 3 sets of range queries: short, medium, and long, each consisting of 200 range queries whose lengths were selected from a pre-specified range of lengths. The length ranges for these sets were the same as the selectivity intervals, used for the uniform mixed range queries (described above).

We performed the experiments against a Key Value store with warmed-up cache. For this purpose, each query workload had also a corresponding *warmup* workload, consisting of up to 10% of the queries in the original workload. Our experiments on a specific query

workload was always preceded by running the warmup workload (before we start measuring the performance metrics), to make sure that we are fair with respect to the starting conditions of the simulations.

Our client machines, which were responsible for issuing the queries to the Key Value store, were always selected from a separate set of machines, other than those serving as the Key Value store nodes. This way we could make sure that we were considering all the expected delays, while the server and client processing resources are also separate.

As explained before, Voldemort does not support range queries and we considered different techniques, based on indexing and direct access to the storage to add such support. Table 2.2 summarizes different schemes that we used. In this table, we specify how each type of query is processed, using each scheme. Specifically we mainly use the Voldemort built-in API (shown as V-API) for doing lookups and updates. For the range queries, the specific technique of use depends on the type of the index that we are using. For the value-embedded (VIX) and reference-embedded index (RIX) cases we need to scan index tree nodes to find the qualifying keys (shown as IX-Scan in Table 2.2). If using VIX, then corresponding values are also embedded within the index nodes. However, for the RIX case, we need to perform a sequence of lookups, on a separate store containing Key Value pairs, to obtain the values. For the case of no-IX scheme, we execute the range queries directly on the storage servers (in our case MySQL servers) within the cluster nodes. This process is denoted by *storage scan* in Table 2.2.

For the experimental results, we report the average response time (for each query), and the average throughput (during the whole query execution process). As a result, because of the inherent tradeoff between the response time and throughput, different systems can be compared with each other with respect to a fixed set of resources. We have categorized our experiments into different groups, based on the systems, datasets, and metrics that were studied.

Scheme	Fanout	Lookup	Update	Range Query
Voldemort API	-	V-API	V-API	-
No-IX	-	V-API	V-API	Storage Scan
VIX	70	IX-API	IX-API	IX-Scan
RIX	70	V-API	V-API	IX-Scan and V-API
Hybrid-VIX	70	V-API	IX-API and V-API	IX-Scan or Storage Scan
Hybrid-RIX	70	V-API	V-API	IX-Scan and V-API or Storage Scan

Table 2.2: Different Query Processing Schemes on Voldemort (V-API stands for Voldemort built-in API and IX-API stands for our implemented Index API)

2.7.2 Effect of node capacity in VIX and RIX

The first set of experiments shows the impact of choosing different node capacities, i.e., the node fanout, when using VIX or RIX techniques in Voldemort. Figures 2.3 and 2.4 show the average response time for the insertion, lookup, and update. Increasing the node capacity, the size of the index objects that are stored in Voldemort gets larger, which results in reducing the height of the index tree. As a result, the total cost of insertion or update increases, since we are modifying (reading and writing) larger objects in the Voldemort store. In fact, using VIX, the unit of values that are stored or accessed via the put/get API is of the size of the index nodes. Using larger node capacities, the average size of an index node that is accessed increases. But in this case, there is a gain for the lookup, as its cost gets lower, given the fact that we need to traverse fewer tree levels. In the case of RIX, lookup and update can be done via the Voldemort-API, because we just need to directly access a data object with its key (i.e., no need for index tree traversal).

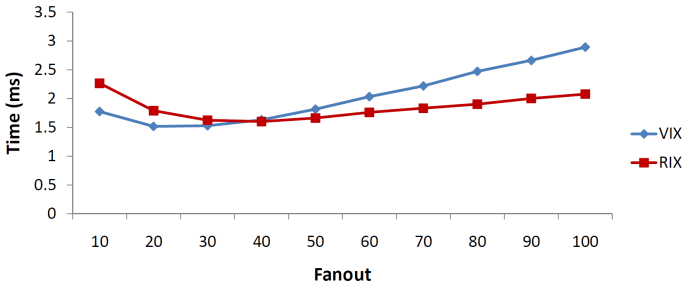

Figure 2.3: Effect of node capacity on insert

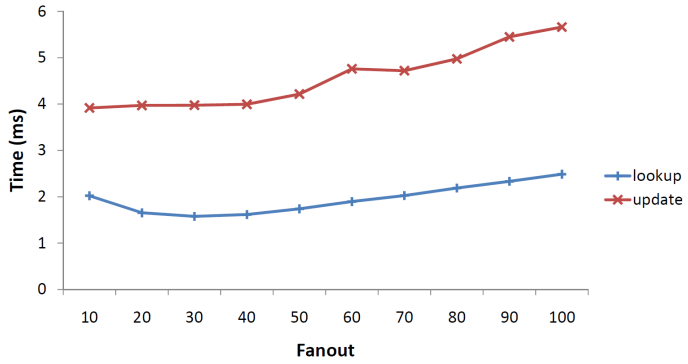

Figure 2.4: Effect of node capacity on lookup and update time (VIX scheme)

2.7.3 Range Selectivity

The second set of experiments shows the impact of the *selectivity* of range queries on the performance. Selectivity describes the expected portion of the keyspace, needed to be scanned during the query execution. In the uniform keys case, since the keys are generated from the $[0, 1)$ space, selectivity can simply be interpreted as the length of the interval used in the the range query predicate. Figure 2.5 shows the average response time of different systems and techniques, based on the query selectivity, where we have 1 million records. Going to the 10 million case, we observed the same relative performance between different systems, while the absolute response times had increased proportionally. Obviously there is no winner as the best technique for all the range selectivities. While the index-based techniques do much better than the no-IX technique for the short range queries, no-index technique would be the best choice to select for the longer range queries. Moreover, as the range query support in HBase and Cassandra is a new feature, and because of the design decisions made to let

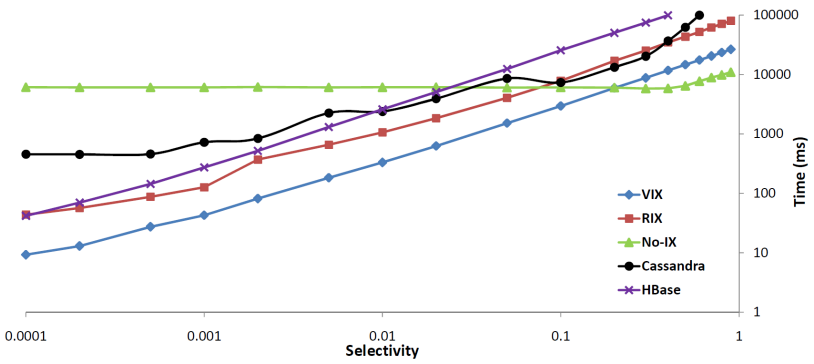

Figure 2.5: Selectivity test - Average time for a range query

these systems work efficiently on workloads that normally contain point and short range queries (as the major types of read workloads, they try to support) you can see that the current range query support works reasonably in these two systems for the shorter queries, but performs poorly for the medium and long queries.

2.7.4 Mixed query workloads

The above results indicate that there is no technique that outperforms others for all the range query selectivities. Thus the idea of using hybrid approaches comes to mind, where we create and maintain an index structure, for the short range queries, beside using the no-IX technique for long range queries.

Our next set of experiments considers the performance of different techniques, we developed on Voldemort, for the *mixed* range query workloads against datasets with the uniform distribution. As explained at the beginning of this section, we have created 3 mixed workloads, each consisting of a combination of range queries with different selectivities. Using these workloads, we actually attempted to simulate different types of real case scenarios. Mixed short range queries try to be served as an example of the typical workloads in interactive applications (OLTP), in which we have a large number of range queries, issued by different clients, while each query needs to access a small portion of data. In contrast, mixed long

range queries try to capture the workloads in analytical types of application (OLAP), where we need to retrieve large portions of data, based on our needs. Medium mixed workload is a combination of these two scenarios.

Figure 2.6 shows the response time of each technique, for these three scenarios and Table 2.3 shows the corresponding exact values as some of them are not clearly visible in the chart because of the scaling used for visualization.

The no-IX technique works almost the same for the three cases, since its execution cost is dominated by the direct access calls made to all the storage nodes. On the other hand, for the indexing techniques the response time changes significantly based on the number of tree nodes, needed to be scanned, and their average size. In fact, you can see that choosing the ultimate technique to use is highly dependent on the type of the range queries that you mostly need to process in your system. For OLAP applications, going after the no-IX technique seems to be a better choice, while for OLTP applications one may consider using the indexing techniques (especially VIX).

Hybrid approaches give us the possibility to get the best time, based on the expected query selectivity. As it can be seen in the results, Hybrid VIX outperforms others in almost all the cases. Hence, the runtime decision made based on the query selectivity may eliminate the need of the upfront decision on the choice between no-IX and VIX.

However, the flexibility of these hybrid approaches comes with the extra cost on update operations as shown in the next set of experiments.

2.7.5 Impact of hybrid schemes on other operations

Considering 10 million Key Value pairs, Figures 2.7 and 2.8 show the average response time for lookup and update operations in various systems. (Table 2.4 shows the exact values

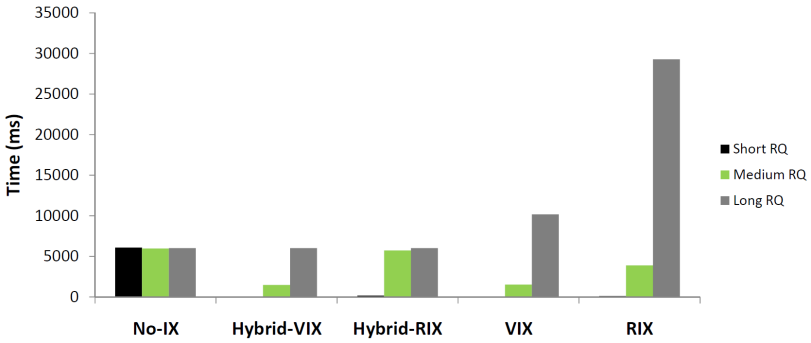

Figure 2.6: Mixed range queries workloads

Scheme	Short RQ	Medium RQ	Long RQ
No-IX	6082.59	5964.75	6021.39
Hybrid-VIX	44.84	1467.71	6009.41
Hybrid-RIX	156.27	5723.23	6014.23
VIX	37.54	1509.38	10170.54
RIX	124.65	3890.16	29286.03

Table 2.3: Mixed range queries response time for Figure 2.6 (in ms)

shown in these figures.)

First, comparing the performance of VIX to the original Voldemort (with the Berkeley DB storage engine) and RIX shows that VIX has a significant overhead for both operations. As discussed earlier, VIX has two sources of overhead: (1) index tree traversal for lookup and update; (2) reading/writing a larger Key Value pair that contains multiple data objects.

Next, notice the overhead of no-IX and hybrid approaches. They use MySQL (instead of BDB) for the underlying data store of Voldemort. In fact, the performance of no-IX is equivalent to the performance of the original Voldemort with MySQL.

Figure 2.9 shows the average time for inserting one Key Value pair, during the loading phase, among various systems. Notice that, unlike lookup and update, an insert operation increases the number of Key Value pairs, which occasionally involves additional costs such as index node split. The characteristics of such additional cost depend on a specific architecture. In

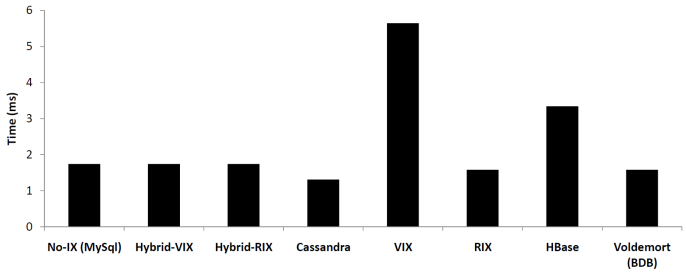

Figure 2.7: Lookup time for different systems and schemes

order to compare amortized insertion cost among different schemes, we measure the total time of sequential insertion of records into an empty data store. This time should not be confused with the expected response time to insert one record into an already populated data store with 10 million records.

Considering hybrid approaches, for all three operations (lookup, update and insert), you can see that they have further overheads in addition to the MySQL overhead. Especially, Hybrid-VIX needs to replicate the whole data object in order to not only store it in a data object store (for no-IX access) but also embed it in an index store (for VIX access). As a result, insert and update operations become very expensive. On the other hand, lookup response time is better than the case of VIX, since it can directly access data object in the store without index tree traversal. Thus, Hybrid-VIX is considered to be a highly read-optimized scheme.

The results for 1 million records are similar: the response times for the lookups were decreased, for most of the techniques (but the relative order was unchanged). With a decreased data size, we got more benefits from the cached Key Value pairs in the main memory of the servers, which actually reduces the operation time.

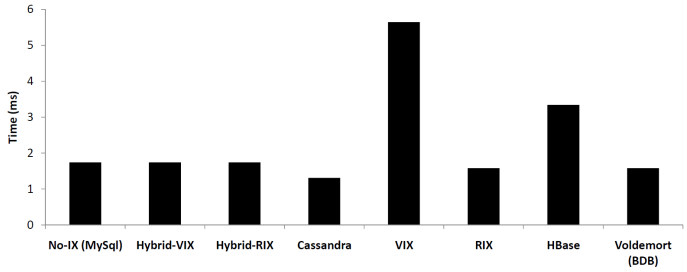

Figure 2.8: Update time for different systems and schemes

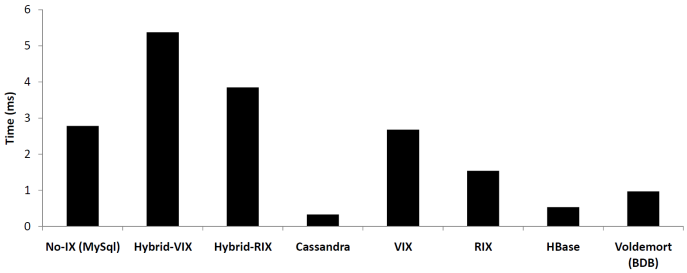

Figure 2.9: Insert time for different systems and schemes

System or Scheme	**Insert**	**Lookup**	**Update**
No-IX	2.78	1.74	5.02
Hybrid-VIX	5.38	1.74	11.83
Hybrid-RIX	3.85	1.74	5.02
Cassandra	0.33	1.31	1.08
VIX	2.68	5.64	9.09
RIX	1.54	1.58	4.12
HBase	0.53	3.34	0.02
Voldemort (BDB)	0.97	1.58	4.12

Table 2.4: Operations time for figures 2.7 to 2.9 (in ms)

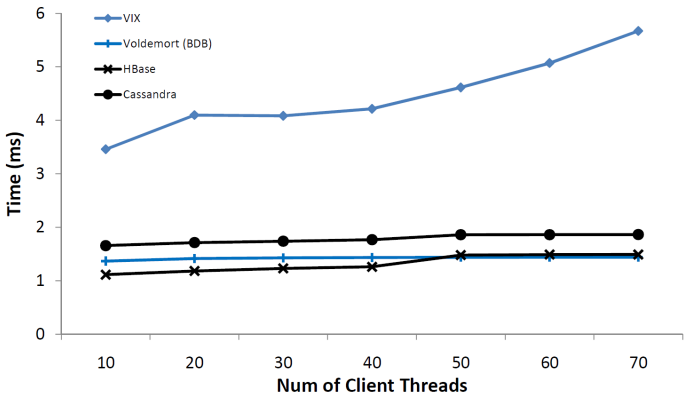

Figure 2.10: Lookup time in different systems

2.7.6 Lookup and update scalability

We have also compared the performance of lookups and updates between different systems. Here, we are interested in scalability in terms of the number of concurrent clients. Figures 2.10 and 2.11 show the response time and throughput of different systems for lookups, using uniform keys, as the number of clients increases, while Figures 2.12 and 2.13 show the same metrics for the updates. To control the number of clients, we used 1 to 8 client machines while each machine was running 10 threads of queries concurrently. We set the number of threads by making sure that the threads, on the client side, never turned to be a bottleneck for the query processing.

Recall that RIX is equivalent to the original Voldemort for lookup and update operations (hence it is omitted). VIX underperforms other systems, for both operations, as it always needs to get and put several index nodes into the Key Value store (the exact number of nodes is equal to the height of the tree). These nodes are much larger than single Key Value pairs, causing the extra overhead for VIX operations. Considering the update case, HBase's sequential write policy (appending commit logs to the tail of the transaction log file), which initially keeps the updates in memory, makes it outperform others.

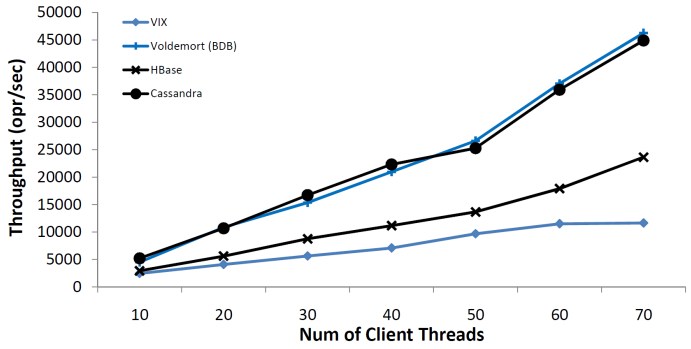

Figure 2.11: Lookup throughput in different systems

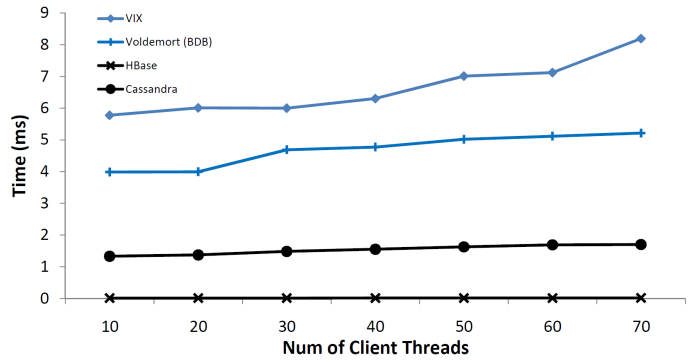

Figure 2.12: Update time in different systems

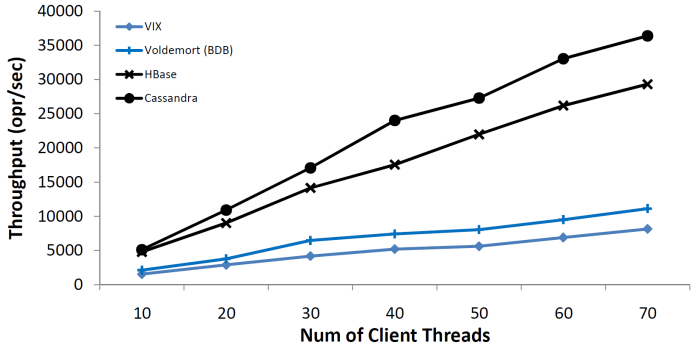

Figure 2.13: Update throughput in different systems

2.7.7 Range query scalability

We also investigated how scalable range queries are in terms of the number of concurrent clients. This scenario is important when the application needs to handle a large number of range queries from different clients. For this purpose, we selected three main techniques: VIX, RIX, and no-IX, along with the mixed range queries, and we measured the response time and throughput of the systems, using different numbers of clients (up to 8), each running different numbers of concurrent query threads (up to 10). Figures 2.14 to 2.19 show the results.

Ideally, if there is no significant contention among queries, the performance should look similar to the lookup operations (as shown in Figures 2.10 and 2.11), i.e., response time is almost constant and throughput linearly increases. However, in Figures 2.14 to 2.19, we observe the throughput saturates and response time increases. We further observe different degrees of saturation among three techniques for different range sizes.

Notice that, for the no-IX technique, the impact of client concurrency to the overall throughput is small regardless of the range length. With this technique, all the clients must access all the storage nodes at the same time. Thus, the effect of the client concurrency is bounded by the client concurrency of each storage node (i.e., MySQL). The difference between the performance of no-IX and VIX/RIX is significant for short range queries, where one client of VIX/RIX will not consume server resources as much as no-IX does.

While the performances of VIX and RIX schemes are close to each other for short range queries, you can see that they substantially differ with each other for medium and long range queries, with respect to the number of client threads. The average size of the BLink tree nodes, and the number of such nodes, needed to be retrieved concurrently, are the main reasons for this difference. While the average node size is larger in VIX compared to RIX (because of storing both keys and values together within VIX leaf nodes), we need to access

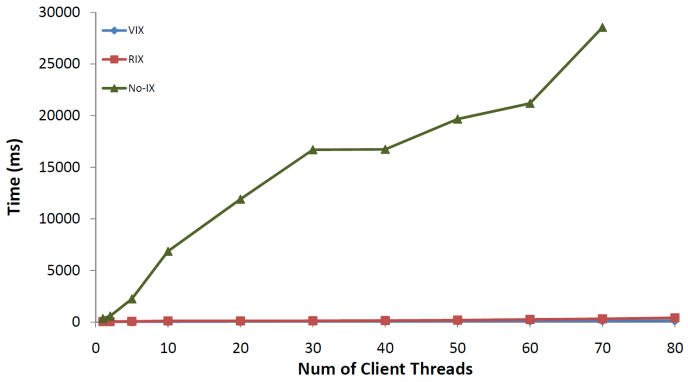

Figure 2.14: Multi client test - Short range queries response time

more leaf nodes in the medium and long range queries cases, compared to the short range queries case for both schemes. Thus as the number of client threads increases, RIX faces with less overhead and shows a better performance. But for smaller numbers of client threads, the extra lookups, needed to be done in the second store for RIX, make RIX's performance drop below VIX's.

For long range queries, none of the techniques is scalable in terms of the client concurrency, which is not surprising because each client needs to access a large fraction of the entire data, consuming a large amount of resources of each storage node. In this case, even indexing techniques would access almost all the storage nodes, as the length of range query covers a large portion of key space. No-ix is the most efficient technique, as we have seen in previous experiments, for this scenario.

Figure 2.20 shows the impact of changing the index node capacity (fanout) in VIX , for the medium range queries. As you can see, using larger fanout results in a gain for the throughput. Doing the same experiment, for the short and long range queries, we got similar relative results.

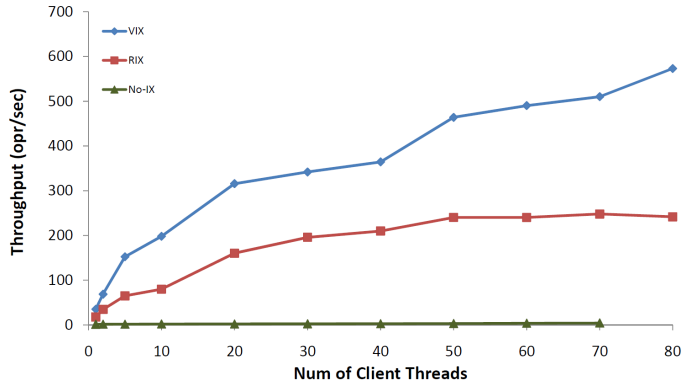

Figure 2.15: Multi client test - Short range queries throughput

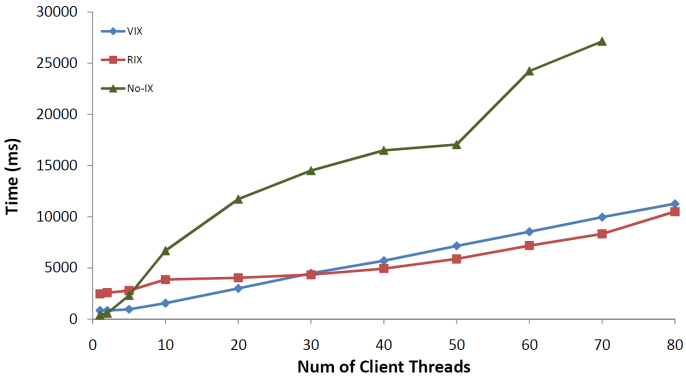

Figure 2.16: Multi client test - Medium range queries response time

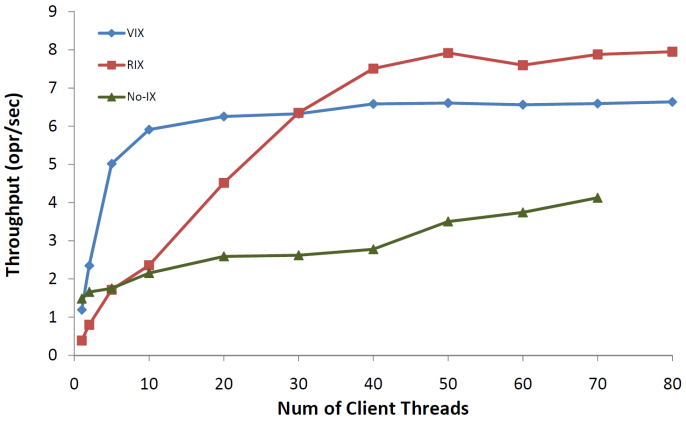

Figure 2.17: Multi client test - Medium range queries throughput

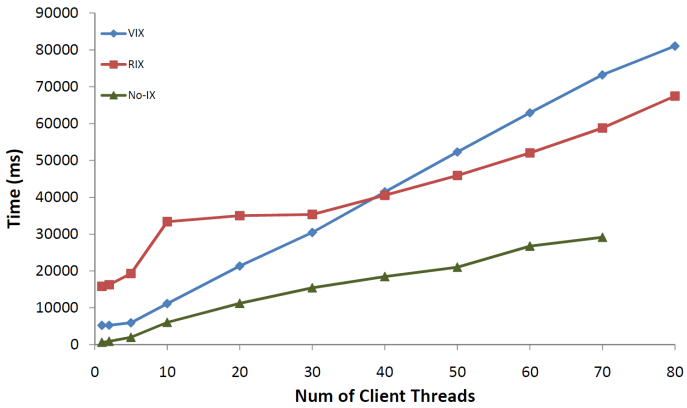

Figure 2.18: Multi client test - Long range queries response time

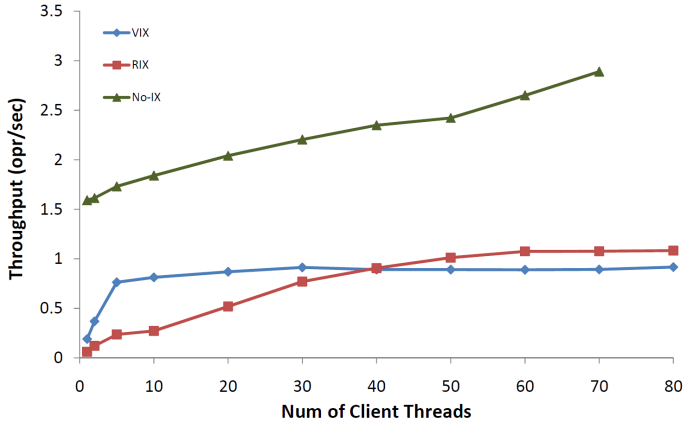

Figure 2.19: Multi client test - Long range queries throughput

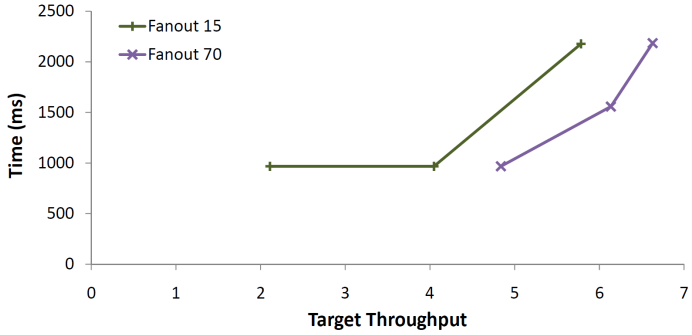

Figure 2.20: Multi client test - Impact of fanout and client threads - Medium range queries

2.7.8 Non-uniform data test

Given the fact that none of the techniques we developed on Voldemort to support range queries outperformed others in all the cases for uniform data; this set of experiments aimed at exploring the impact of key distribution on the performance of those schemes. For these tests we used 10 million keys, each associated with a value of size 1KB, while keys are generated from a Zipf distribution in the [0, 1) space, where most of the keys were close to 0 (smaller keys are more popular).

Impact of query length

Figure 2.21 shows the response time of different schemes for range queries on Voldemort, varying the range query length. You should note that unlike the uniform data, the length of a range query in this case can not be directly interpreted as the query selectivity (expected number of Key Value pairs, being retrieved). In fact, as the start point of a range query gets closer to 0, while its length is fixed, the expected number of keys, being accessed, would get larger. Based on the setting, used in Figure 2.21, VIX outperforms other two schemes. Moreover, in contrast to the uniform case, the no-IX method does not turn to be the technique of choice for the longer queries.

This result implies that a hybrid approach should consider the distribution of the keys carefully to estimate the cost of range queries more precisely. Similar to a query optimizer of RDBMSs, it is desirable to introduce histograms of the datasets in order to estimate the cardinality of a given interval in the key space.

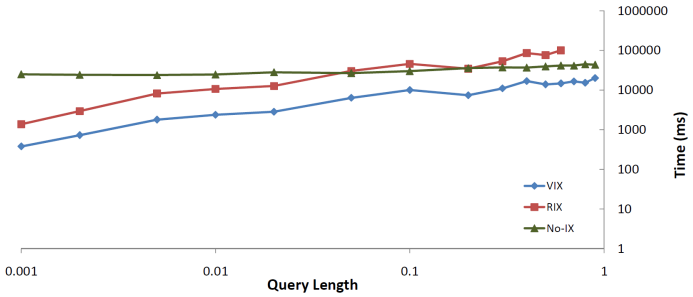

Figure 2.21: Non-uniform data test - Response time for different range query lengths

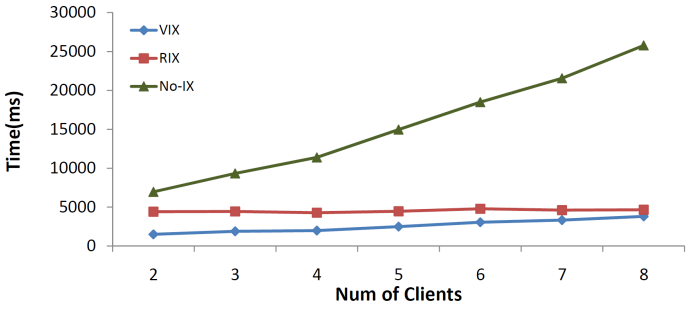

Figure 2.22: Non-uniform data test - Short range queries response time

Mixed query workloads

This set of experiments reports the results of the mixed query workloads, described earlier, for the non-uniform keys. Figures 2.22 to 2.27 show the response time and throughput of different schemes, based on the number of clients. For this set of experiments, we used 2 to 8 client machines, each running a stream of queries, against the Key Value store. An important point in this setting is the performance of VIX vs. no-IX technique: dealing with non-uniform keys, for the longer range queries, no-IX is no longer always outperforming VIX, as the difference between lengths of distinct range queries, does not show a proportional difference between their selectivities. These results again show the importance of taking data distribution into account, when considering hybrid approaches.

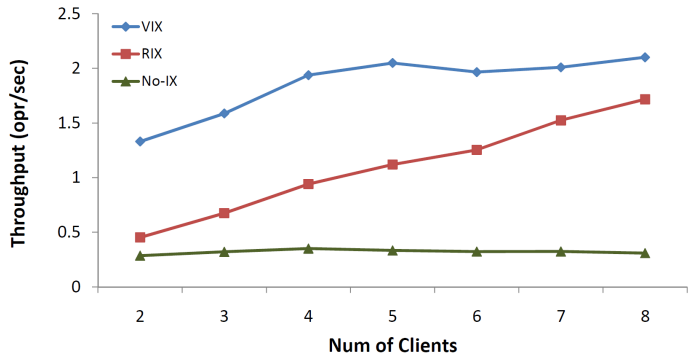

Figure 2.23: Non-uniform data test - Short range queries throughput

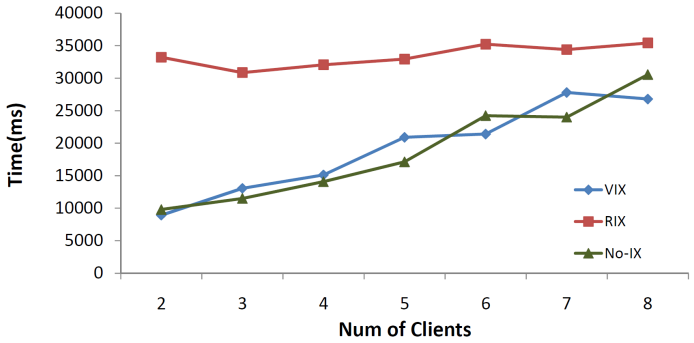

Figure 2.24: Non-uniform data test - Medium range queries response time

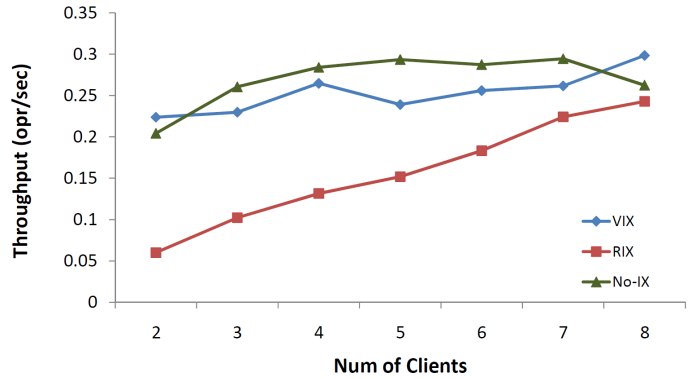

Figure 2.25: Non-uniform data test - Medium range queries throughput

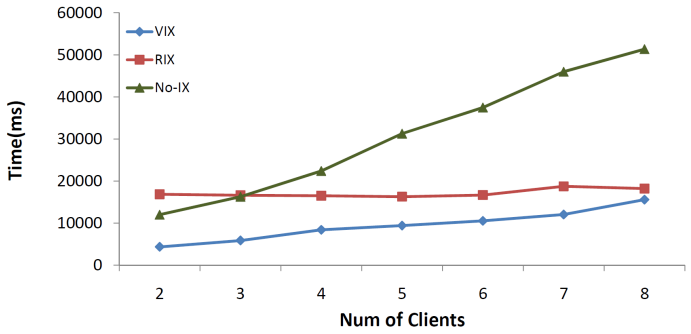

Figure 2.26: Non-uniform data test - Long range queries response time

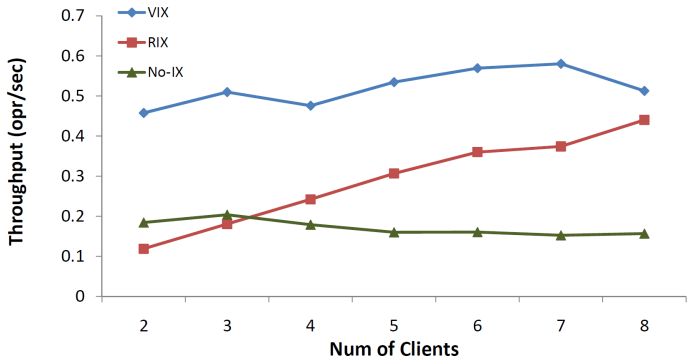

Figure 2.27: Non-uniform data test - Long range queries throughput

2.8 Discussions

In this section, we first summarize our observations based on the experiments and discuss the implications of these observations. We then suggest possible extensions on hash-partitioning Key Value stores to natively support range queries.

2.8.1 Summary of the performance results

The following summarizes our observations on the experiments:

- As reported in [61], HBase (BigTable) achieves excellent update performance. This is because of its sequential writes (using LSM-based storage, which is known for good performance under write heavy workloads). However, it underperforms RIX for lookup

and range queries, as HBase needs to do merged read to get the latest values.

- RIX achieves reasonable range query performance without sacrificing the lookup and value update performance of hash-partitioning Key Value stores. However, data insertion in RIX suffers from the additional cost of reference insertions into the index (two structures need to be modified per update: the index and the actual store).

- Although VIX outperforms others in terms of short to medium range queries, its overhead hinders scalability of lookup and update.

- The no-IX method incurs significant cost for the client to access all the storage nodes, costs that can only be amortized for a long range query. In addition, the impact of concurrent clients on the overall throughput is limited by the efficiency of each storage node to handle concurrent range queries. If the storage node (MySQL) could benefit from an optimized plan for concurrent range scans (such as shared scans among queries), no-IX would show more throughput increase from client concurrency. Nevertheless, no-IX would be a reasonable option if a workload consists of two subsets: a large number of lookup and update queries and a small number of long range queries.

Our observation that there is no clear winner implies the need for *physical design tuning*. Given various ways to store Key Value pairs with range accessibility, an application developer should choose one of those approaches to optimize the application's performance. In order to make such a tuning task efficient, a proper abstraction layer should be introduced. Although we implemented a BLink tree index on top of Key Value stores, such an implementation should be hidden from an application so that it can transparently access Key Value pairs with a common API. With such a logical abstraction, the task will be similar to physical database design tuning for relational databases (such as index selection). We expect that various technologies, originally developed for RDBMS, will be applicable to physical design tuning of Key Value stores.

We also notice that different approaches to range query support have different impacts on data consistency. Although our implementation of an index structure which supports range queries on hash-partitioning Key Value stores performs correctly for concurrent access, it does not support isolation between range queries and updates. In fact, none of the systems we explored in this chapter supports such an isolation (except for the fact that HBase's multi-versioning support is useful to implement some degree of isolation). A more detailed discussion of consistency and isolation is beyond the scope of this chapter.

2.8.2 Range index support by hash-partitioning Key Value stores

Our experiments indicate that employing a range index on a hash-partitioning Key Value store is a viable option as an alternative to range-partitioned Key Value stores. It is therefore desirable for hash-partitioning Key Value stores to natively support range indices. The following are possible extensions to Key Value stores for more efficient support of range queries.

Caching internal nodes: In the experiments, we employed a warm-up process before the actual measurement. This process not only warms up the servers but also warms up the clients' index node cache. In practice, however, the system should also perform well for *cold* clients. Thus, it is desirable to incorporate an internal node cache at the server side: the client should be able to access any of the storage nodes, which themselves should operate internal node traversal by leveraging the cache. The client will then receive the key of the leaf node, from which it will start an index scan.

Node split protocols: Recall that node splits must operate in an exclusive manner, which is why we introduced a lock flag within each index node. In this technique, we wanted to use the Key Value store without any change. However, if the storage nodes natively support

range index operations, they should be able to introduce much more efficient protocols to achieve exclusive node splits.

Notice also that the system can perform node splits in an asynchronous manner. Unlike a disk page in an RDBMS's storage, our index node does not have a hard limit on its capacity (fanout). Thus, an index node could be temporarily kept over-capacity and a split operation could be deferred. Managing node splits at the server side as a background task would improve the performance (especially availability) of insertion operations.

Asynchronous maintenance of RIX: RIX's overhead on the response time of insertion can be reduced if the Key Value store supports asynchronous maintenance of the reference-embedded indexes. This overhead appears because of the cost of waiting for the update of both the index and the store which contains the actual Key Value pairs. Having asynchronous view maintenance as a feasible option in the Key Value store, we could just update the Key Value pairs in the store and return to the user, while the index would get modified automatically by the Key Value store asynchronously. As a result, decreased response times upon insertion appear to be achievable at the cost of reduced consistency between the index and the store. Thus, asynchronous view maintenance [32] is a desirable feature for Key Value stores even for simple range queries.

Data access operations to VIX: VIX involves communication overhead between clients and a Key Value store due to large leaf index nodes. This could be reduced by pushing data manipulation (i.e., index node manipulation) to the Key Value storage nodes. For instance, Cassandra supports selective access to a column in a single row. We could introduce such an API to put/get a data object in a VIX leaf node.

Server-side scan: The no-IX technique, once chosen, should be integrated into Key Value stores in more efficient ways. In our experiments, the Hybrid-VIX approach needs to duplicate a data object in order to support both no-IX and VIX access patterns. A Key Value store with native range index support should not need such duplication, as it can scan the content of VIX leaf nodes at each storage node to support the no-IX pattern.

2.9 Conclusion and Future Work

In this chapter, we considered OLTP-style workloads and measured the performance of operations in this class of workloads - with a focus on range queries - in Key Value stores that employ both hash-partitioning (e.g., Voldemort) and range-partitioning (e.g., HBase, Cassandra). We described a micro-benchmark that used different types of data and query workloads against the considered Key Value stores, and we proposed techniques to support range queries in Voldemort. Our results showed that there is no absolute winner for all different types of queries, so we discussed guidelines for choosing the proper system and setting based on the data and the workload that one needs to process. In this Chapter, we focused on range scans on the key attribute, but as the data model in systems such as HBase and Cassandra gives users the ability to define columns and select a subset of them to retrieve, an extension to the work would be considering range queries on non-key attributes or studying the performance for multi-attribute range queries. Considering other types of query workloads such as non-uniform range scans for HBase and Cassandra would be another possible extension. Moreover, it would be interesting to apply the techniques proposed for range queries in Voldemort to other types of hash-partitioning systems, and to consider other systems such as Cassandra that support both hash-partitioning and range-partitioning and analyze their performance.

Chapter 3

Performance Evaluation of Big Data Management System Functionality

3.1 Overview

As described briefly in Chapter 1, the new generation of Big Data management systems (BDMS) differ quite a bit from one another in terms of their architectures and the features that they support. As a result, there is an evident need to benchmark them to facilitate comparisons and to reveal practical guidelines for making BDMS design decisions. In this chapter, we report on a preliminary evaluation of four representative systems: MongoDB, Hive, AsterixDB, and a commercial parallel shared-nothing relational database system. In terms of features, all offer to store and manage large volumes of data, and all provide some degree of query processing capabilities on top of such data. Our evaluation is based on a new micro-benchmark that utilizes a synthetic application that has a social network flavor. We analyze these preliminary performance results and then close with a follow-up discussion on lessons learned from this effort.

3.2 Motivation

The whole IT world is excited about the Big Data "buzz", and it is hard to avoid it. Immense volumes of data are generated continuously in different domains: The users of social networks keep publishing contents of various types on the web; online retailers are deeply interested in tracking users' activities and interests to make real-time suggestions; IoT-connected devices constantly produce and exchange data collected through their sensors. New platforms, with diverse sets of features, are emerging to fulfill the demand to collect, store, process and manage the resulting Big Data. Today's systems can be largely categorized into two groups: interactive request-serving systems (NoSQL), mainly serving OLTP types of workloads with simple operations, and Big Data analytics systems, which process scan-oriented OLAP types of workloads. Developers of new Big Data systems have made various decisions at different levels while designing and building them. These decisions affect the performance and the scope of the applications that each of the systems is appropriate for. This variety makes it difficult for end users to pick the most appropriate system for their specific use cases. In this situation, benchmarking seems to be an appropriate means to gain more insight. It can help end users to achieve a better understanding of the systems. It can also assist them in obtaining a set of guidelines and rules to make the correct decision in picking a system for their applications.

While well-established, comprehensive benchmarks exist to evaluate and compare traditional database systems, benchmarking efforts in the Big Data community have not yet achieved such a milestone. Some of the performance reports and white papers only show "hand-picked" results for scenarios where a system is behaving just as desired. In addition, it seems there are still ongoing debates about what the correct set of performance metrics is for a big data system [46]. These issues make the evaluation and comparison of these systems complicated.

There are a few Big Data benchmarking exercises that have gotten serious attention from the community. These efforts have each mostly focused on a well-defined, but narrow, domain of systems or use cases. Examples are YCSB [61] for OLTP and the work of Pavlo et al., in [86] for OLAP use-cases. Additionally, the types of operations that these efforts have included do not necessarily cover the full functionality set that a complete Big Data system should offer.

In this Chapter, our aim is to contribute to the Big Data performance area from a slightly different angle. We aim at studying and comparing Big Data systems based on their available features. We offer a new domain-centric micro-benchmark, called BigFUN (for Big Data FUNctionality), that utilizes a simple synthetic application with a social network flavor to study and evaluate systems with respect to the set of features they support along with the level of performance that they offer for those features. The BigFUN micro-benchmark focuses on the data types, data models, and operations that we believe a complete, mature BDMS should be expected to support. We report and discuss initial results measured on four Big Data systems: MongoDB [82], Apache Hive [3], a commercial parallel database system, and AsterixDB [36]. Our goal here is not to determine whether one Big Data management system (BDMS) is superior to the others. Rather, we are interested in exploring the tradeoffs between the performance of a system for different operations versus the richness of the set of features it provides.

We should emphasize that BigFUN is a first step towards the goal of studying general-purpose Big Data systems feature-wise. While we did our best in designing our micro-benchmark and using it to evaluate a set of representative platforms, we do not claim that this work is comprehensive. We hope our work can show the merit of the direction it has taken in Big Data benchmarking, and we expect future work to expand on this effort.

3.3 Related Work

The history of benchmarking data management technologies goes back to the 1980s, when first generations of relational DBMSs appeared in both academia and industry. The Wisonsin benchmark [62], which was a single-user micro-benchmark for relational operations, and the Debit-Credit benchmark [96], which was a multi-user benchmark modeling a transaction processing workload for a banking application, are among the very first works in this area. Because of the major influences that these efforts each had in terms of driving progress on data management systems, the Transaction Processing Performance Council (TPC) [24] came into existence, and it developed a series of benchmarks to drive systematic benchmarking. The TPC-H and TPC-C benchmarks are two instances of widely adopted benchmarks that came out of the TPC's efforts. In parallel, a number of benchmarks emerged from the research community. Examples include: OO1 [58] (for object operations), OO7 [53] (for object-oriented DBMSs), BUCKY [54] (for object-relational DBMSs), XMark [93], and EXRT [55] (for XML-related DBMS technologies). All these efforts were micro-benchmarks, each inspired by some popular, practical application.

As the Big Data buzz has started to attract attention from the data management community, work on Big Data benchmarking has also begun to appear. This work can roughly be divided into two main categories. The first group addresses the problem of evaluating Big Data serving technologies, from a broad perspective. An introductory article [46] describes the community-wide efforts for defining the requirements for a Big Data benchmark. BigBench [69] describes an end-to-end Big Data benchmark proposal which models a retailer selling products both in physical and online stores. The data model, data generator and workload specifications for BigBench are covered in [69], and its feasibility was validated on the Teradata Aster DBMS (TAD). A recent overview paper [52] discusses a series of potential pitfalls that may arise in the Big Data benchmarking route, along with a number of possibilities and unmet needs for the efforts in this area. The website for the Workshops on

Big Data Benchmarking (WBDB) [29] is a useful resource that lists and summarizes some of the major works and discussions on this front.

The second group of work in the context of benchmarking Big Data systems are those that propose a new Big Data benchmark and present the results obtained by running it. For Big Data analytics systems, [86] was the very first work that compared Hadoop [2] against Vertica and a row-organized parallel RDBMS (the same one examined here). It used a workload consisting of different selection tasks, aggregations, and a two-way join. On the NoSQL front, as discussed in Chapter 2, YCSB [61] (from Yahoo!) presented a multi-tier benchmark (using an open-system model) that uses mixed workloads of short read and write requests against a number of Key Value stores and sharded MySQL. Other works, such as [85] and [89], have extended this effort further. A recent work [66] looked at both classes of OLAP and OLTP workloads and used existing benchmarks (YCSB for NoSQL systems and TPC-H for DSS) to compare the performance of SQL Server PDW against Hive and sharded SQL Server to MongoDB. Other examples of Big Data micro-benchmarks include PigMix [18], GridMix [8], and most recently BG [45] and LinkBench [43] (from Facebook).

Recently, there have been efforts such as BigDataBench [101] to address issues in Big Data benchmarking that arise because of the complexity and variety of Big Data workloads and rapid changes in their serving systems. Hence, while the Big Data community has already identified the merit and major obstacles and challenges in evaluating Big Data systems, there is still a long way and potential opportunities to go to create practical benchmarks that can be widely adopted and used by users and the industry.

3.4 Systems Overview

In this section, we provide a brief overview of the systems that we are going to use in our evaluation later in the chapter. We picked these systems because of the fact that most of them have been used extensively by various customers and users for a range of different use cases in the Big Data world. Moreover, these systems cover a rich set of features that makes them reasonable candidates to conform to the goals of our performance study.

3.4.1 System-X

System-X is a commercial, parallel, shared-nothing, relational database system (unnamed for licensing reasons). It defines its schemas using the relational data model and uses SQL as the query language to describe requests. System-X partitions the data horizontally among cluster nodes. The data is stored in tables, and System-X manages its storage using efficient native RDBMS storage technology. There is also support for different types of indices and integrity constraints in the system. A client can submit a query to the system through any of its supported APIs, such as standard JDBC drivers, provided by the vendor. Having a mature cost-based query optimizer (which can be equipped with statistics about the data), an input SQL query gets converted into an optimized query plan, whose execution is done in parallel by the nodes in the cluster. System-X is included here as a representative of the older traditional way of managing Big Data.

3.4.2 Apache Hive

Hive [3] is the data warehouse on top of Hadoop [2], which provides a SQL-like interface for data processing. Its query language, called HiveQL, is a subset of SQL. Tables in Hive are created on existing data files in HDFS. Hive supports various file formats and compression

methods for the data which have significant performance differences. Some of the more popular Hive formats are sequence and text files, RCFile, ORC [14], and Parquet [16]. A client can submit a query either through Hive's CLI or through available server interfaces (such as HiveServer2). A user's request gets compiled by the Hive compiler, and an optimized execution plan of map and reduce jobs, in the form of a DAG, is generated for it. These jobs are then executed by the Hadoop framework, and the results can be stored in HDFS or delivered back to the user. Hive is included as a representative of the class of batch-oriented Big Data analytics platforms.

3.4.3 MongoDB

MongoDB is a NoSQL document database that uses a binary serialization of JSON, called BSON [6], to model and store the data. It stores each document, which is basically an extensible set of Key Value pairs, in a collection. Collections in MongoDB are fully schema-less, and a collection may contain heterogeneous documents with totally different schemas. MongoDB supports automated sharding and load balancing to distribute the data across cluster nodes to achieve horizontal scaling. The main MongoDB processes in a cluster environment are *mongod* and *mongos*. Mongos is a routing service that knows about the location of the data items in a cluster. Mongod is the main process for a MongoDB server and it serves data-access requests and performs management operations. Clients directly connect to MongoDB processes to submit their requests. MongoDB supports various types of indices and operations. Aggregations can be done through the aggregation framework as well as MongoDB's mapReduce command. Queries are all single collection in nature, and there is no support for a join operation in MongoDB. Changing the way that data is modeled and stored, such as using embedded documents, or performing client-side joins, are among the suggested alternatives to reduce the need for join operations in MongoDB. MongoDB is included as a representative of the current class of NoSQL stores.

3.4.4 AsterixDB

AsterixDB [36] is a new open source Big Data management system (BDMS) with a rich set of features for storing, managing and analyzing semi-structured data. It has its own declarative query language (AQL) and data model (ADM). ADM can be considered as a "super-set" of JSON, as it has additional primitive data types and type constructors as compared to JSON. AsterixDB uses a hash-partitioned LSM-based storage layer as its native storage [37]. It also has support for external storage (currently HDFS). AsterixDB has a rich set of data types that includes spatial, and temporal data types. It provides users with different types of indexing structures such as B+ Trees, spatial indices (R-Tree), and text indices (inverted index). A built-in data feed service [72] for continuous data ingestion is another available feature in AsterixDB. As the execution engine, AsterixDB uses Hyracks [49], a data-parallel runtime platform for shared-nothing clusters. Clients can use the available HTTP API in AsterixDB to submit queries. To process a query, AsterixDB first compiles it into an algebraic form that is then optimized by a rule-based optimizer and ultimately turned into a corresponding Hyracks job. This job gets executed by Hyracks and the results are delivered back to the client using the results distribution framework in AsterixDB. AsterixDB is included here as a representative of the next generation of Big Data management system platforms.

3.5 Data and Workload Description

The data management community has characterized different aspects of Big Data using the three V's: *Volume*, *Variety*, and *Velocity*. A mature BDMS needs to deal with each of these factors efficiently. It needs to store and manage huge volumes of data, while data items could potentially come from multiple sources with different structures and schemas, and such a system is supposed to handle high rates of incoming new data items and updates.

In the process of designing the BigFUN micro-benchmark, we did our best to reflect these expectations in the data and the workload. For the data, we decided to follow the idea of the Wisconsin benchmark [62] and design a generic synthetic database populated with randomly generated records. Synthetic data can be scaled accurately in a straight-forward manner. In addition, its flexibility in picking attribute values makes it a good choice for controlling the distribution of values and the duplication rate in attributes during data generation and later in generating workloads. We tried to design the BigFUN schema and workload in such a way that they can be directly used to evaluate a wide range of functionality, features, and query types. BigFUN aims to cover a range of basic and rich data types, as well as simple and complex queries and operations, to study their level of support and performance in a given system. We implemented data and query generators that are able to model various types of workloads. In both the data and the query generators, we added knobs to carefully control the major aspects of each synthetic workload.

In this section, we first describe the database that forms the basis of the BigFUN micro-benchmark. We introduce the data types and the datasets and explain how they can be scaled. Next, the benchmark operations are described. We divide the operations into two major categories: "read-only" queries and "data modification" operations. For each operation we provide a concise description that captures its semantics. The actual implementation of an operation depends on the specific system being evaluated, the query language it uses, and the set of features and structures it supports.

3.5.1 Database

The BigFUN schema includes simple, complex, and nested data types along with unique and non-unique attributes that can serve different indexing and querying purposes.

From a data model perspective, we are interested in identifying the existing atomic data

types in a system and checking if there is support for richer data types such as temporal or spatial data types. In addition, we want to check if a system is flexible in terms of storing heterogeneous records in a dataset or if it requires all the records in a dataset to have the same schema. We want to check whether a system is able to store records with one or more level(s) of nesting, or if it only supports flat, normalized records. From an indexing perspective, we want to explore if there is support for secondary indices in a system, and if yes, what types of index structures exist. Examples can be B-Trees, spatial indices such as R-Tree, and text indices.

The data in the BigFUN micro-benchmark is stored in multiple datasets. The actual number depends on whether records can be stored with nesting or need to be normalized.

Data types

We store information drawn from two imaginary social networks in our database: *Gleambook* and *Chirp*. We use five user-defined data types in the schema, two of which are used as nested data types in others (see Figure 3.1). The data types are as follows:

- **GleambookUserType:** It captures the information about specific members of the Gleambook network. An example record of type GleambookUserType is shown in Data 3.1 (in ADM format). Along with basic information such as the unique id and name of the user, each record of this type contains an ordered list of *EmploymentType* data records that shows their employment history. The *end_date* attribute in the *EmploymentType* is optional, and it only exists for already terminated employments. A record of type *GleambookUserType* also contains an unordered list of *friend_ids* that contains the unique ids of other existing users who are connected to this user in the network. *user_since* is a temporal attribute that stores the time when the user joined the network.

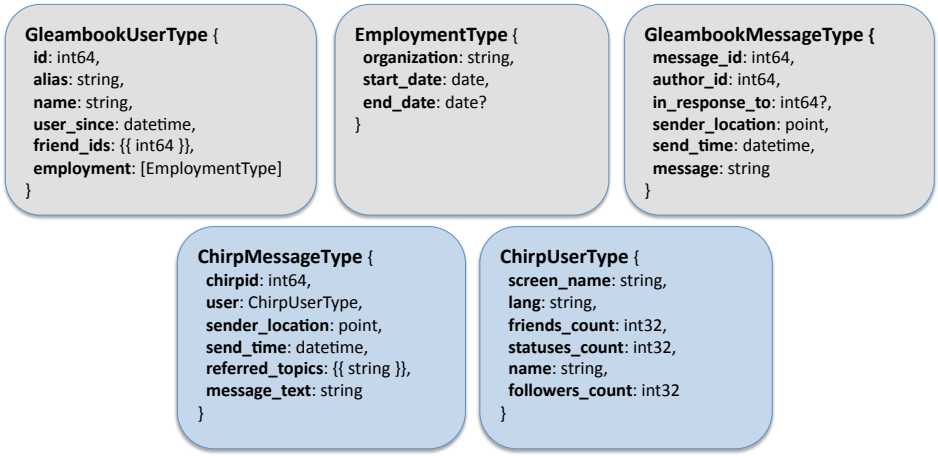

Figure 3.1: Nested BigFUN schema

- **GleambookMessageType:** It contains information about a message sent in the Gleambook network. An example of a record of type GleambookMessageType is shown in Data 3.2 (in ADM format). The message text itself is stored in the *message* attribute. *author_id* is a (foreign key) attribute that associates the message to its sender using the sender's unique id. *send_time* is a temporal attribute capturing the time that the message was sent by its user.

- **ChirpMessagesType:** It captures the information about a chirp message in the Chirp network. An example of a record of type ChirpMessagesType is shown in Data 3.3 (in ADM format). This data type includes *user* as a nested attribute of type *ChirpUserType* to store the information about the sender. It also contains an unordered list of keywords, *referred_topics*, which shows the hash tags associated with the chirp message. *send_time* is a temporal attribute that captures the time when this chirp message was sent. *sender_location* is a spatial attribute that saves the sending location of the chirp message.

The attributes in Figure 3.1 are a super-set of the attributes that a given instance of each data type may have. Some are denoted by "?" indicating that they are optional, such as *end_date* in the *EmploymentType*. In a system that supports semi-structured data, such

attributes can simply be left out of the records that they do not exist in. If a system only supports structured data where all attributes appear in each and every record, a "NULL" value is assigned to missing attributes.

```
{
        "id": 3001,
        "alias": "Kirk3001",
        "name": "Kirk Robinson",
        "user_since": datetime("2005-03-09T14:13:57"),
        "friend_ids": {{9520067, 3850565, 8879709, 4455450}},
        "employment":
        [
            {
                "organization_name": "X-technology",
                "start_date": date("2005-12-29"),
                "end_date": date("2006-02-25")
            },
            {
                "organization_name": "Quadlane",
                "start_date": date("2008-08-13")
            }
        ]
}
```

Data 3.1: GleambookUserType sample record (in ADM)

```
{
        "message_id": 52411101,
        "author_id": 3001,
        "in_response_to": 1774919,
        "sender_location": point("46.1488,68.1515"),
        "send_time": datetime("2006-02-14T17:37:11"),
        "message": "Making call from New York, love Verizon, my coverage is just good"
}
```

Data 3.2: GleambookMessageType sample record (in ADM)

```
{
        "chirpid": 146666644,
        "user": {
                "screen_name": "LawrenceCrom_796",
                "lang": "en",
                "friends_count": 9,
                "statuses_count": 313,
                "name": "Lawrence Halford",
                "followers_count": 154
        },
        "sender_location": point("26.6245,79.6869"),
        "send_time": datetime("2012-05-27T10:37:07"),
        "referred_topics": {{"Surface","screen","hardware","technology","device"}},
        "message_text": "Just got Surface tonight, like the screen, it is amazing"
}
```

Data 3.3: ChirpMessageType sample record (in ADM)

Datasets

The number of datasets in an implementation of the BigFUN micro-benchmark depends on the ability of a system to store records with nested and/or multi-valued (collection) attributes. In a system with a rich data model, we can store the BigFUN records in three datasets: *GleambookUsers*, *GleambookMessages*, and *ChirpMessages*, where each dataset stores records of one of the top-level data types described in Figure 3.1. If a system only supports flat records, then we need to normalize the schema. One way of doing that is storing the records in six datasets (see Figure 3.2). The *GleambookMessages* dataset remains intact, as its data type does not contain nested or multi-valued attributes. A single *GleambookUsers* record is stored in three separate datasets: one stores the basic information, while the other two store the employment details and friend ids. The unique id of the GleambookUser is the referential key in the *Employment* and *FriendIds* datasets. The *ChirpMessages* dataset becomes two datasets: one for the basic and sender information per chirp message, and one (*ReferredTopics*) for the hash tags for each chirp message that uses the unique *chirpid* to refer to the parent chirp message. The contents of the original schema's *user* attribute (which was nested) are flattened and stored along with the basic information in each chirp message.

Secondary Indices

Exploiting secondary indices is a way to improve the performance of some queries in a system. Different systems have different levels of support for various types of secondary indices. The BigFUN micro-benchmark aims at studying the performance of systems for various features, so it creates and uses secondary indices for its workload when it is justified and the system under test has support for them. B-Trees on numerical and temporal attributes, a spatial index on the spatial attribute capturing chirp message locations, and a keyword index on

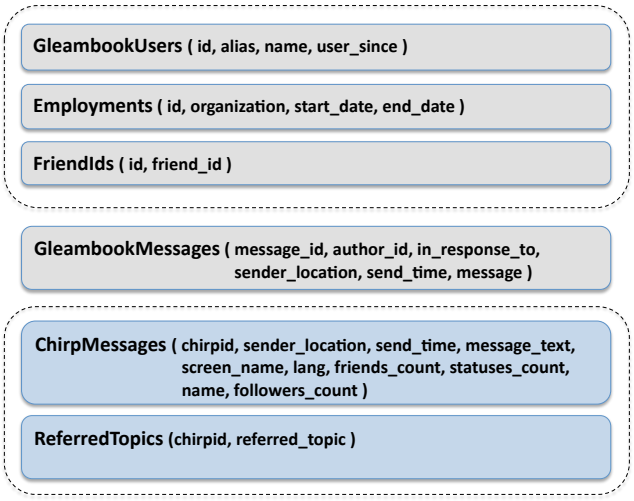

Figure 3.2: Normalized BigFUN schema

chirp messages for text search are the secondary index types included in BigFUN. More details are provided later in the experiments section.

Data Generation

We created a data generator for BigFUN that uses a simple, scalable approach to generate synthetic data in parallel across multiple partitions. It uses a scale factor that is interpreted as the total size of data (in bytes) across all datasets to characterize one specific load for the micro-benchmark. The data generator has a number of input parameters to control the total number of users that exist in each social network and the average number of messages per user. A set of adjustable policies are followed to generate the values for the attributes in each record. Here are the details about the generation process of BigFUN's most important attributes:

- **Identifiers:** Each partition receives minimum and maximum values for the identifiers it can generate, and it assigns a unique id to each record that it creates accordingly.
- **Temporal attributes:** The generator picks a random timestamp (a specific day for

date values, or a specific moment for datetime values), uniformly selected from the time interval between two pre-defined starting and ending moments.

- **Spatial attributes:** Location values are uniformly selected from a defined 2D region identified by minimum and maximum latitude and longitude limits. Currently, we fix the bounding box boundaries based on the territory of the United States on the map.

- **Employment:** Each GleambookUser has 1 to 4 employments, and in each the organization is selected randomly from a dictionary of company names. The value for the start_date is set similar to the temporal attributes, and there is a 40% chance that the user is still employed (so no end_date is needed). Otherwise, the value of the end_date is generated randomly between the selected start_date and the ending moment of the temporal attributes.

- **Message content:** The generator uses a set of message templates and a dictionary of keywords to generate synthetic, yet "meaningful", messages for GleambookMessages and ChirpMessages records. The messages mainly contain feelings about tech-related services or devices. There are multiple categories of keywords about devices, parts, vendors, and location names and each category has three separate lists of possible choices: rare, medium and frequent. These lists capture the popularity of keywords among the generated messages. Rare keywords appear in 5% of the messages. Those in the medium lists appear in 20% of the messages, and frequent keywords appear in 75% of the messages. The message text (and its referred_topics, if it belongs to a ChirpMessage) is set according to the values that the generator picks from the dictionary. In addition, there is a total of 5% of noise, in the form of misspellings or swappings of one or two characters in a keyword, that is injected into the text of the generated messages. This noise aims at reflecting the possibility of human errors when editing a message is used in the text similarity search queries.

3.5.2 Read-Only Queries

The first group of operations in BigFUN consists of read-only queries. Table 3.1 summarizes these queries. While designing this portion of the workload, we tried to meet these goals:

- **Clarity:** One should be able to associate each query with a real world operation or user request that could arise in the context of a valid Big Data application. For example, the "unique record retrieval" query (Q1) can be mapped to fetching the profile information for a specific user in a social network.
- **Simplicity:** Each query should be an independent operation that evaluates a well-defined and reasonably small set of features. For example, the "global aggregation" query (Q6) is focused on measuring the performance of aggregating the results of applying a built-in function on selected records.
- **Coherence:** A relationship exists between groups of queries in the benchmark, such that comparing results among them can reveal more about the performance of a system and the overhead of specific operations in it. For example, Q6, Q7, and Q8 all measure the performance of aggregation on a selected set of records. Q6 performs a simple aggregation, Q7 adds grouping to the aggregation pipeline, and Q8 adds ordering and limiting to produce ranked output.

We divide the read-only queries into groups based on the set of functionality and features that each explores and the dataset(s) that each accesses. Figure 3.3 summarizes the groups, and they are discussed in more detail below. Each operation is also shown for both the nested schema (Figure 3.1) in AQL and the normalized schema (Figure 3.2) in SQL (whenever available). The parameter values in the query statements are shown in a generic form (preceded by @ symbol) as their actual values are set by the query generator during the benchmark execution and according to the query version.

QId	Name	Description
Q1	Unique record retrieval	Retrieve an existing user using his or her user id.
Q2	Record id range scan	Retrieve the users whose user ids fall in a given range.
Q3	Temporal range scan	Retrieve the users who joined the network in a given time interval.
Q4	Existential quantification	Find basic and employment information about users who joined the network in a given time interval and are currently employed.
Q5	Universal quantification	Find basic and employment information about users who joined the network in a given time interval and are currently not employed.
Q6	Global aggregation	Find the average length of chirp messages sent within a given time interval.
Q7	Grouping & aggregation	For chirp messages sent within a given time interval find the average length per sender.
Q8	Top-K	Find the top ten users who sent the longest chirp messages (on average) in a given time interval.
Q9	Spatial selection	Find the sender names and texts for chirp messages sent from a given circular area.
Q10	Text containment search	Find the texts and send-times for the ten most recent chirp messages containing a given word.
Q11	Text similarity search	Find the texts and send-times for the ten most recent chirp messages that contain a word similar (based on edit distance) to a given word.
Q12	Select equi-join	Find the users' names and message texts for all messages sent in a given time interval by users who joined the network in a specified time interval.
Q13	Select left-outer equi-join	For users who joined the network in a given time interval, find their name and the set of messages that they sent in a specified time interval.
Q14	Select join with grouping & aggregation	For those users who joined the network in a given time interval, find their ids and the total number of messages each sent in a specified time interval
Q15	Select join with Top-K	For those users who joined the network in a given time interval, find their ids and the total number of messages each sent in a specified time interval; report only the top ten users based on the number of messages.
Q16	Spatial join	For each chirp message sent within a given time interval, find the ten nearest chirp messages.
U1	(Batch) Insert	Given the information for a set of new users, add the information to the database.
U2	(Batch) Delete	Given a set of existing user ids, remove the information for each user from the database.

Table 3.1: BigFUN operations description

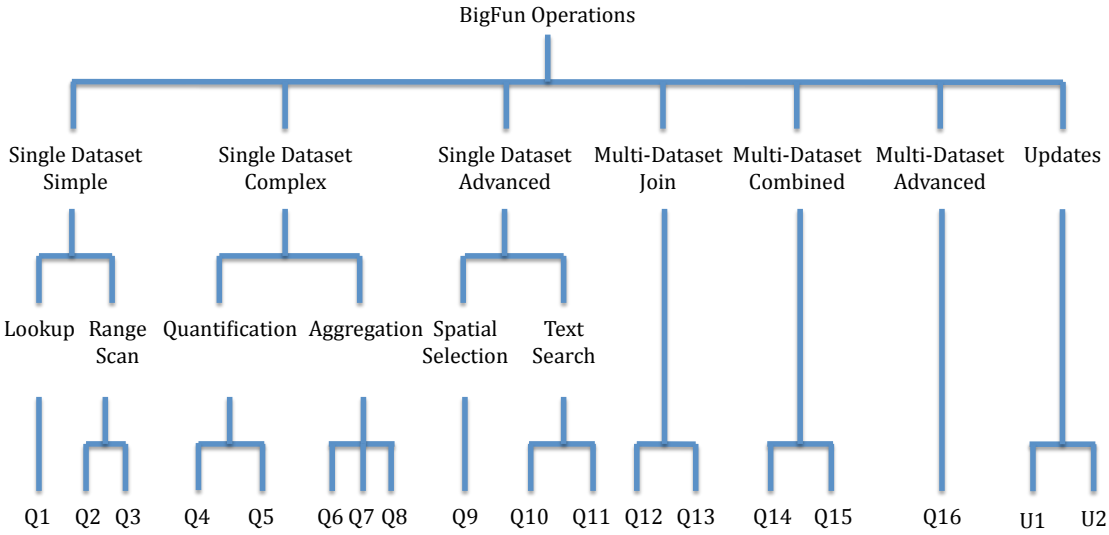

Figure 3.3: BigFUN operations

G1. Single Dataset - Simple

The first group of operations consists of three queries, Q1, Q2, and Q3. These queries focus on a basic, but frequently used, operation in a data store: record retrieval using identifying key(s), where all information about a set of records is fetched. All three queries retrieve records from the GleambookUsers dataset. Q1 and Q2 use the primary key for this purpose (a given key value identifies one unique existing record). Q3 specifies its selection criterion using a range of valid values for a temporal attribute that is not necessarily unique among the records. For Q2 and Q3, we consider multiple range sizes to test the impact of the number of qualifying records on performance. The performance of these queries can be improved if a system has support for indexing: primary index for Q1 and Q2, and a secondary index for Q3. Otherwise, scanning the whole dataset and checking each and every existing record is necessary.

Q1 - Unique record retrieval

This query uses the primary key as the selection criterion to retrieve all the information corresponding to a single existing record. As GleambookUsers is used as the target dataset

the retrieved record will have nested and collection attributes. (Check Listing 3.1).

```
for $u in dataset GleambookUsers
where $u.id = @INT64
return $u
```

Listing 3.1: Q1 (in AQL for the nested schema)

In the normalized schema case, the query needs to access multiple tables (GleambookUsers, Employments, and FrindIds) to fetch the required information and combine them together. We use "UNION ALL" (in SQL) to combine the retrieved information from three tables and list the retrieved GleambookUsers, Employments, and FriendIds tuples with an expanded schema. (Check Listing 3.2).

```
(select u.id as uc0, u.alias as uc1,
        u.name as uc2, u.user_since as uc3,
        null as ec4, null as ec5, null as ec6,
        null as fc7
from GleambookUsers u
where u.id = @INT64
)
UNION ALL
(select null as uc0, null as uc1, null as uc2,
        null as uc3, e.org as ec4, e.start_date as ec5,
        e.end_date as ec6, null as fc7
from Employments e
where  e.id = @INT64
)
UNION ALL
(select null as uc0, null as uc1, null as uc2,
        null as uc3, null as ec4, null as ec5,
        null as ec6, f.friend_id as fc7
from FriendIds f
where f.id = @INT64
)
```

Listing 3.2: Q1 (in SQL for the normalized schema)

Q2 - Record identifier range scan

This query retrieves a group of existing records from the GleambookUsers dataset based on a valid range of primary key values. (Check Listing 3.3). In the normalized schema case and similar to Q1, we need to access three tables. In order to combine the results, we order the retrieved tuples by their "id" value so that all the tuples corresponding to one user come after each other (so we can get as close as possible to the results in the nested schema case). (Check Listing 3.4).

```
for $u in dataset GleambookUsers
where $u.id >= @int64
and $u.id < @int64
return $u
```

Listing 3.3: Q2 (in AQL for the nested schema)

```
(select u.id as uc0, u.alias as uc1,
        u.name as uc2, u.user_since as uc3,
        null as ec4, null as ec5, null as ec6,
        null as fc7
from GleambookUsers u
WHERE
        u.id >= @int64 AND
        u.id < @int64
)
UNION ALL
(select e.id as uc0, null as uc1, null as uc2,
        null as uc3, e.org as ec4, e.start_date as ec5,
        e.end_date as ec6, null as fc7
from Employments e
WHERE
        e.id >= @int64 AND
        e.id < @int64
)
UNION ALL
(select f.id as uc0, null as uc1, null as uc2,
        null as uc3, null as ec4, null as ec5,
        null as ec6, f.fr_id as fc7
from FriendIds f
WHERE
        f.id >= @int64 AND
        f.id < @int64
)
ORDER BY uc0
```

Listing 3.4: Q2 (in SQL for the normalized schema)

Q3 - Temporal range scan

This query retrieves a group of records from the GleambookUsers dataset based on a valid range of values for a temporal, non-unique attribute. (Check Listing 3.5).

```
for $u in dataset GleambookUsers
where $u.user_since >= @datetime
and $u.user_since < @datetime
return $u
```

Listing 3.5: Q3 (in AQL for the nested schema)

In the normalized schema case, as the selection predicate is based on an attribute which only exists in the parent table (the *user_since* attribute in GleambookUsers) we need to join the parent and each child table to fetch corresponding Employments and FriendIds tuples. Similar to Q2, the retrieved tuples are ordered by the primary key (foreign key for child tables) so all the tuples for a user appear together. (Check Listing 3.6).

```
(select u.id as uc0, u.alias as uc1,
        u.name as uc2, u.user_since as uc3,
        null as ec4, null as ec5, null as ec6,
        null as fc7
from GleambookUsers u
WHERE
        u.user_since >= @datetime AND
        u.user_since < @datetime
)
UNION ALL
(select u2.id as uc0, null as uc1, null as uc2,
        null as uc3, e.org as ec4, e.start_date as ec5,
        e.end_date as ec6, null as fc7
from GleambookUsers u2, Employments e
WHERE   u2.id = e.id AND
        u2.user_since >= @datetime AND
        u2.user_since < @datetime
)
UNION ALL
(select u3.id as uc0, null as uc1, null as uc2,
        null as uc3, null as ec4, null as ec5,
        null as ec6, f.fr_id as fc7
from GleambookUsers u3, FriendIds f
WHERE   u3.id = f.id AND
        u3.user_since >= @datetime AND
        u3.user_since < @datetime
)
ORDER BY uc0
```

Listing 3.6: Q3 (in SQL for the normalized schema)

G2. Single Dataset - Complex

This group of operations include five queries, Q4 to Q8. They cover two core features that arise in both OLTP and OLAP types of workloads: Quantification and Aggregation. All these queries use a temporal filter to select a set of records from their target dataset prior to the main operation. We use the selectivity of this filter to control the number of qualifying records to create multiple versions of each query. The quantification queries (Q4, and Q5) access the GleambookUsers dataset and use the *employment* attribute in their predicates to find employed and unemployed users. The aggregation queries (Q6, Q7, and Q8) use the ChirpMessages dataset and report aggregate values for a selected set of messages. The operations in this group of queries (quantification, grouping, and aggregation) can be implemented and optimized in various ways. This group of queries aims at providing a basis to evaluate and compare systems from that perspective.

Q4 - Existential quantification

This query tries to find information about currently "employed" users who joined the Gleambook network in a given time interval. Lack of end_date for at least one of the employments of a user shows that he is currently employed and this query uses existential quantification to find users with such a status. (Check Listing 3.7.) In the normalized schema case, as the information about the employments of users is stored in a separate table, the query needs to access two tables and combines the retrieved information to evaluate and return the final results. (Check Listing 3.8).

```
for $u in dataset GleambookUsers
where
      (some $e in $u.employment
      satisfies is-null($e.end_date)) and
      $u.user_since >= @datetime
and
      $u.user_since < @datetime
return {
        "uname": $u.name,
        "emp": $u.employment
}
```

Listing 3.7: Q4 (in AQL for the nested schema)

```
SELECT u.name, em.id, em.organization,
       em.start_date, em.end_date
FROM GleambookUsers u, Employments em
WHERE   u.id = em.id AND
        u.user_since >= @datetime AND
        u.user_since < @datetime
AND EXISTS (
SELECT *
from Employments e
WHERE   e.id = u.id AND
        e.end_date is NULL
)
ORDER BY u.id
```

Listing 3.8: Q4 (in SQL for the normalized schema)

Q5 - Universal quantification

This query is similar to Q4 with the difference that it tries to find currently "unemployed" users from a set of users of the Gleambook network. All the employments of an unemployed user are already terminated i.e., they all have valid values for their *end_date* attributes. Therefore, the query uses a universal quantification to find if a user is unemployed or not. (Check Listings 3.9 and 3.10).

```
for $u in dataset GleambookUsers
where
      (every $e in $u.employment
      satisfies not(is-null($e.end_date))) and
      $u.user_since >= @datetime
and
      $u.user_since < @datetime
return {
        "uname": $u.name,
        "emp": $u.employment
}
```

Listing 3.9: Q5 (in AQL for the nested schema)

```
SELECT u.name, em.id, em.organization,
       em.START_DATE, em.END_DATE
FROM GleambookUsers u, Employments em
WHERE   u.id = em.id AND
        u.user_since >= @datetime AND
        u.user_since < @datetime
AND NOT EXISTS (
SELECT *
from Employments e
WHERE   e.id = u.id AND
        e.end_date is NULL
)
ORDER BY u.id
```

Listing 3.10: Q5 (in SQL for the normalized schema)

Q6 - Global aggregation

This is the basic query in the aggregation family that calculates the average length of the messages in given set of records picked from ChirpMessages. For both of the nested and normalized schema cases all the information for this query is stored in one dataset/table. (Check Listings 3.11 and 3.12).

```
avg(
 for $t in dataset ChirpMessages
 where  $t.send_time >= @datetime and
        $t.send_time < @datetime
 return string-length($t.message_text)
)
```

Listing 3.11: Q6 (in AQL for the nested schema)

```
SELECT avg( length(m.message_text) )
FROM ChirpMessages m
WHERE   m.send_time >= @datetime AND
        m.send_time < @datetime
```

Listing 3.12: Q6 (in SQL for the normalized schema)

Q7 - Grouping and aggregation

This is the second query in the aggregation family which adds grouping to Q6. The

aggregated value is reported per group of Chirp messages, grouped based on the name of the users who sent them. (Check Listings 3.13 and 3.14).

```
for $t in dataset ChirpMessages
where   $t.send_time >= @datetime and
        $t.send_time < @datetime
let $m := string-length( $t.message_text )
group by $uname := $t.user.screen_name with $m
return {
    "user": $uname,
    "avg": avg($m)
}
```

Listing 3.13: Q7 (in AQL for the nested schema)

```
SELECT m.screen_name, avg( length(m.message_text) ) AS avg
FROM ChirpMessages m
WHERE   m.send_time >= @datetime AND
        m.send_time < @datetime
GROUP BY m.screen_name
```

Listing 3.14: Q7 (in SQL for the normalized schema)

Q8 - Top-K

This is the last query in the aggregation family. It adds ranking to Q7 and returns the top users with the longest messages, sent in a given time interval, in the Chirp network. (Check Listings 3.15 and 3.16).

```
for $t in dataset ChirpMessages
where   $t.send_time >= @datetime and
        $t.send_time < @datetime
let $m := string-length( $t.message_text )
group by $uname := $t.user.screen_name with $m
let $a := avg($m)
order by $a desc
limit 10
return {
    "user": $uname,
    "avg": $a
}
```

Listing 3.15: Q8 (in AQL for the nested schema)

```
SELECT m.screen_name, avg( length(m.message_text) ) AS avg
FROM ChirpMessages m
WHERE   m.send_time >= @datetime AND
        m.send_time < @datetime
GROUP BY m.screen_name
ORDER BY avg( length(m.message_text) )  DESC
FETCH FIRST 10 ROWS ONLY
```

Listing 3.16: Q8 (in SQL for the normalized schema)

G3. Single Dataset - Advanced

This group consists of three queries, Q9, Q10, and Q11, that evaluate a system's ability in selecting records from ChirpMessages using spatial or textual similarity predicates. Q9 performs spatial selection to retrieve all the messages sent from a given region. Q10 and Q11 retrieve a set of chirp messages based on their messages' contents. The predicate in Q10 uses an exact string matching criterion, while the predicate in Q11 uses edit distance as the similarity metric. Like the previous groups of queries, we create different versions for each query by adjusting the expected selectivity of the predicates. For Q9, the boundaries of the region are varied, while in Q10 and Q11 the expected occurrence frequency of the search keyword is varied by picking keywords with different popularities (the query keywords are selected from the dictionary that the data generator uses). These queries can be processed if the system has support for spatial and textual data types and proper comparators for them. If a system supports relevant indexing techniques, it may achieve a better performance by using those indices.

Q9 - Spatial selection

This query uses a predicate on the spatial attribute (*sender_location*) in the ChirpMessages dataset to select a set of messages sent from a given circular area. The area is defined using a valid point as the center of the circle and its radius. (Check Listing 3.17).

```
let $p := create-point(@double, @double)
let $r := create-circle($p, @double)

for $t in dataset ChirpMessages
where spatial-intersect($t.sender_location, $r)
return {
        "name": $t.user.screen_name,
        "message": $t.message_text
}
```

Listing 3.17: Q9 (in AQL for the nested schema)

Q10 - Text containment search

This query uses a query keyword picked from the dictionary used during data generation to select a set of messages from the ChirpMessages dataset. The sending time and text of the most recent messages which contain an "exact" occurrence of the given query keyword are returned as the results of the query. (Check Listing 3.18).

```
for $t in dataset ChirpMessages
where contains($t.message_text, @string)
order by $t.send_time desc
limit 10
return {
        "time": $t.send_time,
        "message": $t.message_text
}
```

Listing 3.18: Q10 (in AQL for the nested schema)

Q11 - Text similarity search

This query is similar to Q10 with the difference that it uses "fuzzy" matching for text similarity search (instead of "exact" word containment). The similarity metric of choice is edit distance with a fixed threshold to control the maximum number of one missing/modified characters between the query keyword and the words used in the messages. (Check Listing 3.19).

```
for $t in dataset ChirpMessages
where
edit-distance-contains($t.message_text, @string, 1)[0]
order by $t.send_time desc
limit 10
return {
        "time": $t.send_time,
        "message": $t.message_text
}
```

Listing 3.19: Q11 (in AQL for the nested schema)

G4. Multi-Dataset - Join

Join is a fundamental operation in databases that arises in a wide range of use cases. Its frequency of use is somewhat lessened in a data model with nesting, but it is still an important operation. In the Big Data world, where a system needs to deal with a huge amount of

(distributed) data, performing join in a scalable, efficient manner can be a challenge. Q12 and Q13 are join queries that join the users from GleambookUsers with their messages in the GleambookMessages dataset. Q12 is a regular equi-join while Q13 is a left outer join. Multiple versions of each query are generated by modifying the expected selectivity of the temporal filter on the GleambookUsers side.

Q12 - Select equi-join

This query is a select-equi join which returns pairs of user names and messages which capture information about all the messages, sent in a given time interval, by a group of users in the Gleambook network. For this purpose this query joins these two datasets using the primary key from the GleambookUsers side which is stored as the foreign key (i.e., the *author_id* attribute) in the GleambookMessages records. (Check Listings 3.20 and 3.21).

```
for $m in dataset GleambookMessages
for $u in dataset GleambookUsers
where   $m.author_id = $u.id and
        $u.user_since >= @datetime and
        $u.user_since < @datetime and
        $m.send_time >= @datetime and
        $m.send_time < @datetime
return {
        "uname": $u.name,
        "message": $m.message
}
```

Listing 3.20: Q12 (in AQL for the nested schema)

```
SELECT u.name, m.message
FROM GleambookUsers u, GleambookMessages m
WHERE   m.author_id = u.id AND
        u.user_since >= @datetime AND
        u.user_since < @datetime AND
        m.send_time >= @datetime AND
        m.send_time < @datetime
```

Listing 3.21: Q12 (in SQL for the normalized schema)

Q13 - Select left-outer equi-join

This query does a left-outer join and uses similar filtering predicates as Q12 on the GleambookUsers and GleambookMessages sides to find all the messages that each user, from a selected set of users, has sent. The set of messages of a user could be empty if the user did

not send any message within the considered time interval. (Check Listing 3.22.) For the normalized schema case, we order the retrieved messages based on the name of their senders so that all the messages that a given user had sent appear together. This way we try to get as close as possible to the results that the query on the nested schema would return. (Check Listing 3.23.)

```
for $u in dataset GleambookUsers
where   $u.user_since >= @datetime and
        $u.user_since < @datetime
return {
        "uname": $u.name,
        "messages":
                for $m in dataset GleambookMessages
                where   $m.author_id = $u.id and
                        $m.send_time >= @datetime and
                        $m.send_time < @datetime
                        return $m.message
}
```

Listing 3.22: Q13 (in AQL for the nested schema)

```
SELECT u.name, m.message
FROM GleambookUsers u LEFT OUTER JOIN GleambookMessages m
ON m.author_id = u.id
WHERE   u.user_since >= @datetime AND
        u.user_since < @datetime AND
        m.send_time >= @datetime AND
        m.send_time < @datetime
ORDER BY u.name
```

Listing 3.23: Q13 (in SQL for the normalized schema)

G5. Multi-Dataset - Combined

This group of queries (Q14 and Q15) combines joins with aggregation. They use the GleambookUsers and GleambookMessages datasets and report grouped aggregation and ranked results. The main goal of these queries is to measure the performance of a system for two of the core operations in a data store (join and aggregation) once they are combined together and compare it to the results of the aggregation-only and join-only queries. Such a combination could occur in an application where one wants to calculate aggregated values or to rank records coming from different data sources. Similar to the join queries, different versions of these queries are created by changing the expected selectivity of the temporal filter on the users side.

Q14 - Select join with grouping and aggregation

This query combines join and aggregation operations by joining GleambookUsers and GleambookMessages datasets to find the number of messages that a selected set of users sent during a given time interval. For this purpose the resulting tuples from the join are grouped according to the id of the senders and the aggregated count is reported for each group. (Check Listings 3.24 and 3.25.)

```
for $m in dataset FacebookMessages
for $u in dataset GleambookUsers
where   $m.author_id = $u.id and
        $u.user_since >= @datetime and
        $u.user_since < @datetime and
        $m.send_time >= @datetime and
        $m.send_time < @datetime
group by $uid := $u.id with $u
let $c := count($u)
return {
        "uid":  $uid,
        "count": $c
}
```

Listing 3.24: Q14 (in AQL for the nested schema)

```
SELECT u.id, count(*) AS count
FROM GleambookUsers u, GleambookMessages m
WHERE   m.author_id = u.id AND
        u.user_since >= @datetime
        AND u.user_since < @datetime
        AND m.send_time >= @datetime
        AND m.send_time < @datetime
GROUP BY u.id
```

Listing 3.25: Q14 (in SQL for the normalized schema)

Q15 - Select join with top-K

This query is similar to Q14 with the addition of ranking and it returns top groups i.e., the users who have sent the most number of messages in a given time interval. (Check Listings 3.26 and 3.27.)

G6. Multi-dataset - Advanced

The last group of queries contains a select-join query (Q16) whose join predicate is on a spatial attribute and it tries to find recent messages that are sent from a given neighborhood

```
for $m in dataset GleambookMessages
for $u in dataset GleambookUsers
where   $m.author_id = $u.id and
        $u.user_since >= @datetime and
        $u.user_since < @datetime and
        $m.send_time >= @datetime and
        $m.send_time < @datetime
group by $uid := $u.id with $u
let $c := count($u)
order by $c desc
limit 10
return {
        "uid":  $uid,
        "count": $c
}
```

Listing 3.26: Q15 (in AQL for the nested schema)

```
SELECT u.id, count(*) AS count
FROM GleambookUsers u, GleambookMessages m
WHERE   m.author_id = u.id AND
        u.user_since >= @datetime
        AND u.user_since < @datetime
        AND m.send_time >= @datetime
        AND m.send_time < @datetime
GROUP BY u.id
ORDER BY count(*) DESC
FETCH FIRST 10 ROWS ONLY
```

Listing 3.27: Q15 (in SQL for the normalized schema)

around a set of Chirp messages. Similar to Q9, the boundaries of the the neighborhood are modified to create different versions of the query. Spatial indexing techniques can improve a system's performance for this query.

Q16 - Spatial join

This query is a spatial join which selects a set of messages from the ChirpMessages dataset, using the *send_time* attribute and performs a spatial (self-)join to find "near-by" messages for each of the selected chirp messages. A near-by message is one that was sent from a specified neighborhood around the sending location of the original chirp message. The query returns the top 10 near-by messages (ranked based on the sending time) per selected chirp message. (Check Listing 3.28.)

```
for $t1 in dataset ChirpMessages
where $t1.send_time >= @datetime and
$t1.send_time < @datetime
let $n := create-circle($t1.sender_location, @double)
return {
"chirp": $t1.chirpid,
"nearby-chirps":
   for $t2 in dataset ChirpMessages
   where spatial-intersect($t2.sender_location,$n) and
        $t1.chirpid != $t2.chirpid
        let $d := spatial-distance($t1.sender_location, $t2.sender_location)
        order by $d desc
        limit 10
        return $t2.message_text
}
```

Listing 3.28: Q16 (in AQL for the nested schema)

3.5.3 Data Modification Operations

The second group of operations in BigFUN consists of data modification operations, shown in Table 3.1. The goal for their inclusion is to study a system's behavior when dealing with data addition (insert) or data removal (delete). We made the following decisions while designing the data modification portion of the workload for BigFUN:

- To examine the impact of complex records and index structures (which must be maintained properly through data modifications) on the cost of updates, we used GleambookUsers as the target dataset for this workload. It contains records with nested and collection attributes as well as a secondary index.
- Real applications do inserts and deletes both individually and in batches. The BigFUN micro-benchmark includes both singleton and bulk operations, with varying sizes, to examine the performance impact of grouping data modification operations together.

As shown in Table 3.1, a single insert (or delete) operation adds (or removes) a record and all its attributes' values from the database. As a result, if a system stores the target dataset in a normalized fashion, a single data modification operation turns into multiple corresponding operations on several datasets. If a GleambookUser record is stored as described in Figure 3.2, then three datasets - GleambookUsers, Employments, and FriendIds - need to be mod-

ified in each operation. Performing multiple correlated changes can add extra cost for the data modification workload compared to the case where the records are stored in a single nested dataset.

Batch insert

This operation modifies the contents of the GleambookUsers dataset by adding full information about a set of non-existing users. (Check Listing 3.29.)

```
insert into dataset GleambookUsers (
for $t in [
        @ADM-Full-Rec-1,
        @ADM-Full-Rec-2,
        ...,
        @ADM-Full-Rec-k
        ]
return $t
);
```

Listing 3.29: U1 (in AQL)

Batch delete

This operation modifies the GleambookUsers dataset by removing all the information about a set of existing users which are identified via their primary key values. (Check Listing 3.30.)

```
delete $u from dataset GleambookUsers
where   $u.id = @int64-1 or
        $u.id = @int64-2 or
                ...
        $u.id = @int64-k;
```

Listing 3.30: U2 (in AQL)

3.6 Experiments

This section presents a set of experimental results that we obtained by running the BigFUN micro-benchmark on the systems listed in Section 3. We first go over the details of the

	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	Mongodb
9 Partitions	169	285	292	18	276
18 Partitions	338	571	587	36	553
27 Partitions	508	856	881	53	833
Updates	162	279	272	-	278

Table 3.2: Total database size (in GB)

setup for the systems and the tests. Then, the results for the read only queries and data modification operations are presented separately.

3.6.1 Setup

For the experiments, we used a 10-node IBM x3650 cluster with a Gigabit Ethernet switch. Each node had one Intel Xeon processor E5520 2.26GHz with four cores, 12 GB of RAM, and four 300GB, 10K RPM hard disks. Each node was running 64-bit CentOS 6.6, and on each machine three disks were used as data partitions to store each system's persistent storage files. The fourth disk was used as the log partition for the transaction and system logs.

Client

The BigFUN client driver code was running on a separate machine with 16 GB of memory and 4 CPU cores (hyper threaded) that was connected to the main cluster through the same Ethernet switch. While running the workload, we monitored the resource usage on the client machine to make sure that it was never the performance bottleneck. A single-user, closed-system model was used for our tests; queries were independent of one another, and each new request was only triggered once the previous one was finished. As the performance metric, we measured and report the average end-to-end response time per operation from the client's perspective. We also ran each workload multiple times, considered the first few initial runs

as warm-up rounds and discarded them, and reported measurements on a warmed-up cache for each system.

System-X

For System-X, we used a version dated approximately 2013. Each node in the cluster served 3 database partitions. Data was partitioned using the system's built-in hash-partitioning scheme. We used the system's storage manager to maintain the tablespaces. Our client used the vendor's JDBC driver to run the workload.

Hive

For Apache Hive, we used stable release 0.13. The client driver used Hive's JDBC client through *hiveserver2* to execute queries against the data warehouse and retrieve the results. The data was stored using Hive's *Optimized Row Columnar* (ORC) file format [14], which has inherent optimization for storage savings and enhanced read performance when a query needs a subset of the rows and/or the columns within them. We used 4 map and 4 reduce slots per node. The machines in the cluster were running datanode and tasktracker daemons. Three disks per machine were used by HDFS to store data (with a replication factor of 1) and the fourth disk served as the log partition. The NameNode and JobTracker daemons were running on a separate machine (with 4 cores, and 16 GB of memory) connected to the cluster using the same Ethernet switch.

MongoDB

For MongoDB, we used version 2.6.3. The client used MongoDB's Java Driver 2.12.4 to interact with the document store. Data was stored in sharded collections, using hashed shard keys. On each machine, we put three shards (on three separate disks) and three mongod processes, one per shard. We used the fourth disk on each machine to store MongoDB's journal files.

AsterixDB

For AsterixDB, we used version 0.8.7. The native RESTful API was used to run the workload. The data was stored in internal datasets, hash-partitioned using the primary key of each dataset. One node controller with three data partitions was running on each machine. We assigned 6 GB of memory to each node controller, along with 1 GB of bufferpool. We measured AsterixDB's performance for two different data type approaches: *AsterixSchema*, where the data type definitions pre-declared all possible attributes for each data type, and *AsterixKeyOnly*, in which the data type definitions only defined the minimal set of attributes (required for indexing) in each data type. These two variations lie on the two end-points of the data type definitions spectrum (for semi-structured records) in AsterixDB: providing the system with complete vs. the least amount of information about the attributes in a data type. The overhead of the information stored per record differs between these two cases.

3.6.2 Read-Only Workload Results

Our read-only experiments focused on the scale-up characteristic of the tested systems. We used three scales with 9, 18, and 27 partitions using 3, 6, and 9 machines respectively. (Each machine served 3 partitions.)

The data generator described in Section 3.5 was used to create the data for the three scales. For the 9-partition scale, 90 million GleambookUsers, almost 450 million GleambookMessages, and more than 220 million ChirpMessages were generated and used. These cardinalities were scaled up proportionally for the 18 and 27-partition scales. We tried to place a reasonable amount of data on each node based on the available memory per machine so that when running the tests against a warmed-up system, read and write requests turn to be physical IO requests and are not simply served by the OS cache. For this purpose, the amount of data in our final loads for each scale was at least five times the total available

Dataset	Attribute	AsterixDB	MongoDB
GleambookUsers	user_since	BTree	BTree
GleambookMessages	author_id	BTree	BTree
ChirpMessages	send_time	BTree	BTree
ChirpMessages	sender_location	RTree	2d-IX
ChirpMessages	message_text	inverted-IX	text-IX

Table 3.3: Nested schema - Secondary index structures

Table	Column	System-X
GleambookUsers	id	BTree (PIX, Clustered)
GleambookUsers	user_since	BTree
GleambookMessages	message_id	BTree (PIX)
GleambookMessages	author_id	BTree (clustered)
ChirpMessages	chirpid	BTree (PIX, Clustered)
ChirpMessages	send_time	BTree
Employments	id	BTree (clustered)
FriendIds	id	BTree (clustered)
ReferredTopics	chirpid	BTree (clustered)

Table 3.4: Normalized schema - Index structures

memory size across the machines. We loaded the data into AsterixDB and MongoDB as records with nesting as shown in Figure 3.1. For System-X and Hive, we used the normalized schema as described in Figure 3.2. Table 3.2 shows the total database size per system after loading. As the systems use different storage formats to persist their data, they differ in terms of the total database size. One can see a size difference between the *AsterixSchema* and *AsterixKeyOnly* cases because of the extra information stored per record in the latter case. One can also see the impact of the storage optimization in the ORC file format for Hive; ORC's built-in compression reduced the total size of the stored tables significantly.

Auxiliary index structures

Table 3.3 lists the secondary indices that we created in AsterixDB and MongoDB for the nested schema. The indices on *user_since* (GleambookUsers dataset) and *send_time* (ChirpMessages dataset) attributes were extensively used to filter and select a subset of users

or chirp messages in various queries. The *author_id* index (GleambookMessages dataset) was exploited for join queries. The spatial index on *sender_location* and the text index on *message_text* (both in ChirpMessages) were used for the spatial and text similarity queries respectively. AsterixDB and MongoDB use different types of indices for spatial and text fields. AsterixDB uses RTrees for indexing spatial data types, while MongoDB uses a 2D index (created by computing "geo-hash" values for the coordinate pairs while the two dimensional map is divided into quadrants recursively [23]). For indexing textual data, AsterixDB has support for n-gram inverted indices that can be used for both exact text search and text similarity search. For the purpose of our experiments, we created the inverted index on 2-grams. MongoDB's more traditional text index is created by using stemming, filtering stop words, etc.

Table 3.4 shows the indices used for the normalized schema in System-X. Besides showing the base table and column for each index, we have noted the clustered indices. A clustered index specifies how the rows of a table are physically ordered in a tablespace. Clustering can provide a considerable performance advantage for operations that involve accessing many records. While the primary index was used as the clustered index in both the GleambookUsers and ChirpMessages tables, we clustered GleambookMessages rows based on the *author_id* attribute, as our workload tended to access this table for fetching all the messages sent by a specific user (rather than accessing the messages directly using their primary key values). Moreover, as Employments, FriendIds, and ReferredTopics are three tables created as a result of normalizing the schema, we used indices on the parent tables' id attributes as their clustered indices. These tables did not have a primary index, as they were only accessed through their parent tables. We included corresponding referential constraints for them, defined as foreign keys, when executing the DDL. We gathered statistics in System-X to provide its cost-based optimizer with the required information to generate optimal plans. We did not use indices in Hive because of their level of support; unlike other systems, the optimizer in Hive does not automatically consider indices when generating a plan, so users

need to re-write each query in such a way that indices can be exploited.

As MongoDB has no built-in support for joins, we had to perform its join operations on the client side. For this purpose, we added a code snippet to the client program that performed an index nested loop join using primary and secondary indices on the collections involved in the query.

Performance results

Table 3.5 shows all the results for the read-only queries. Each row shows the average response time (in seconds) for one query across all systems for all three scales. If we could not run a query against a system (due to the fact that the features tested by the query were not directly supported by that system), the corresponding cell in the table contains a "-". A cell with "NS" means that the query did not scale properly and did not produce reliable results for that case. Each query may have several versions whose results are reported in separate rows. For most queries we have 3 versions: small (S), medium (M), and large (L). These versions are defined with respect to the expected selectivity of the filters in a given query. Each filter could be of one of the following types:

- **Primary key filter:** It defines a range of valid existing ids for GleambookUsers. The small version is chosen so as to select 100 users; the medium version selects 10,000 users, and the large version selects 1 million users.
- **Temporal filter:** It defines a range of valid timestamps according to the start and end dates the generator used during the record creation process. Depending on the dataset used in the query, the filter is defined either on the *user_since* attribute (in GleambookUsers) or the *send_time* attribute (in ChirpMessages). For the base scale (9 partitions), an expected number of 100 records would pass the filter in the small

version; the number is 10,000 for the medium version, and 1 million for the large version.

- **Spatial filter:** It defines a circular neighborhood whose center is a valid point within the region the generator used when creating values for the spatial attributes. Changing the radius of the neighborhood changes the selectivity of the filter. The filter is applied on the values of the *sender_location* attribute in the ChirpMessages dataset. In the 9-partition scale, the small version picks an expected number of 100 records; for the medium and the large versions, this value is 10,000 and 1 million records respectively.

- **Text filter:** It uses a keyword selected from the dictionary of keywords that the generator used in creating message texts. The filter is applied on the *message_text* attribute in the ChirpMessages dataset and selects a subset of the chirp messages in the text similarity search queries. The small version picks a keyword with an expected occurrence frequency of 5% in the whole dataset, while the medium version picks a keyword that exists in approximately 20% of the records.

In Table 3.5, two of the aggregation queries, Q6 and Q8, have a fourth version, denoted by F (standing for "full"). In this version, the query runs against the full dataset (no pre-aggregation filter) and computes the aggregation over all records. Such aggregations arise in OLAP types of workloads, so we included these versions to measure the performance in such scenarios.

For the queries involving joins (Q12 to Q15), we report results for two different variations of the query: one in which no secondary index exists on the author_id attribute from the messages side of the join, and one in which this attribute is indexed (this variation is denoted by an "ix" suffix in Table 3.5). We considered these two cases since existence of the secondary index could change the actual join technique a system would use. In AsterixDB, the user can add a hint to the query's join predicate to switch from hybrid hash join (its default technique

for equi-joins) to an indexed nested loop join. Currently, indexed join is only available for inner joins; For left-outer joins (cells with a * in Table 3.5), a hybrid hash join is always used. For MongoDB we performed a client-side indexed nested loop join. In System-X, the cost-based optimizer picks the indexed technique when it is statistically expected to out-perform the hybrid hash technique. Our client-side join for MongoDB used the indexed nested loop join technique driven by the client program.

Below, we go over some important details about the read-only queries' results, case by case.

General observations

Hive does not have support for automatic use of secondary indices. As a result, it needs to perform a full scan on the involved dataset(s) in all the queries. The ORC format has built-in optimizations that make scanning faster than other file formats in Hive, but the requirement to access all the records worsened Hive's performance for most of the queries, specifically the short ones, compared to the other systems. (To be fair, Hive was designed for batch jobs over large datasets and not for OLTP types of workloads or real-time queries.)

MongoDB's performance degrades considerably when going from small to medium queries. In addition, for most of the large queries, we faced issues that prevented us from obtaining reliable results. We hit memory related issues in running the mapReduce command, and we observed that the results of selections on secondary indices were hitting result cursor time-outs on the server side. By switching to "immortal" cursors, results came in bursts from the shards, and the cursors were idle for long periods of time waiting for more results, and they returned many documents in periodic sudden spikes. These behaviors happened because our queries tended to be of "scatter-gather" style, where all shards had documents that could contribute to the final results. MongoDB is known to behave well for short queries that need to access one or a few shards, and not for large scatter-gather like queries. That is why for most of the large versions of the queries we have "NS" for MongoDB in Table 3.5.

Full record retrievals

For the single dataset, simple queries (Q1, Q2, and Q3), AsterixDB and MongoDB needed to access only one dataset (GleambookUsers) as a result of storing their records with nesting. However, because of normalization, System-X and Hive had to access three tables to fetch all the attributes in the qualifying records and combine them together (we used "UNION ALL" or joins, depending on the query, to combine the values). These steps added their own overheads that, for example, showed their negative impact in the large version of Q3.

Aggregations

For the aggregation queries (Q6, Q7, and Q8) in MongoDB we used the mapReduce command since the strings' length values were used in the aggregation. This command hit memory issues for the large and full versions of the queries, and we had trouble getting reliable performance.

The full version of Q6 ran a global aggregation on all the records in the ChirpMessages dataset. As a result, all systems needed to perform a full dataset scan to access every record. Hive benefited from its compact storage format and achieved better performance for this version of Q6 compared to other systems. In the full version of Q8, the overhead of grouping and ranking reduced this storage-format related performance advantage for Hive.

Spatial Selection and Text search

We report the results for this group of queries only for AsterixDB and MongoDB, as they have built-in support for loading and indexing both spatial and textual data. The two systems use different techniques to index such data types. The results for Q9 showed AsterixDB performing more robustly, compared to MongoDB, for the medium and large versions of the spatial selection query. In AsterixDB, spatial predicates were evaluated via an R-Tree, while MongoDB used a 2D-index and its $geoWithin operator against this geospatial

index. The large version of Q9 in MongoDB suffered from the "scatter-gather" query issue described before. For the exact text search (Q10), AsterixDB used an inverted index and outperformed MongoDB, which was using its own text index. AsterixDB's inverted index was also used for text similarity search (Q11), where the performance of the system was comparable to Q10.

Joins

In the small versions of queries that involve joins (Q12 to Q15), the indexed nested loop join technique outperformed hybrid hash join by orders of magnitude, as would be expected. The indexed technique lost its performance advantage in the medium and large versions as a result of performing too many random I/Os, while the hybrid hash technique showed more robust performance.

Another important observation in Q12 and Q13 is the drop in the performance of the client-side join in MongoDB. While the client program showed reasonable performance for the small versions of the queries, it quickly worsened for the medium version as a result of running too many find() requests against the GleambookMessages collection.

3.6.3 Data Modification Workload Results

Our data modification experiments measured systems' performance when processing batches of insert or delete operations against the GleambookUsers dataset. We ran this set of tests on 9 partitions. We considered two batch sizes: a batch size of 1 to measure a system's base performance when dealing with a single operation, and a batch size of 20 to check the impact of grouping similar operations together. In the insert workload, each record in a batch was a valid, non-existing GleambookUsers record with values for all attributes (including the employment history and friends' ids). The delete workload deleted existing users in the

	9-partitions					18-partitions					27-partitions				
	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB
Q1	0.045	0.047	0.06	55.24	0.017	0.046	0.048	0.064	55.71	0.019	0.048	0.053	0.07	55.95	0.023
Q2-S	0.089	0.096	0.102	72.67	0.029	0.104	0.125	0.108	73.61	0.042	0.126	0.135	0.113	74.1	0.054
Q2-M	0.35	0.387	0.354	76.92	1.124	0.378	0.403	0.363	78.11	1.24	0.433	0.474	0.377	78.13	1.354
Q2-L	6.027	6.066	15.36	251.9	107.2	15.37	15.68	16.34	250.5	130.2	24.92	25.31	17.36	255.1	152.6
Q3-S	0.278	0.279	0.576	815.9	0.113	0.299	0.322	0.657	856	0.127	0.3	0.369	0.762	898.1	0.135
Q3-M	6.439	6.647	39.86	824.7	32.82	6.784	7.223	40.51	861	70.32	7.067	7.532	41.43	903.5	106.2
Q3-L	29.24	43.32	279.4	947.1	NS	32.04	47.48	294.1	1031	NS	43.58	63.81	312.8	1314	NS
Q4-S	0.28	0.307	0.335	526.6	0.133	0.316	0.336	0.34	545.3	0.138	0.319	0.359	0.345	570.5	0.141
Q4-M	6.503	6.603	15.66	533	49.07	6.715	6.881	15.87	554.2	117	6.732	7.491	16.27	575.9	183.3
Q4-L	31.56	44.31	76.85	583.8	NS	37.89	54.01	78.01	635.2	NS	57.81	80.72	80.13	686.9	NS
Q5-S	0.109	0.206	0.319	598.4	0.132	0.214	0.242	0.328	637.1	0.136	0.236	0.248	0.336	666.8	0.141
Q5-M	6.061	6.229	20.16	607.4	47.69	6.234	6.606	19.77	641	109.5	6.279	6.756	20.95	676.6	171.9
Q5-L	29.4	43.31	193.5	631.6	NS	33.52	51.92	194.6	648	NS	48.51	71.49	194.6	684.3	NS
Q6-S	0.246	0.247	0.126	51.29	0.174	0.25	0.268	0.135	50.91	0.19	0.271	0.281	0.148	52.07	0.208
Q6-M	6.769	6.806	4.581	51.82	9.057	6.78	6.865	4.776	52.37	10.13	6.872	6.886	4.924	52.94	11
Q6-L	76.51	112.5	90.98	53.86	NS	89.44	113.9	91.4	54.81	NS	93.99	116.5	93.32	56.02	NS
Q6-F	91.66	192.4	68.04	57.86	NS	92.59	200.3	70.12	58.8	NS	96.91	201	71.65	59.95	NS
Q7-S	0.468	0.475	0.133	54.55	0.199	0.553	0.612	0.136	55.81	0.204	0.712	0.740	0.136	56.41	0.212
Q7-M	6.975	7.022	4.739	57.49	9.987	7.082	7.304	4.901	58.11	10.61	7.411	7.687	4.925	59.92	11.3
Q7-L	96.11	200.6	89.43	66.1	NS	96.94	200.6	92.35	73.22	NS	100.1	200.7	94.69	80.54	NS
Q8-S	0.513	0.586	0.139	69.73	0.201	0.836	0.938	0.143	72.45	0.211	1.067	1.27	0.146	75.18	0.219
Q8-M	7.049	7.366	5.065	71.34	10.6	7.219	7.715	5.158	74.11	11.42	7.695	8.657	5.244	77.94	11.9
Q8-L	102.6	200.6	90.3	77.46	NS	147.63	200.7	93.11	78.98	NS	173.1	201	95.28	80.68	NS
Q8-F	200.8	323.8	135.9	151.1	NS	222.2	349.2	155.9	173.9	NS	236.1	386.8	176.9	196.3	NS

Table 3.5: Read-only queries - Average response time (in sec) - Part 1

	9-partitions					18-partitions					27-partitions				
	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB
Q9-S	4.96	18.85	-	-	1.1	6.072	23.07	-	-	1.42	6.085	23.55	-	-	1.748
Q9-M	11.64	24.65	-	-	46.9	11.77	33.9	-	-	103.8	12.4	36.61	-	-	160.9
Q9-L	97.77	183.8	-	-	NS	102.9	193.3	-	-	NS	111.6	200.9	-	-	NS
Q10-S	94.54	174	-	-	990.7	95.82	175.7	-	-	1233	96.7	200.8	-	-	1481
Q10-M	100.2	190.4	-	-	1297	101.5	196	-	-	1393	103	198.3	-	-	1492
Q11-S	97.61	200.4	-	-	-	98.61	200.6	-	-	-	102	200.8	-	-	-
Q12-S	133.7	176.5	114	237.5	-	137.9	178.2	116.7	267.9	-	138.3	187.6	118.9	300	-
Q12-M	142.3	182.7	118.1	239.1	-	144.6	184.5	120	270.8	-	145.5	193.9	123.9	301.6	-
Q12-L	164.1	200.3	145	240.1	-	171.2	220.3	148.2	270.9	-	176.2	234	151.1	302	-
Q12-S-IX	1.078	1.091	1.219	-	1.149	1.703	1.823	1.918	-	2.769	2.289	2.524	2.552	-	4.396
Q12-M-IX	35.16	49.98	58.81	-	455.8	48.35	69.24	62.42	-	533.1	49.25	71.9	66.1	-	612.3
Q12-L-IX	120.9	192.1	142.4	-	NS	149.8	229.2	145.2	-	NS	155.7	241.3	146.9	-	NS
Q13-S	135.1	177.8	114.3	630.7	-	139	178.8	118.1	720.7	-	139.3	187.7	120.4	811.9	-
Q13-M	143.3	186.4	119.1	641.5	-	147.7	187.3	122.7	726.9	-	148.2	195.1	125.8	814.1	-
Q13-L	167.9	216.6	150.2	644.5	-	172.8	224.3	150.8	740.6	-	178.9	237.9	151.6	837.1	-
Q13-S-Ix	*	*	1.356	-	1.425	*	*	2.015	-	3.059	*	*	2.673	-	4.677
Q13-M-Ix	*	*	59.21	-	465.7	*	*	62.72	-	545.3	*	*	66.23	-	623.9
Q13-L-Ix	*	*	145.9	-	NS	*	*	146.7	-	NS	*	*	148.8	-	NS

Table 3.5: Read-only queries - Average response time (in sec) - Part 2

	9-partitions					18-partitions					27-partitions				
	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB
Q14-S	140	184.6	114.2	293.5	-	144.5	188.7	117.1	296.9	-	144.8	189.5	118.9	299.7	-
Q14-M	148.5	193.8	119.2	296.6	-	153.2	197.4	120.5	298.8	-	154.5	199.7	122.6	302.1	-
Q14-L	166.6	213.4	151.3	297.5	-	169.3	222.6	152	301.1	-	173.3	235.1	151.5	304.5	-
Q14-S-Ix	1.142	1.174	1.423	-	-	1.714	1.834	2.051	-	-	2.293	2.709	2.677	-	-
Q14-M-Ix	37.51	52.61	59.16	-	-	50.42	71.48	62.81	-	-	51.51	72.14	66.48	-	-
Q14-L-Ix	121.1	193.6	142.8	-	-	152.7	235.4	145.9	-	-	157	243.7	147.4	-	-
Q15-S	140	184.8	114.5	313.7	-	144.6	189.8	117.1	315.7	-	144.9	189.8	119.2	319.2	-
Q15-M	148.8	193.9	120	315.3	-	153.3	198.2	123.2	319.7	-	154.7	200.9	125.5	323.1	-
Q15-L	169.5	216.1	151.3	316.7	-	170.6	224.6	151.4	320	-	174	235.3	151.7	323.3	-
Q15-S-Ix	1.25	1.479	1.467	-	-	1.81	1.941	2.161	-	-	2.475	2.757	2.842	-	-
Q15-M-Ix	38.01	53.39	59.73	-	-	51.91	73.87	62.97	-	-	53.3	74.32	67.57	-	-
Q15-L-Ix	121.8	195.6	143.6	-	-	152.8	238.7	145.3	-	-	157.2	244	148.8	-	-
Q16-S	1789	1876	-	-	21.89	1911	2014	-	-	57.41	2147	2236	-	-	90.19
Q16-M	1912	1996	-	-	68.41	2053	2207	-	-	148.1	2165	2482	-	-	230.7
Q16-L	4062	4315	-	-	NS	4662	4789	-	-	NS	4972	5052	-	-	NS

Table 3.5: Read-only queries - Average response time (in sec) - Part 3

dataset based on randomly selected ids. In AsterixDB and MongoDB where the schema contained records with nesting, only one dataset (GleambookUsers) along with its secondary index (on the user_since attribute) needed to be updated for each operation. In System-X, the normalized schema forced the system to update three datasets, GleambookUsers, Employments, and FriendIds, along with their indices. We did not include Hive for this set of experiments since Hive does not offer row-level updates and the life cycle of its data is normally maintained outside of the system.

AsterixDB uses LSM-trees at the storage layer to store internal datasets. AsterixDB's updates (inserts or deletes) modify the "in-memory" component of an index, while the system guarantees durability for the changes by recording the updates in the transaction log. We set the size of the "in-memory" component budget for the LSM indices to be 256MB per node controller in AsterixDB. In System-X, we increased both the number and the size of the transaction log files to improve performance, as suggested by the vendor. In MongoDB, we used the journaled write concern to provide the same durability level as AsterixDB and System-X. We also decreased the commit interval for the journal from its default value to 2ms to make sure that our serial update client was not limited by the MongoDB's group commit policy.

While running the data modification workload, we realized that the reported results will be reliable only if the systems have warmed up enough prior to the measurements. The duration of the warm-up phase for a system has to be carefully picked so that enough updates have been run against the system (according to the total available memory size) to take the system to a steady state before starting measurements, as what is measured in the early stages of an update workload (during the warm-up phase) is mostly the cost of in-memory updates. Such a number can be significantly smaller than the real cost of an update in a system that has been up and running for a long time. We therefore conducted our data modification experiments so that, during the warm-up phase, we performed enough updates to make the

systems' buffer cache be filled with dirty pages. Our real measurements happened once the expected cache misses and IOs were happening per operation.

Table 3.6 shows the performance of the systems when running batches of inserts and of deletes. For each case, we show the average response time in milliseconds from the client's perspective for "one" operation. When the batch size was 20, we measured the average response time per batch (shown within () in the table) and divided it by the batch size to calculate this number. Grouping updates together lets a portion of the overhead in running them to be amortized over the operations in the group; this can reduce the average per-record response time. Table 3.6 shows that all three systems benefited from batching, but the level of improvements differed among them. AsterixDB benefited the most, moving from being the slowest when dealing with single updates to the fastest in processing groups of updates. This is due the current overhead in AsterixDB for compiling and generating jobs that happens for each request. System-X did not benefit as much as the others from batching. This was mainly because of its normalized schema and the fact that it needed to run multiple updates against several structures for a single request. We also used JDBC's "prepared statements" to execute updates in System-X, which reduced a portion of the overhead by letting the database run SQL statements without having to compile them first.

Support for continuous data ingestion is a unique feature in AsterixDB that enables the system to process fast streams of incoming data (feeds) efficiently. AsterixDB uses a long running job that is scheduled to execute on the cluster to process data feeds, which removes the extra overhead of repeatedly creating query plans and setting up jobs in performing updates. Table 3.6 also includes the performance of AsterixDB when we ran the same insert workload, described above, using data feeds. We report the average insert time per record in milliseconds, which is several orders of magnitude faster compared to the U1 numbers.

	Batch Size	Asterix *Schema*	Asterix *KeyOnly*	System-X	Mongodb
U1	1	73.75	73.97	46.34	13.85
U1	20	6.20	6.23	30.15	7.9
U2	1	73.96	79.3	49.01	19.93
U2	20	4.73	4.89	33.79	14.2
Feeds	-	0.029	0.031	-	-

Table 3.6: Data modification - Avg response time (in ms)

3.7 Discussion

In this section, we summarize the important lessons we learned from this effort.

L1. Flexible schemas and their impact on performance

Many Big Data applications need to deal with complex records with heterogeneous structures and nested or collection attribute values. Unlike systems with the relational data model, which require all the records in a table to strictly comply with its schema, most NoSQL systems are able to store and process heterogenous records in a straightforward manner. In fact, many of them consider this characteristic as one of their design points. However, this flexibility normally comes with extra overhead at the storage level, as the system needs to store enough metadata per record for later parsing. While some systems, like MongoDB are completely schemaless (always with maximum flexibility), data type definitions in AsterixDB let users decide about the level of flexibility in the schema and its expected impact on performance. A data type can be fully schemaless (similar to the AsterixKeyOnly data type definition in our experiments) or it can provide as much as information as possible (similar to the AsterixSchema data type definition in our experiments). The extra overhead in storage size can directly impact a system's performance for queries performing large scans or those that access a large number of records. Large and full versions of aggregation queries (Q6-L/F, Q7-L, Q8-L/F) in BigFUN were examples of such queries. Comparing the performance in AsterixSchema case vs. AsterixKeyOnly case, we can see that the AsterixKeyOnly

case was worse (almost proportional to the difference in the storage size between these two data type definitions) as it needed to deal with records with a larger size. Moreover, if we compare the performance of AsterixSchema against System-X, which requires a full schema definition, we can see that AsterixDB achieved comparable performance once we provided it with complete data type information.

L2. Pros and cons of normalized schemas

In a relational system, records are stored with normalization. This requirement impacts the performance for specific types of queries. Depending on the query, this impact can be positive or negative. For those queries that include full record retrieval for a large number of records, the system needs to access more than one table to fetch the normalized attribute values for a record and combine them together. A similar situation exists for record insertions or deletions, in which several structures need to be updated per data modification request. These structures are the parent and children tables containing different parts of each record along with the auxiliary index structures (typically clustered) which are created on the children tables to avoid full scans during read operations. As an example, consider System-X's performance vs. AsterixSchema's for queries Q2-L and Q3-L in Table 3.5, where System-X needed to access three tables (GleambookUsers, Employments, and FriendIds), whereas AsterixDB found all the information in one dataset. The performance for operations U1 and U2 in Table 3.6 (batch size 20) shows the negative impact of modifying three tables and their clustered indices on System-X's performance for updates compared to AsterixDB and MongoDB. However, for queries that include projection and just need a subset of attributes that exist in one of the normalized tables, a system with a normalized schema does not fetch unwanted nested data; in a nested schema, all attributes are stored together per record and the system needs to read them all during read operations. Reading less data can improve performance once a large number of records are involved in a query. The large and full versions of the aggregation queries in Table 3.5, such as Q6-F or Q7-L, are examples of

this case, where System-X did not retrieve *referred_topics* for the chirp messages as it only needed the *send_time*, *message* and user's *screen_name* attributes from the parent table ChirpMessages. In this case, AsterixDB could not avoid fetching *referred_topics*, as the dataset was stored with nesting. (Of course AsterixDB's flexible type definitions would permit storing chirp messages with normalization as well, if one so chooses.)

L3. Optimized storage format and its performance gain

The total size of the stored data directly impacts the performance of a system in scanning a large number of records. The ORC format in Hive has built-in optimizations for reducing its stored size on disk and also for doing faster scans and projections. Table 3.2 shows that Hive's total stored data size is much smaller than other systems for this reason, and it helps the system to perform better comparing to other systems for some of the queries which include large scans. For example, Hive achieved the fastest time for Q6-F and Q7-L, where all or a large number of records needed to be accessed to calculate an aggregation.

L4. Advanced and mature query optimization

Decisions made during query optimization can have a significant impact on the ultimate performance of a system. The benefits of an efficient index structure or run-time operator may fade as a result of missing some known, effective optimizations that could have been included in the query plan. As an example in Table 3.5, Q9 (spatial selection) shows that the spatial index (R-Tree) in AsterixDB had a better performance comparing to MongoDB's 2D-index for the medium version, but in Q16 (spatial join), AsterixDB was orders of magnitude slower than MongoDB's client-side join for the medium version. Beside the difference that the systems have in terms of the type of spatial index they use, in this query MongoDB pushed the limit clause (finding first 10 chirp messages) down into sorting, which helped it to simply skip processing a huge, but unnecessary, number of documents. AsterixDB, on the other hand, failed to apply this optimization and lost the potential advantage of its efficient

spatial indexing for this query. Another example, which has been known for a long time, is picking the indexed nested loop join technique instead of hybrid hash join whenever it can result in better performance (based on the join predicate and the expected selectivity of the filters before joining). System-X is able to make such a decision automatically, while AsterixDB requires a hint for this purpose (check Q12 in Table 3.5). In fact, the mature, cost-based optimizer and advanced query evaluation techniques that System-X uses enabled it to generate more efficient plans and achieve robust behavior across the different versions of a query, while it also showed stable performance characteristics in scaling up. In general, this is an expected advantage (at least for now) for the relational database systems as compared to NoSQL solutions [66].

L5. MongoDB performance issues

MongoDB is a representative system for NoSQL data stores, and based on its design points it targets a narrow, yet popular, set of use cases and shows reasonable performance for short queries against a single collection which tend to access a small number of documents coming from one or a few shards. We observed a significant performance drop when switching to medium queries with a scatter-gather nature, and the system showed unstable performance for large queries (check Q3, Q4, and Q5 as examples). Similar issues existed in running its mapReduce command for large aggregation queries (check Q6, Q7, and Q8). By using a client-side join to run queries that access more than one collection, we observed that MongoDB's performance dropped quickly as the number of qualifying documents grew because of the increased number of separate find() requests sent to the system by the client. The join performance can be improved (for example by batching lookups as shown in [50]) but client-side joins generally suffer from a scalability issue.

L6. Job generation path and its overhead

The results for updates in Table 3.6 show that the overall query path and job generation

task in AsterixDB currently add significant overhead to the performance of the system for small operations. In our update tests, AsterixDB went from being the slowest system to fastest one when similar operations were grouped together in a batch so that its job generation overhead was amortized by several operations. There are a number of known solutions for this problem such as parameterized queries, query plan caching, and simplified plan serialization, that AsterixDB still needs to add. Unlike AsterixDB, System-X (because of its mature built-in optimizations) and MongoDB (because of its narrower operations scope) have built-in means to avoid this overhead.

L7. Performance changes in running updates

A system's performance can change significantly between initial and later stages of update tests, especially in "update in-place" systems (MongoDB and System-X in our tests). This is mainly due to the fact that the buffer cache (in the system and/or the OS) normally has enough free space available in the early stages of running updates to serve the requests. However, once enough updates happen (according to the total available memory size), the number of dirty pages in the buffer cache increases and the system needs to perform page evictions and flushing of writes to make the older changes persistent on disk and create room for the newly requested pages. The overhead of these operations increases the average response time in serving an update request, and in fact this slower time is the correct time to report as it is the expected performance that one would get from a system that has been up and serving requests for a while.

L8. "One size fits a bunch"

Looking at the overall performance of AsterixDB across different operations, one could see that although AsterixDB did not always outperform the other systems, it did not scarifies its core performance by delivering the widest range of features and functionality. In fact, in most cases, AsterixDB was able to offer a performance level comparable to the fastest system. This

performance study supports a "one size fits a bunch" conjecture (which argues for a solid system with a rich set of features to serve different use cases rather than involving multiple narrower systems and stitching them together), and it shows that it is indeed possible to build such a robust system as a solution.

L9. Revealing functional and performance issues via benchmarking

One of the goals of benchmarking can be identifying unknown functional or performance issues in a given system. Running BigFUN micro-benchmark achieved this goal by identifying a number of issues in AsterixDB during the course of this work. Some of these issues have already been fixed by the developers. Examples are issues which were related to evaluating spatial or text-similarity predicates and queries involving disjunction. There are on-going efforts to address the remaining open issues, such as the overhead of the query path, which we already pointed to.

3.8 Conclusion

In this chapter, we reported on an evaluation of four representative Big Data management systems according to their features and functionality set and the performance that they provide for Big Data operations. We used a micro-benchmark, BigFUN, as the basis of our study. We described the schema and operations (including read-only queries and updates) in the micro-benchmark, and reported scaleup results for read-only workloads as well as the update performance of the systems. This work has attempted to look at benchmarking Big Data systems from a different angle by evaluating the systems with respect to their feature vectors and measuring the base performance of each feature. We believe this direction in Big Data benchmarking is important, as it is expected that Big Data systems will eventually converge on the set of features and operations provided to process both OLAP and OLTP

types of workloads efficiently. Interesting extensions to this work could be expanding the set of systems to include other Big Data systems and frameworks; considering non-uniform distributions for the data and query predicates to simulate some popular operations arising in Big Data applications (such as fetching the latest messages or most popular users), and adding multi-client tests with a focus on measuring the overall throughput of a system.

Chapter 4

Performance Evaluation of Big Data Analytics Platforms

4.1 Overview

This chapter of the thesis looks at another major class of workloads in Big Data applications: Big Data analytics (OLAP-style) workloads. Using the read-only queries from the TPC-H benchmark [26], we study and compare four representative Big Data frameworks: Hive [3], Spark [4], AsterixDB [36], and a commercial parallel relational database system (the same one used in Chapter 3) which we refer to as System-X. In terms of their query processing capabilities, all of these systems have rich query APIs and built-in runtime operations that enable them to process all 22 TPC-H queries. On the other hand, these systems also have major differences in terms of their architectures, preferred storage formats, and query optimization processes, which makes them a reasonable set of representative Big Data systems to explore with respect to the goals of this thesis. We present the results that we have obtained from running these systems at different TPC-H scales using various settings and we compare

them to one another. We also analyze a number of "interesting" queries in more detail. A follow-up discussion is included at the end regarding the lessons learned from this effort.

4.2 Motivation

Big Data is turning to be an essential factor in the process of making decisions at leading companies that seek to outperform their competitors in setting directions and serving their customers. Big Data analytics is an important tool that an organization can exploit in order to use the wealth of data it has access to. Big Data analytics is about collecting, storing, and analyzing large volumes of data efficiently with the goal of extracting invaluable, but hidden, information from it. According to a 2014 report by Forbes [7]: "87% of enterprises believe Big Data analytics will redefine the competitive landscape of their industries within the next three years. 89% believe that companies that do not adopt a Big Data analytics strategy in the next year risk losing market share and momentum".

Unlike transaction processing (OLTP-style workloads, considered in Chapter 2), which is characterized by a large number of (concurrent) short requests initiated by independent users, Big Data analytics workloads are characterized by fewer requests, submitted by fewer users (mostly experts and business analysts), with each query tending to be complex and resource intensive. Here is a list of characteristics that, depending on its complexity, an analytical workload normally has at least a few of [63]:

- Extreme data volume,
- Complex data model,
- Bulk operations,
- Multi-step/multi-touch analysis algorithms,

- Temporary or intermediate staging of data.

The response times of the queries in an analytical workload mostly tend to be on the order of tens or hundreds of seconds, and depending on the available resources, CPU, I/O, the network, or a combination thereof could be the processing bottleneck.

Because of the inherent differences between OLTP and OLAP workloads, the serving systems for them are also built in different manners. The key design points in an analytical processing framework are "functional richness" and "efficient resource usage". Moreover, as the requirements and demands of applications using them tend to evolve over time and become more complex, the richness of the API(s) to interact with the system and adequate expressive power in the supported query language(s) of the system (declarative languages are preferred) are among other design points for analytical processing environments.

Parallel relational databases and data warehouses were the prominent solutions for serving analytical workloads for a long time. With the advent of Big Data, however, various new frameworks have been developed to manage and run analytics to serve different applications such as log analysis, text analytics, or the task of organizational decision-making. Examples of these systems include Hive [3] (from Hortonworks), Presto [19] (from Facebook), Impala [12] (from Cloudera), Dremel [81] (from Google), Apache Drill [1] (from MapR), Spark SQL [42] (from Databricks), HAWQ [9] (from Pivotal), and IBM Big SQL (from IBM). These frameworks normally require an environment with characteristics such as high storage capacity, fast data movement capacity, and large amounts of available memory in order to achieve a desired performance level and proper (horizontal) scaling. In terms of their data storage, all of these systems can operate on data which is stored in HDFS. In terms of their APIs, they all provide SQL-based languages to describe queries. However, they differ significantly in terms of their architecture, optimization and code generation techniques, and run-time processing approaches. As analytical workloads tend to be complex and resource-

intensive, these differences can considerably impact their performance.

In this chapter, we evaluate four different systems picked from different parts of the Big Data platform space, in terms of their ability to execute an analytical workload. They are Hive, Spark SQL, AsterixDB, and System-X (the parallel RDBMS included in Chapter). For the benchmark data and workload, we use the read-only query suite from the TPC-H benchmark [26]. We chose to use TPC-H because its data model and workload are complex enough to serve as a reasonable set of analytics tasks. Moreover, many vendors are still using this benchmark to evaluate and report on the performance of their technology. The main goal in this chapter is to study the performance of these systems and their scale-up behavior with a focus on the impact of their different approaches to storage formats and query optimization. We report results for three TPC-H scales and analyze a number of "interesting" queries in more detail to extract important lessons. In the rest of the chapter, we first briefly discuss some related work and then review some details of the systems and the workload that we consider. The full set of details regarding the experiments and performance results come next, and are then followed by a discussion of the results.

4.3 Related Work

The performance evaluation of data stores for analytical workloads has been a major topic in the context of Big Data benchmarking. Among the released benchmarks by TPC, TPC-H [26] and TPC-DS [25] include analytical and OLAP-style queries with different levels of complexity. These benchmarks have been used in various works on Big Data benchmarking. From the Big Data community, one of the very first works on the topic of Big Data analytics was [86], which compared MapReduce (Hadoop) to RDBMSs. Being focused on basic architectural differences (e.g., programming model, expressiveness, and fault-tolerance) of the frameworks, the analytical workload in [86] included four simple, non-TPC queries:

filtered selection, grouped aggregation, join, and UDF aggregation. A more recent study that compared Hive against SQL Server parallel data warehouse (PDW) using the TPC-H benchmark, and showing that the relational systems can provide a significant performance advantage over Hive was [66]. Another recent paper [64] compared the performance of Hive vs. Impala using the read-only queries of the TPC-H benchmark and using TPC-DS inspired workloads. With a focus on the I/O efficiency of their columnar storage formats, the results in [64] showed that Impala was faster than Hive for different queries. Its main conclusion was to reaffirm the clear advantages of using a shared-nothing architecture for analytical SQL queries over a MapReduce-based runtime. Our work in this chapter has some overlap with [66] and [64] in terms of its workload (using TPC-H queries), and also in terms of one of the evaluated frameworks (Hive). However, the overall set of systems that we consider here, our assumptions, and the settings used for the tests are different.

4.4 Systems Overview

Of the four Big Data platforms that we consider in this chapter, three of them (System-X, Hive, and AsterixDB) were previously examined and summarized in Section 4 of Chapter 3. This section provides a brief overview of the fourth framework, i.e., Spark, and also of Tez [5] (a new runtime engine for use with Hive). We also describe the two storage formats that we use here for storing and using data in HDFS, namely Parquet [16] and ORC [14].

4.4.1 Apache Spark

Apache Spark [4] is a general purpose computing engine that runs on clusters of arbitrary size and uses a multi-stage, in-memory computational model to achieve fast performance. In addition to its core platform, Spark's ecosystem consists of multiple projects and libraries for

handling streaming data, graph processing, and machine learning tasks. Two key components that we are specifically interested in related to our goals in this chapter are Spark core and Spark SQL.

Spark core [103] is the central component of the Spark project and it is responsible for creating parallel tasks and scheduling them. *Resilient Distributed Datasets (RDDs)* [102], the programming abstraction that Spark uses for performing in-memory computation in a fault tolerant manner, is part of Spark core and is exposed via language-level APIs to Spark's applications and libraries.

Spark SQL [42] is a component on top of Spark core that integrates relational processing with Spark's functional programming API. *Data frames*, a data abstraction designed to support access to and computation on structured and semi-structured data, is included in Spark SQL. Another important part of Spark SQL is *Catalyst*, an extensible query optimizer. Catalyst uses features such as pattern-matching and composable optimization rules in conjunction with a general transforming framework to analyze query plans and do runtime code generation. In order to create data frames, write queries in SQL, and work with structured data in Spark SQL, Spark applications use SQLContext which is the entry point at the programming API level. Applications also have access to HiveContext which offers additional features such as supporting queries written in HiveQL and reading data from Hive tables [11].

4.4.2 Apache Tez

Tez [5] is an open-source framework designed to build dataflow-driven processing runtimes [91]. It generalizes the MapReduce paradigm to execute complex computations more efficiently. Data processing in Tez is expressed as a DAG that can be created via the programming API that Tez provides. A vertex in this DAG defines the application logic and resources needed to execute one step of the job. An edge defines a connection between producer and

consumer vertices and shows the data movement between them. Such a DAG models the computation at a logical level, and Tez expands this logical graph at runtime to perform the operations in parallel using multiple tasks, similar to Hyracks [49] and its role in AsterixDB [36].

Using Tez instead of MapReduce as the execution engine for Hive can improve performance by removing many inefficiencies that exist in query planning and query execution. Pipelining, using in-memory writes (skipping disk writes), and allowing for multiple reduce stages are some of the Tez primitives that lead to improved performance [10], again similar to the lessons learned from Hyracks [49]. An optimized query tree in Hive can be directly translated into a DAG in Tez. In addition, Hive adds some customized edges written in Tez's API for more complex operations such as joins. These algorithmic optimizations are combined with the inherent execution efficiency that Tez offers on top of Yarn [99] to help Hive gain significant performance benefits [91].

4.4.3 Storage Formats

The advantages of using columnar data are well known for analytical workloads in relational databases [30] [98]. The direct impact that columnar models have on the primary storage efficiency of a system and on moving only relevant portions of data into memory during query processing can help a system achieve significant performance gains when dealing with workloads that access a few columns across many rows and run costly operations on them. Columnar storage formats have been available for storing data in HDFS [41] for a while, and first started with the RCFile [77] and Trevni [28] formats. There have been discussions on the efficiency and performance of these formats (e.g., [65] on RCFile). Currently, the ORC format [14] (introduced by Hortonworks and Microsoft) and the Parquet format [16] (introduced by Cloudera and Twitter) are the two most popular and widely used columnar

storage formats for data stored in HDFS. As ORC is mainly optimized for Hive, and Parquet has become the suggested file format for Spark, we chose these two formats to run the TPC-H workload on to examine the impact of optimized storage formats on Big Data systems' performance. We briefly review each of these file formats below.

Optimized Row Columnar (ORC)

According to the ORC file format specification [15], ORC is an optimized version of the RCFile format and was created mainly for overcoming certain limitations in RCFile. ORC uses the type information obtained from a table's definition and utilizes type-specific writers that use lightweight compression techniques to create much smaller files. A user may also decide to apply other compression techniques, such as Snappy [21], on top of the built-in compression in order to achieve an even greater reduction in size. Besides compression, ORC achieves I/O efficiency via arranging data into *row groups* (of 10,000 rows) and including lightweight indexes that contain the minimum and maximum values for each column in each row group. Combining this lightweight index with a filter push-down optimization, ORC file readers can skip entire groups of rows that are not relevant to a query based on the query's predicate.

Internally, each ORC file stores data as *stripes* of row data along with a *file footer* that contains auxiliary information. Each stripe itself consists of three parts [15]:

1. Index data: This contains the minimum and maximum values for each column as well as the row offsets for each column.

2. Row data: This is the actual data, stored in a columnar fashion, which is accessed during scanning. Each column is stored as several "streams" that are stored next to one another in the file. The type of a column and its value determines the number of streams. For example, an Integer is stored as two streams: "PRESENT", which

is just one bit indicating whether the value is present or not (i.e. is it NULL), and "DATA", which captures the actual non-null values. Binary data, for example, uses another stream besides these two, a "LENGTH" stream that indicates the length of the value.

3. Stripe footer: This indicates the encoding of each column and includes a directory of streams.

The stripe size in an ORC file is configurable and is normally set as 256 MB.

Parquet

Inspired by Dremel's storage format [81], the Parquet file format also tries to benefit from columnar data representation and compression to achieve efficiency in I/O. Supporting complex nested data structures was also one of the key points considered in Parquet's design. Parquet stores data by considering logical horizontal partitions of rows, called *row groups*, where a row group in turn consists of a *column chunk* for each column in the dataset [17]. This is similar to PAX [34]. Column chunks are further sub-divided into *pages*, and each column chunk may have one or more pages. In Parquet, it is guaranteed that all of the pages for a specific column chunk will be stored contiguously on disk. Associated with each column chunk in a page, there is some *column metadata* which is part of the page header. The start locations of the metadata information are stored in the *file metadata* of a Parquet file. Readers use this metadata to find all of the column chunks that are needed for a specific query and to access them. Columns chunks are read sequentially.

Parquet was implemented so that multiple projects in the Hadoop ecosystem could all use it. There are currently several systems, such as Impala [12] and Spark SQL, that suggest it as their best practice for the storage format to use with them.

4.5 TPC-H Benchmark

TPC-H [26] is a decision support benchmark managed by the TPC [24]. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The TPC-H benchmark models decision-support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The workload that we use in this chapter consists of the read-only queries that are included in the TPC-H specifications.

4.5.1 TPC-H Database

The database for TPC-H consists of eight separate tables. Listing 4.31 shows the schema for these tables. There are a set of requirements included in the benchmark specification that should be met in populating the database. DBGen is a TPC-provided software package that can be used to produce the benchmark data. A single "scale factor" (SF) is used with DBGen and that describes the total size of the data in GB across all eight tables.

4.5.2 Queries

The read-only portion of the workload of TPC-H consists of a total of 22 SQL queries. At a high level, each query tries to answer a specific "business question" which illustrates the business context in which the query can be used [26]. A functional query definition, which defines the function that query has to execute, is included in SQL form in the benchmark specification. We used these functional query specifications as our queries for the systems that support SQL as their query language.

Among the four systems considered, only System-X was (currently) able to fully support the

```
NATION
        (N_NATIONKEY:INTEGER, N_NAME:TEXT, N_REGIONKEY:INTEGER, N_COMMENT:TEXT)

REGION
        (R_REGIONKEY:INTEGER, R_NAME:TEXT, R_COMMENT:TEXT)

PART
        (P_PARTKEY:INTEGER, P_NAME:TEXT, P_MFGR:TEXT, P_BRAND:TEXT, P_TYPE:TEXT,
        P_SIZE:INTEGER, P_CONTAINER:TEXT, P_RETAILPRICE:DECIMAL, P_COMMENT:TEXT)

SUPPLIER
        (S_SUPPKEY:INTEGER, S_NAME:TEXT, S_ADDRESS:TEXT, S_NATIONKEY:INTEGER,
        S_PHONE:TEXT, S_ACCTBAL:DECIMAL, S_COMMENT:TEXT)

PARTSUPP
        (PS_PARTKEY:INTEGER, PS_SUPPKEY:INTEGER, PS_AVAILQTY:INTEGER,
        PS_SUPPLYCOST:DECIMAL, PS_COMMENT:TEXT)

CUSTOMER
        (C_CUSTKEY:INTEGER, C_NAME:TEXT, C_ADDRESS:TEXT, C_NATIONKEY:INTEGER,
        C_PHONE:TEXT, C_ACCTBAL:DECIMAL, C_MKTSEGMENT:TEXT, C_COMMENT:TEXT)

ORDERS
        (O_ORDERKEY:INTEGER, O_CUSTKEY:INTEGER, O_ORDERSTATUS:TEXT,
        O_TOTALPRICE:DECIMAL, O_ORDERDATE:DATE, O_ORDERPRIORITY:TEXT,
        O_CLERK:TEXT, O_SHIPPRIORITY:INTEGER, O_COMMENT:TEXT)

LINEITEM
        (L_ORDERKEY:INTEGER, L_PARTKEY:INTEGER, L_SUPPKEY:INTEGER,
        L_LINENUMBER:INTEGER, L_QUANTITY:DECIMAL, L_EXTENDEDPRICE:DECIMAL,
        L_DISCOUNT:DECIMAL, L_TAX:DECIMAL, L_RETURNFLAG:TEXT, L_LINESTATUS:TEXT,
        L_SHIPDATE:DATE, L_COMMITDATE:DATE, L_RECEIPTDATE:DATE,
        L_SHIPINSTRUCT:TEXT, L_SHIPMODE:TEXT, L_COMMENT:TEXT)
```

Listing 4.31: TPC-H database

standard TPC-provided SQL. As a result, we used the exact set of SQL statements included in the TPC-H specification for System-X (with proper values for its substitution parameters determined according to each test's SF). As HiveQL does not fully support all the features in SQL, the original TPC-H queries in SQL needed to be modified for it. The developers of Hive have already published a modified set of queries in HiveQL [20], but an issue with this set is that it is now fairly old and does not consider some of the latest features in Hive (such as the now available support for nested sub-queries). For this reason, the authors of [64] revisited these queries and re-wrote 11 of them to optimize them further. The authors of [64] kindly shared their revised queries with us and we have used this set in our experiments for both Hive and Spark SQL. Finally, AsterixDB currently supports AQL as its only query language, so we translated the original TPC-H queries into AQL. The full set of TPC-H queries in AQL are available at [27] and included as the Appendix of this thesis.

Table	Primary Key Column(s)
NATION	N_NATIONKEY
REGION	R_REGIONKEY
PART	P_PARTKEY
SUPPLIER	S_SUPPKEY
PARTSUPP	PS_PARTKEY, PS_SUPPKEY
CUSTOMER	C_CUSTKEY
ORDERS	O_ORDERKEY
LINEITEM	L_ORDERKEY, L_LINENUMBER

Table 4.1: TPC-H schema, Primary keys

4.5.3 TPC-H Auxiliary Index Structures

The TPC-H benchmark specification includes a set of rules for primary/foreign key constraints and indexing. Table 4.1 lists the column(s) used as the primary key when creating the TPC-H tables. Listing 4.32 shows the referential constraints that are captured by DBGen during data generation. According to the benchmark rules, it is allowable to create auxiliary index structures on primary/foreign key attributes. Moreover, indices on single columns of the DATE datatype are also allowed. Listing 4.33 shows the columns that we indexed in our tests in those systems that include indexing functionality.

```
NATION: N_REGIONKEY (referencing R_REGIONKEY);
PARTSUPP: PS_PARTKEY (referencing P_PARTKEY);
PARTSUPP: PS_SUPPKEY (referencing S_SUPPKEY);
CUSTOMER: C_NATIONKEY (referencing N_NATIONKEY);
ORDERS: O_CUSTKEY (referencing C_CUSTKEY);
LINEITEM: L_ORDERKEY (referencing O_ORDERKEY);
LINEITEM: L_PARTKEY (referencing P_PARTKEY);
LINEITEM: L_SUPPKEY (referencing S_SUPPKEY);
LINEITEM: L_PARTKEY, L_SUPPKEY (referencing PS_PARTKEY, PS_SUPPKEY);
```

Listing 4.32: TPC-H referential constraints (foreign key)

```
ORDERS: O_ORDERDATE
LINEITEM: L_SHIPDATE
LINEITEM: L_RECEIPTDATE
```

Listing 4.33: TPC-H auxiliary index structures

4.6 Experiments

In this section, we first discuss the details of the experiments and describe the details of the hardware used and the settings used in different systems. We then present the performance results that we obtained in our experiments.

4.6.1 Experimental Setup

Cluster: We used a 10-node IBM x3650 cluster with a Gigabit Ethernet switch. Each node had one Intel Xeon processor E5520 2.26GHz (4 cores), 12 GB of RAM, and four 300GB, 10K RPM hard disks. The machines were running 64-bit CentOS 6.6 as their operating system. Three disks on each machine were used as persistent storage. The fourth disk was dedicated to storing the transaction and systems logs.

Client: Our client was running on a separate machine (with 16GB of RAM and 4 CPU cores) and was connected to the tests' cluster via the same switch. Using a closed-system model, we measured the average end-to-end response time of each individual query from the client's perspective to use as the performance metric. In each test, we ran the full set of 22 TPC-H queries sequentially, one after the other, three times per system/settings, and we discarded first run as the warm-up run. We report the average for the other two runs on the warmed-up cache as the response time for each query.

HDFS: We used Apache Hadoop version 2.6.0 with a heap size of 8GB and a replication factor of 1. Each node in the cluster was running the DataNode daemon for HDFS and Yarn's NodeManager daemon (if needed). The NameNode and ResourceManager daemons were running on a separate machine with a similar configuration as the cluster nodes.

Hive: We used Apache Hive stable version 1.2.0. For Tez, we built and used the stable version 0.7.0, using the code from the Apache repository. We enabled optimizations in Hive such as predicate push-down and map-side joins.

Spark SQL: We used Apache Spark version 1.5.0 with the Kryo serializer [13]. Each worker was running four executors (one per core) with 2GB of memory. Our queries were run from a Spark application written using Spark's Java API and Spark SQL's HiveContext. The driver program for running our Spark benchmarking application was run on a separate machine with 6GB of memory. Our application used Hive's metastore to access the schema and data for the tables created on files in HDFS.

AsterixDB: We used AsterixDB version 0.8.7 with one NC running on each node with 3 partitions. We set a maximum of 8GB of memory per NC and 2GB of buffer cache size. The HTTP API in AsterixDB was used to submit the AQL queries to the system.

System-X: For System-X, we used a version that was released in 2013 by its vendor. Each node in the cluster was serving three database partitions. The memory manager provided in System-X was responsible for tuning its resource allocations. A JDBC driver, provided by the vendor, was used to submit the queries to the system.

Data and storage: We conducted our experiments with a focus on the scale-up characteristics of the systems. For this purpose, we considered three system scales, 9, 18, and 27 partitions running on 3, 6, and 9 machines respectively. We placed 50GB of data on each machine, which is almost five times their available memory. We used the DBGen program to generate a total of 150GB, 300GB, and 450GB of data for these different system scales.

For Hive, we used two types of tables: external tables created on raw text data files and ORC

format tables. In Spark SQL we considered both external and Parquet tables. For the ORC and Parquet cases, the CUSTOMER and SUPPLIER tables were created as partitioned tables using the NATION_KEY column as the partitioning attribute (there are total of 25 possible values for NATION_KEY). For loading the ORC and Parquet tables, we used Hive's insert statement in HiveQL to read the data from the external text tables and add it to the target ORC or Parquet tables. We loaded the partitioned tables partition by partition to reduce the high memory usage overhead of partitioning data dynamically during the loading.

As AsterixDB has support for both internal and external datasets [35], we considered both options there as well. The external datasets were created using the exact same files in HDFS as used for the external text tables in Hive and Spark SQL. For creating the internal (managed) datasets, we included all of the attributes in the DDL statements. Internal datasets were hash-partitioned based on the primary keys shown in Table 4.1. Moreover, we created secondary indices on the attributes listed in Listing 4.32 and 4.33 for both the external and internal datasets. For loading the internal datasets we used AsterixDB's native bulk load statement.

In System-X, we created hash-partitioned tables using the primary key(s) in each table. Similar to AsterixDB, we also created the secondary indices of Listing 4.32 and 4.33. We also included the definition of referential constraints (PK/FK relationships) in our DDL statements. We used the load utility provided in System-X to bulk-load the data and populate the tables. After loading each scale factor, we ran the statistics gathering scripts in System-X to provide its cost-based optimizer with the information required for generating effective query plans.

4.6.2 Experimental Results

In this section, we review the experimental results that we obtained from running all systems on the three different scales.

Loading

Tables 4.2, 4.3, and 4.4 show the original size of the data generated per table in each scale as well as the total size of each table after loading based on the target system and/or format. As one can see, there are significant storage size differences between the different formats and systems. As an example, in Table 4.4 (for SF=450) one can see that, because of its built-in compression, ORC is the most efficient format in terms of storage size. This is consistent with what we saw in Chapter 3 as well. Parquet also shows storage size advantages for a similar reason i.e., the use of columnar storage and compression. These two formats use different compression techniques which is why they differ from one another in terms of their total size. In AsterixDB, we could load the data in parallel and across all partitions, while the load utility in System-X required us to load each hash partitioned table in the database partition by partition. AsterixDB (with internal datasets) and System-X use their own managed storage layer to store the data. Each of these systems has its own storage format and loads the data into a variant of B-Tree and organizes the records in pages with enough information stored per page for future access. This comes with additional costs in terms of storage space compared to the raw data format.

Table 4.5 shows the total amount of time used for loading each storage format per scale. For each scale factor, we report the total amount of time that it took to load all 8 TPC-H tables and prepare the database (e.g., adding indices and gathering statistics). You can see that the loading times for ORC and Parquet do not change a lot as the data scales up. It is due to the fact that the transformation of records from text to ORC or Parquet

	Nation	Region	Part	Supplier	Partsupp	Customer	Orders	Lineitem
Text	2KB	389 Bytes	3.68	0.21	18.38	3.7	26.75	119.92
ORC	1.75 KB	1 KB	0.59	0.08	4.63	1.23	6.3	26.91
Parquet	3.2 KB	1.1 KB	1.91	0.22	16.8	3.57	20.16	51.93
AsterixDB LSM	3.45 MB	1.92 MB	5.45	0.3	23.40	4.89	38.7	207
System-X	18 MB	18 MB	4	0.26	19	4	28	137

Table 4.2: TPC-H tables size (in GB) - SF 150

	Nation	Region	Part	Supplier	Partsupp	Customer	Orders	Lineitem
Text	2 KB	389 Bytes	7.36	0.42	36.7	7.4	53.5	239.84
ORC	1.75 KB	1 KB	1.18	0.16	9.25	2.46	12.63	53.91
Parquet	3.2 KB	1.1 KB	3.81	0.44	33.6	7.17	40.16	103.26
AsterixDB LSM	6.91 MB	1.92 MB	10.91	0.61	46.8	9.79	77.4	414
System-X	36 MB	36 MB	8.33	0.52	38	8	56.67	274.33

Table 4.3: TPC-H tables size (in GB) - SF 300

is happening as a map-only job per table, where each mapper reads its input data (from HDFS) and writes its output data (to HDFS) locally. These jobs are running in parallel, therefore adding more partitions does not add too much overhead. As we load System-X partition by partition, the loading time increases proportional to the increase in the number of partitions. In AsterixDB's case, the increase in the loading time is under investigation.

	Nation	Region	Part	Supplier	Partsupp	Customer	Orders	Lineitem
Text	2 KB	389 Bytes	11.07	0.64	55.4	11.12	80.97	363.67
ORC	1.75 KB	1 KB	1.77	0.23	13.86	3.68	19	81
Parquet	3.2 KB	1.1 KB	5.7	0.67	50.4	10.8	60	154
AsterixDB LSM	9.6 MB	1.92 MB	16.39	0.92	70.2	14.71	116.1	621
System-X	54 MB	54 MB	13	0.79	57	12	86	412

Table 4.4: TPC-H tables size (in GB) - SF 450

	9 Partitions	18 Partitions	27 Partitions
ORC	3,055	3,187	3,318
Parquet	3,535	3,728	3,920
AsterixDB LSM	2,838	5,843	9,014
System-X	8,608	17,057	25,345

Table 4.5: TPC-H tables loading time (in sec)

Failed Queries

There were a couple of cases where we could not obtain a stable time from certain queries for a given system, mainly as the result of a failure:

In Spark, Q11 took a very long time (on the order of hours) within an aggregation step at the beginning of the last stage of the query for both the text and Parquet formats at all scales, and we had to kill that query. Q18 was another problematic query in Spark. As the result of a failure in allocating the required memory, Q18 failed for both the text and Parquet formats at all scales.[1] Q21 was the third query with issues in Spark. While it worked fine in the 9-partition case, this query failed for the other scales for a similar reason as Q18, i.e., a failure in memory allocation. One last query with minor issues in Spark was Q16. The parser in Spark SQL failed to parse the modified and optimized version of this query in HiveQL (provided by authors of [64]) because of a lack of parsing rules. We had to revert this query to an older version which did not use the latest sub-query support in HiveQL in order to run it successfully.

In Hive, we encountered one query with issues. Q4 did not complete while running on Tez with the ORC tables. A Yarn container was killed by the system after running this query for a long time resulting in a failed vertex exception during the execution of the query's DAG.

[1] An open issue for a similar failure exists in the project's JIRA for the stable version of the software that we used [22]

Query Times

Tables 4.6, 4.7, and 4.8 show the full set of results that we obtained on all three scales. (The cells for the failed queries are marked with "-".) As one can see, no system or storage format was found to provide the best performance for all of the queries. In addition, the different systems showed different scale-up behavior. Figures 4.1 to 4.9 show how the response times for each system changed as we scaled up the benchmark. The 9-partition case is considered as the base response time in these figures, and we then show the ratios of the times for the 18-partition and 27-partition cases relative to the base case.

In terms of overall scale-up, AsterixDB shows reasonable behavior for both its external and internal datasets. For most queries, the response times remain more or less the same as the data and number of partitions grow. In terms of performance, internal datasets in AsterixDB show better response times as they are stored in AsterixDB's binary data format (ADM) and they do not need any translation when being read. Moreover, the I/O requests against internal datasets are done in a single-threaded fashion per I/O device by AstreixDB, while in HDFS multiple threads may access different splits of data simultaneously and with interfere at the I/O device level which can have a negative impact on the performance [35]. In Spark SQL, we can see that the response time grows as the data grows in the text data case. With Parquet tables, however, Spark SQL shows better scale-up behavior. One major reason for this which we observed during the tests, was related to differences in the way that data is read for the two formats. In the Parquet case we could see parallel reads happening among all workers, while the text data case suffered from the lack of enough parallelism in scheduling and doing I/Os among workers. For most of the problematic cases, we could see that all workers except one were idle, with that one being busy reading data from the file system. Changing the number of executors and workers did not really affect this behavior. For Hive, we can see that Hive on Tez shows the best scale-up behavior. System-X also showed proper scale-up behavior for most of the TPC-H queries. There are two queries in System-X which

show strange behavior: Q10 shows a non-linear performance change between different scales, and Q22's response time grows as the total data size grows. We will look at these two cases in more detail in the next Section.

In terms of the storage formats, all of the queries in Spark SQL benefit as we switch from the text format to the Parquet file format. Besides its smaller size, one other reason for it is the way that I/O requests were handled in the Parquet case. As explained above, we could see better I/O parallelism with Parquet. The ORC format in Hive also showed a performance gain for most of the queries versus the text format. This is an expected behavior because of the inherent optimizations in ORC.

In terms of Hive's runtime, most of the queries benefited from Tez, especially those involving multiway joins. The small job-starting overhead and skipping of intermediate materialization steps in Tez as compared to MapReduce are the main reason for that. The combination of Tez with ORC also showed better behavior in terms of utilizing the Map-side join optimization in Hive. There were several queries that used Map-sides join only with this combination (e.g., Q2 and Q5).

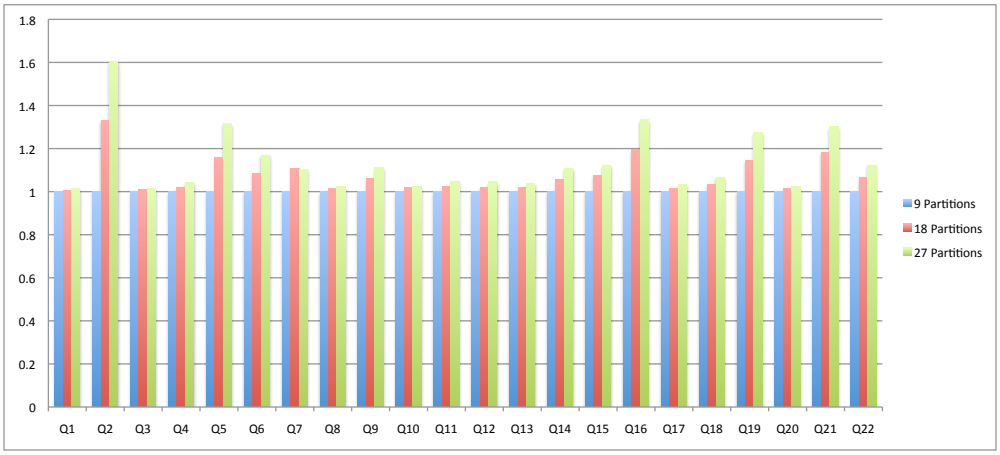

Figure 4.1: AsterixDB scale-up on external datasets

	Asterix *External*	Asterix *Internal*	SparkSQL *Text*	SparkSQL *Parquet*	Hive *MR Text*	Hive *MR ORC*	Hive *Tez Text*	Hive *Tez ORC*	System-X
Q1	879	657	827	154	508	315	465	273	270
Q2	164	123	162	73	352	332	173	141	56
Q3	815	524	983	263	957	695	758	489	417
Q4	673	357	842	114	1137	754	500	-	409
Q5	840	510	1212	500	1613	1155	1036	891	421
Q6	248	376	695	82	219	84	218	90	259
Q7	1066	856	1256	385	3145	2799	1372	1044	590
Q8	936	688	1196	358	1522	1023	1134	1061	375
Q9	2585	1988	1929	1057	4224	3607	4552	3523	587
Q10	704	408	901	168	832	532	586	359	1604
Q11	145	72	-	-	820	534	227	228	41
Q12	479	428	877	123	509	315	473	265	320
Q13	652	260	244	133	404	426	292	280	99
Q14	269	234	687	86	310	166	269	142	442
Q15	266	245	1366	165	614	360	469	186	319
Q16	180	147	158	44	353	278	180	147	33
Q17	1548	1330	1711	445	2246	1582	1811	1716	269
Q18	1044	788	-	-	2396	2018	1262	1702	375
Q19	621	428	827	158	1729	1302	2141	1394	265
Q20	506	431	823	137	666	474	481	319	2316
Q21	3069	2634	2575	725	2900	2093	2184	1649	8776
Q22	333	322	181	42	335	337	201	203	336

Table 4.6: TPC-H queries response time (in sec) - SF 150

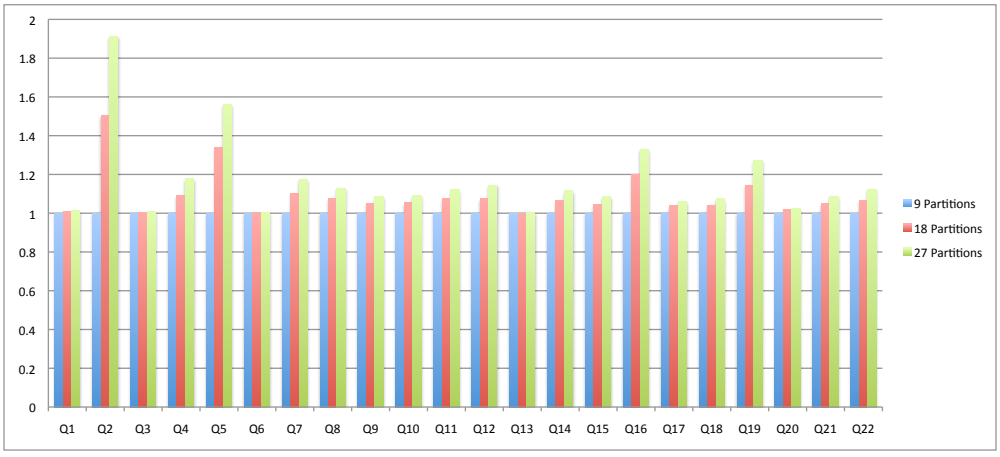

Figure 4.2: AsterixDB scale-up on internal datasets

	Asterix *External*	Asterix *Internal*	SparkSQL *Text*	SparkSQL *Parquet*	Hive *MR Text*	Hive *MR ORC*	Hive *Tez Text*	Hive *Tez ORC*	System-X
Q1	885	663	1863	162	510	320	467	277	277
Q2	218	186	246	76	357	343	186	149	58
Q3	822	525	2381	491	1002	705	790	518	429
Q4	686	390	2042	118	1148	756	650	-	415
Q5	973	683	2859	930	1622	1290	1081	982	433
Q6	269	378	1742	86	224	91	236	93	262
Q7	1181	945	2804	635	3351	3041	1473	1127	698
Q8	948	742	2396	388	1565	1112	1169	1089	452
Q9	2742	2092	3354	1152	4572	3925	4781	3629	595
Q10	718	429	2231	298	909	534	712	378	1618
Q11	148	78	-	-	916	781	245	246	42
Q12	489	461	2166	220	626	325	574	273	340
Q13	665	260	422	141	577	453	348	292	116
Q14	284	249	1697	92	316	177	277	144	446
Q15	286	256	3397	189	644	360	489	189	328
Q16	215	176	225	69	387	334	187	150	44
Q17	1572	1380	3767	475	2307	1642	1963	1757	271
Q18	1079	821	-	-	2445	2060	1320	1751	386
Q19	711	490	1896	179	1770	1324	2154	1489	269
Q20	512	439	1960	140	711	531	524	331	2914
Q21	3620	2762	-	-	2957	2274	2387	1845	11102
Q22	355	343	349	45	357	357	216	210	868

Table 4.7: TPC-H queries response time (in sec) - SF 300

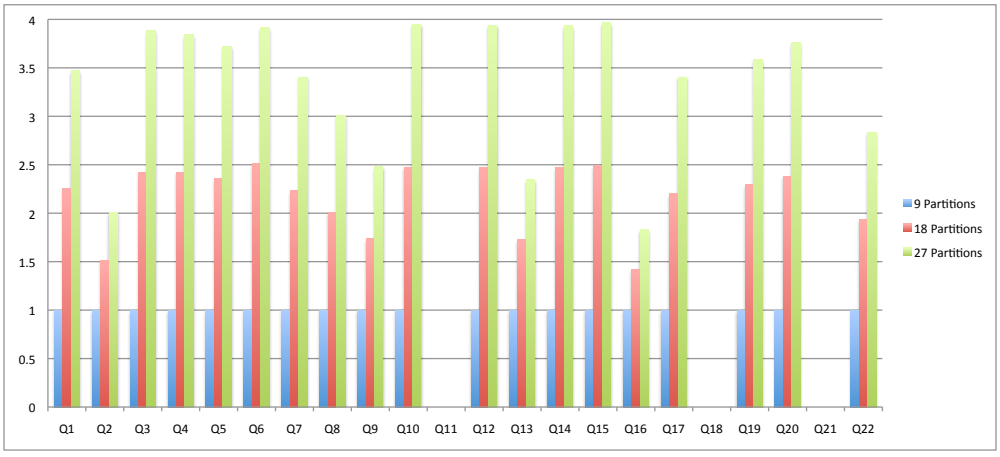

Figure 4.3: Spark SQL scale-up on the text format

	Asterix *External*	Asterix *Internal*	SparkSQL *Text*	SparkSQL *Parquet*	Hive *MR Text*	Hive *MR ORC*	Hive *Tez Text*	Hive *Tez ORC*	System-X
Q1	889	665	2870	168	511	324	471	283	286
Q2	263	236	326	80	362	351	196	152	60
Q3	828	528	3825	709	1046	714	821	537	436
Q4	703	421	3236	121	1155	759	794	-	424
Q5	1105	795	4512	1359	1628	1385	1114	1082	442
Q6	290	378	2717	90	229	96	252	98	265
Q7	1176	1006	4272	879	3507	3261	1569	1218	796
Q8	959	776	3599	417	1616	1190	1188	1104	524
Q9	2878	2156	4785	1260	4844	4423	5007	3745	601
Q10	723	445	3558	417	983	535	832	389	455
Q11	152	81	-	-	1006	1025	268	266	42
Q12	501	490	3451	313	738	339	680	285	355
Q13	677	261	573	149	741	474	400	300	125
Q14	298	261	2702	97	321	186	284	146	454
Q15	298	265	5419	210	673	360	514	193	334
Q16	240	195	290	90	416	385	198	154	51
Q17	1602	1411	5827	504	2372	1695	2104	1784	272
Q18	1114	846	-	-	2516	2105	1372	1792	399
Q19	790	545	2962	201	1809	1341	2165	1573	273
Q20	517	442	3097	141	752	575	559	346	3535
Q21	3990	2862	-	-	3013	2417	2577	2016	12986
Q22	374	362	513	47	377	368	226	213	1393

Table 4.8: TPC-H queries response time (in sec) - SF 450

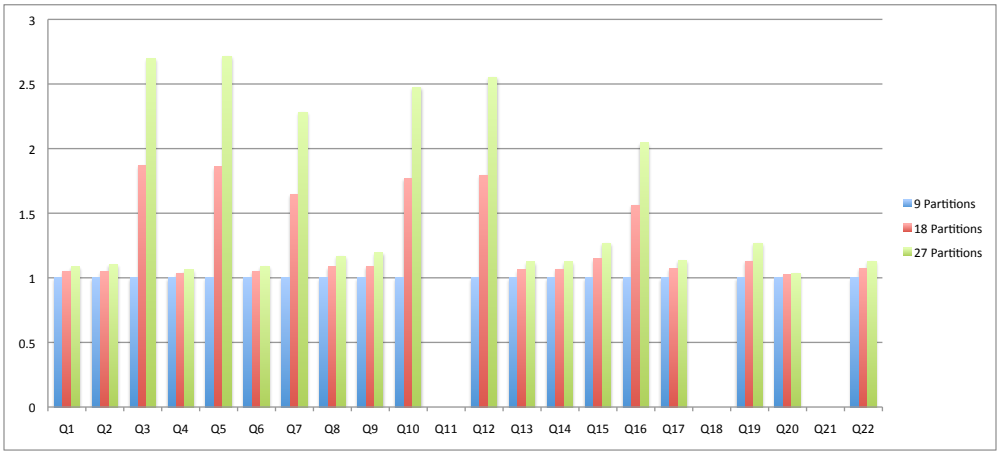

Figure 4.4: Spark SQL scale-up on the Parquet format

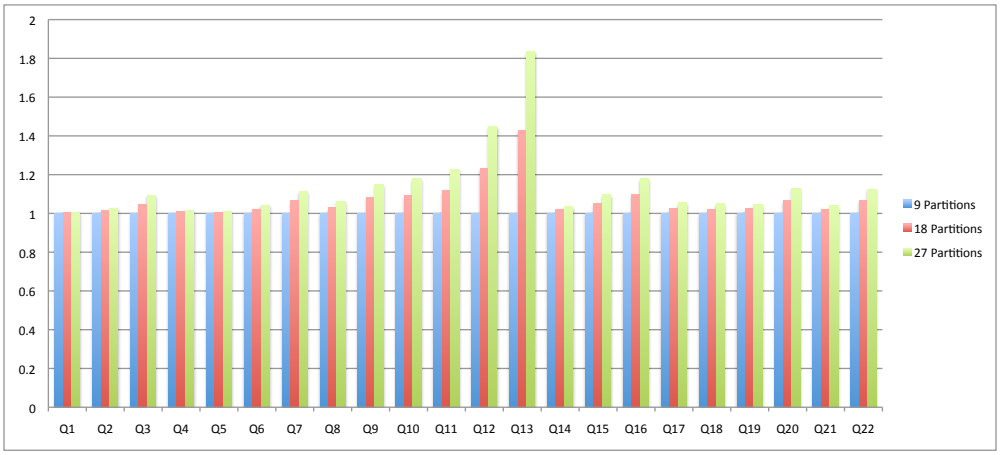

Figure 4.5: Hive scale-up on MR and the text format

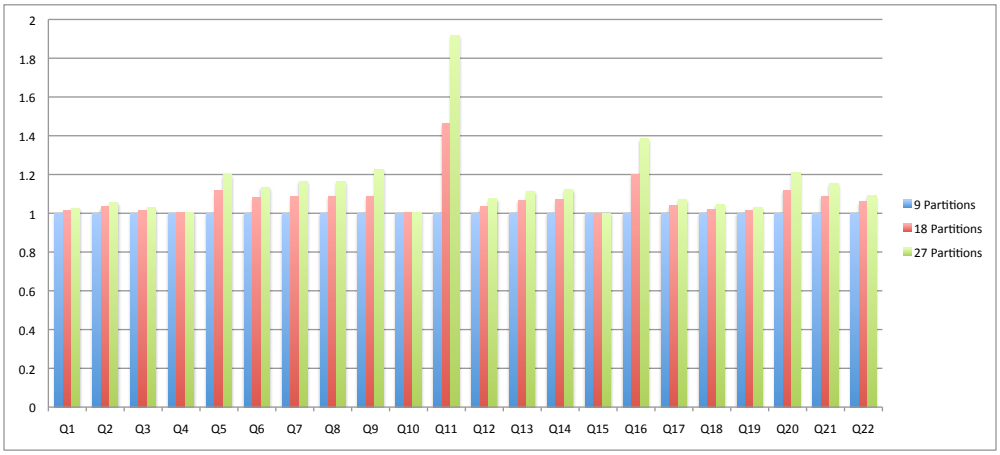

Figure 4.6: Hive scale-up on MR and the ORC format

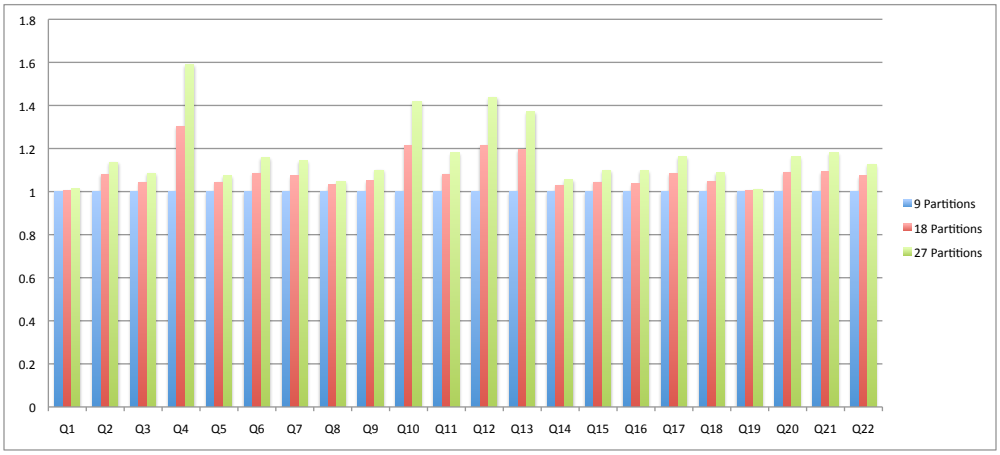

Figure 4.7: Hive scale-up on Tez and the text format

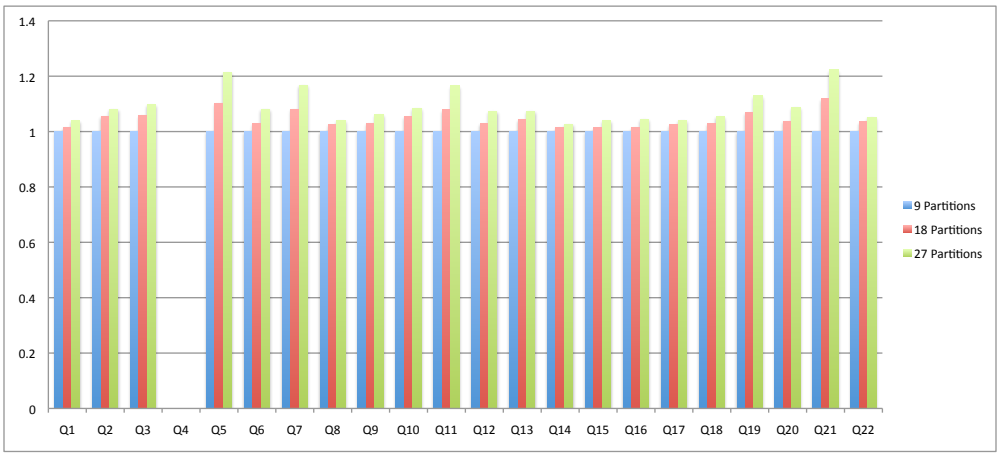

Figure 4.8: Hive scale-up on Tez and the ORC format

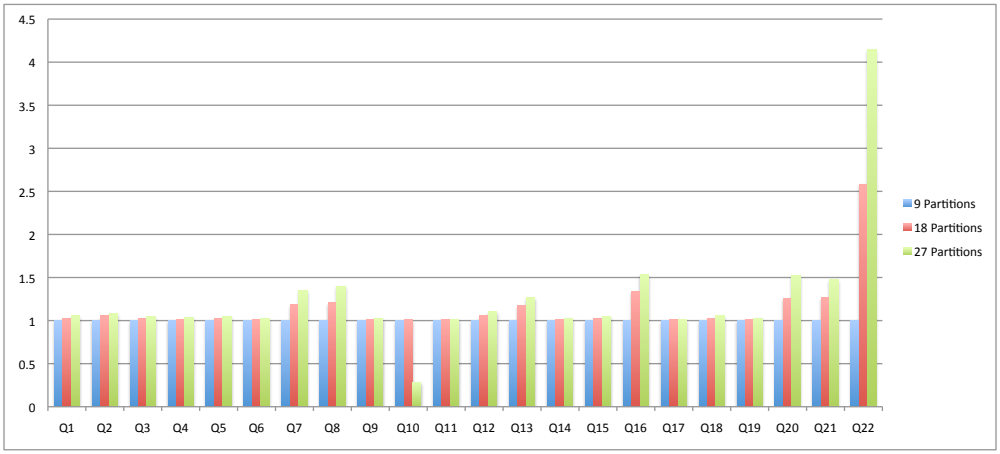

Figure 4.9: System-X scale-up

4.7 Selected queries

In this section, we examine a subset of the queries in greater detail. The main reason to choose these particular queries is that either the performance of the different systems differed significantly for them, or a specific system showed a different scale-up behavior as compared to the other queries.

4.7.1 Query 1

Query 1, shown in Listing 4.34, provides a summary pricing report for all lineitems shipped as of a given date [26]. It only accesses one, but the largest, table in the database, LINEITEM. From the runtime perspective, this is in fact a query that can check how fast a system can perform filtered scans on a large table. The storage size and format, and the way that selection predicate is evaluated on tuples, are the key factors in the performance of a system for this query.

As Table 4.8 shows for the largest scale (SF=450), Spark SQL was able to achieve the best time among all systems when using Parquet. Interestingly, Spark SQL showed the worst performance when running on the text data. The execution of this query in Spark consists of two stages. During the first one, the LINEITEM table is scanned with the selection predicate and a local aggregation is performed. After an exchange, the global aggregation is computed during the second stage. The Spark SQL runtime plan is the same for both the text and Parquet cases. As mentioned before, we observed a much better I/O parallelization when the Parquet format was being used. Similar to Spark's stages, Hive uses two MapReduce jobs to process this query. Hive on Tez with the ORC format shows the best performance for Hive. It is mainly due to the ORC format's advantages along with Hive's efficient data parsing strategy, which is able to skip copying and parsing whole records by just fetching

the relevant columns.

In both AsterixDB and System-X, we have a secondary index on the column used in the query's predicate, i.e., L_SHIPDATE. The selectivity of this predicate is large. In the SF=450 case, it selects more than 887 million tuples out of almost 2.7 billion LINEITEM tuples. As a result, exploiting this index will not help for this query. System-X's cost-based optimizer decides to fully scan the LINEITEM table for this reason. In AsterixDB the "skip-index" hint is available for this purpose, and it was used for the times we reported. The combination of values in L_RETURNFLAG and L_LINESTATUS, which are the grouping columns, has only 4 possible values. In AsterixDB we used hash-based aggregation (enabled with a hint) for this query which resulted in better performance as compared to sort-based aggregation. (This reduced the response time for SF=450 by about 30%). However, System-X's optimizer decided to use sorting for grouping and aggregation here. For a full scan of the LINEITEM table, AsterixDB and System-X are comparable. (For SF=450, scanning took 270 seconds for AsterixDB and 250 seconds for System-X).

```
SELECT L_RETURNFLAG, L_LINESTATUS,
        SUM(L_QUANTITY) AS SUM_QTY,
        SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
        SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
        SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
        AVG(L_QUANTITY) AS AVG_QTY,
        AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
        AVG(L_DISCOUNT) AS AVG_DISC,
        COUNT(*) AS COUNT_ORDER
FROM
        LINEITEM
WHERE
L_SHIPDATE <= DATE('1998-12-01') - 90 DAY
GROUP BY L_RETURNFLAG,
L_LINESTATUS ORDER BY
L_RETURNFLAG, L_LINESTATUS;
```

Listing 4.34: TPC-H Query 1

4.7.2 Query 10

This query finds the top 20 customers in terms of their effect on lost revenue for a given quarter [26]. The query is shown in Listing 4.35 and involves a 4-way join between the CUS-

TOMER, ORDERS, LINEITEM, and NATION tables, for which an optimizer has different join ordering and join strategy choices to choose from. Spark SQL first joins CUSTOMER with filtered ORDERS tuples and then joins these results with filtered LINEITEM tuples, finally joining the results with the NATION table. AsterixDB joins CUSTOMER and filtered ORDERS tuples first, and the available secondary index on O_ORDERDATE is used for applying the predicate on ORDERS. This join is followed by joining its results with the NATION table. Finally the results are joined with the filtered LINEITEM tuples. In terms of the join strategy, AsterixDB currently uses hybrid hash joins for the equi-joins. For this query, Spark SQL uses sort merge joins for the first two joins, and for the join involving the NATION table, it broadcasts this table and performs a hybrid hash join. AsterixDB shows a better scale-up for this query, and the performance of Spark SQL with Parquet and AsterixDB are comparable for SF=450. Hive shows its best performance for this query when it runs on Tez with ORC. With this setting and by using 4 map and 4 reduce steps, Hive follows a similar strategy as Spark SQL by first joining CUSTOMER and ORDERS using sort merge join and then replicating NATION and doing a broadcast hash join with LINEITEM. The performance of these similar plans in Hive and Spark SQL is comparable for SF=450 as well.

The main reason we found this query interesting is that System-X scaled up non-linearly for the 27-partition case. This turns out to be because System-X changed the query plan that it used when dealing with this case (SF=450). Figure 4.10 shows its query plan for 9 partitions, and Figure 4.11 shows the plan for 27 partitions. In both cases, System-X first joins filtered ORDERS and LINEITEM tuples. It exploits the secondary indices on O_ORDERDATE and L_ORDERKEY for filterng and joining. For the smaller scales, System-X proceeds by scanning the CUSTOMER table and joining it with the NATION table, and as the output is already sorted on customer keys, the system chooses a merge join to do the join between the results of the CUSTOMER and the NATION tables join and the results of the first join (ORDERS and LINEITEM). In SF=450, however, System-X decides to broadcast the

NATION table (which consists of only 25 tuples) across all partitions and to do a hash join at the end. As shown in Figure 4.9, the response time of the query changes differently between SF=150 and SF=450 as compared to SF=150 and SF=300.

```
SELECT C_CUSTKEY, C_NAME,
        SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE,
        C_ACCTBAL, N_NAME, C_ADDRESS, C_PHONE, C_COMMENT
FROM
CUSTOMER, ORDERS, LINEITEM, NATION
WHERE
C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND O_ORDERDATE >= DATE('1993-10-01')
AND O_ORDERDATE < DATE('1993-10-01') + 3 MONTH
AND L_RETURNFLAG = 'R'
AND C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE, N_NAME, C_ADDRESS, C_COMMENT
ORDER BY
REVENUE DESC
LIMIT 20;
```

Listing 4.35: TPC-H Query 10

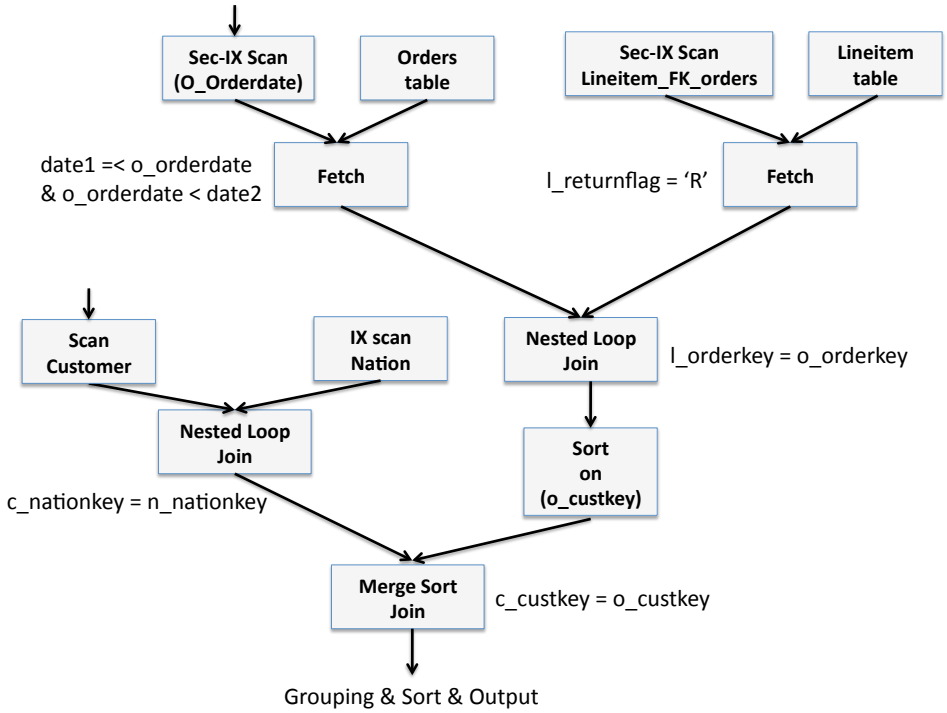

Figure 4.10: Query 10 plan - System-X on 9 partitions

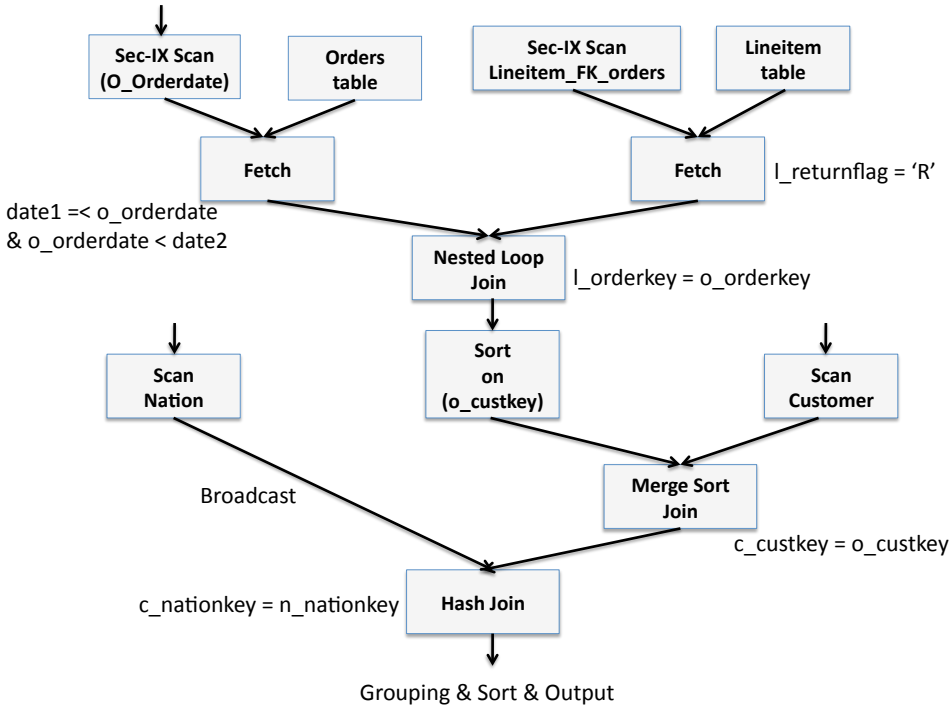

Figure 4.11: Query 10 plan - System-X on 27 partitions

4.7.3 Query 19

This query finds the gross discounted revenue for all orders that were shipped by air and delivered in person for three different types of parts [26]. As shown in Listing 4.36, the query joins the PART and LINEITEM tables on their PK/FK columns and has complex disjunctive and conjunctive predicates on both tables. Looking carefully, one can see that the predicates on L_SHIPMODE and L_SHIPINSTRUCT from LINEITEM are common among all three disjuncts. These predicates in fact are quite selective (in SF=450, less than 97 million tuples pass them out of more than 2.7 billion total LINITEM tuples). As a result, pushing these predicates down and applying them prior to the join step can help a system's overall performance significantly.

Figure 4.12 shows the optimized plan that Spark SQL generates for this query. Spark SQL indeed extracts the common predicate and applies it prior to the join step. It also extracts the common lower bound of the range filter on P_SIZE and pushes it down, but as all the PART

tuples satisfy this lower bound that is of no help in this case. The remaining case-specific (P_BRAND related) predicates are applied to the results of the join, which are then fed to the aggregate step. The plan that System-X generates for this query is similar to Spark's plan. The main difference is in which predicates it pushes down. System-X decides to apply all of the predicates (both the common and case-specific ones) for both the LINEITEM and PART tables while scanning them. The case-specific predicates are grouped and are first applied disjunctively and then applied again in a case-specific manner on the join results. In this way, System-X tries to reduce the input sizes for the join step. For example, in the SF=450 case, about 58 million (out of more than 2.7 billion) LINEITEM tuples and about 216,000 (out of 90 million) PART tuples pass these filters. Unlike Spark SQL and System-X, Hive does not extract the common predicate on LINEITEM from the disjuncts to push it down. Running on Tez, Hive first uses two maps to fully scan both tables and sorts them on the PARTKEY columns. Then two reducers are used to perform a merge join, grouping and aggregation. The negative impact of joining a large number of unneeded tuples is the main reason that Hive's response times for all settings are significantly higher than the others.

In AsterixDB, the optimizer did not factor out the common predicates and push them down, when the AQL query was based on a straightforward translation from SQL to AQL. The system decided to first fully join LINEITEM and PART, and since the attribute from PART used in the join predicate (P_PARTKEY = L_PARTKEY) is PART's primary key, LINEITEM is hash-exchanged. This decision results in a significant overhead in data movement, which impacts AsterixDB's performance negatively. A more careful translation of the query into AQL (shown in Listing 4.37) removes this overhead by helping the system to push down the common predicates; this improves performance by a factor of 3 for the SF=450 case. The optimized plan for this query in AsterixDB is shown in Figure 4.13 and the numbers in Tables 4.6 to 4.8 are reported based on that.

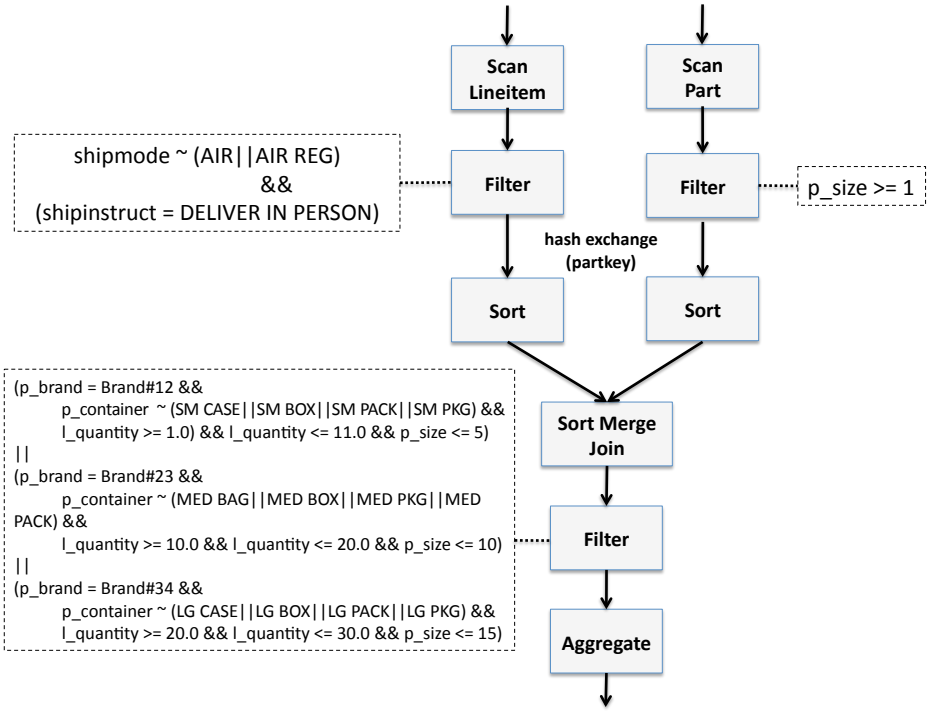

Figure 4.12: Query 19 plan - Spark SQL

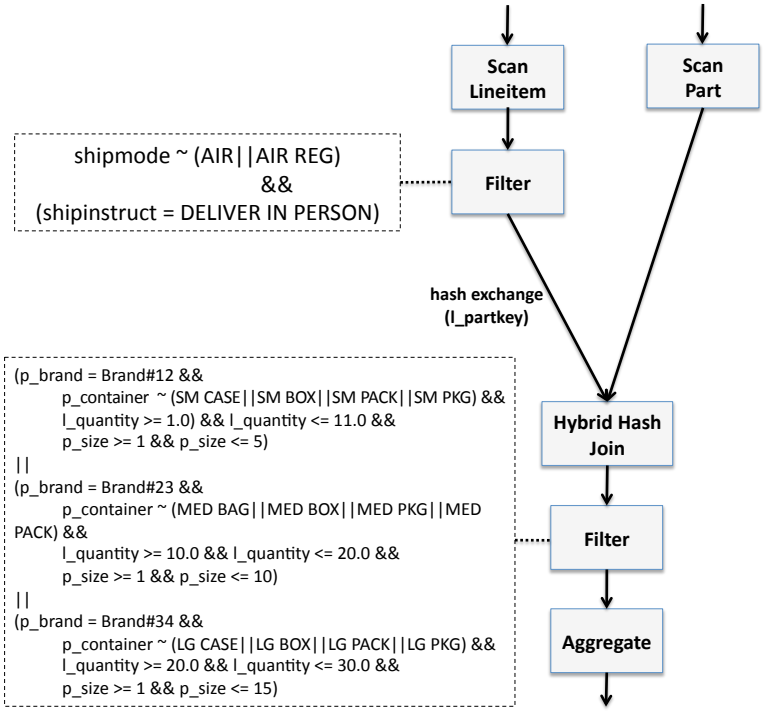

Figure 4.13: Query 19 plan - AsterixDB

```
SELECT
SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT) ) AS REVENUE
FROM
LINEITEM, PART
WHERE
(
        P_PARTKEY = L_PARTKEY
        AND P_BRAND = 'BRAND#12'
        AND P_CONTAINER IN ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        AND L_QUANTITY >= 1 AND L_QUANTITY <= 1 + 10
        AND P_SIZE BETWEEN 1 AND 5
        AND L_SHIPMODE IN ('AIR', 'AIR REG')
        AND L_SHIPINSTRUCT = 'DELIVER IN PERSON'
) OR
(
        P_PARTKEY = L_PARTKEY
        AND P_BRAND = 'BRAND#23'
        AND P_CONTAINER IN ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        AND L_QUANTITY >= 10 AND L_QUANTITY <= 10 + 10
        AND P_SIZE BETWEEN 1 AND 10
        AND L_SHIPMODE IN ('AIR', 'AIR REG')
        AND L_SHIPINSTRUCT = 'DELIVER IN PERSON'
) OR
(
        P_PARTKEY = L_PARTKEY
        AND P_BRAND = 'BRAND#34'
        AND P_CONTAINER IN ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        AND L_QUANTITY >= 20 AND L_QUANTITY <= 20 + 10
        AND P_SIZE BETWEEN 1 AND 15
        AND L_SHIPMODE IN ('AIR', 'AIR REG')
        AND L_SHIPINSTRUCT = 'DELIVER IN PERSON'
);
```

Listing 4.36: TPC-H Query 19 (in SQL)

4.7.4 Query 22

This query counts the number of customers within a specific range of country codes that have not placed orders but have a greater than average positive account balance [26]. This query is shown in Listing 4.38 and it is interesting for us for two reasons: Spark SQL on Parquet shows a very good performance for it, and System-X shows its worst scale-up behavior among all queries for this one. We show the query plan that each system uses in Figures 4.14 to 4.16. The part of the plan that calculates AVG(C_ACCTBAL) on the CUSTOMER table is the same among all three systems. The main performance difference between the systems is based on how the "greater-than" predicate (C_ACCTBAL > AVG(C_ACCTBAL)) and the "anti-join" part (CUSTOMER tuples with no order) of the query are evaluated. Spark (Figure 4.14) decides to use a sort-merge outer join to find customers with no order and then applies the greater-than predicate after doing a Cartesian product. Hive on Tez uses three

```
DECLARE FUNCTION Q19_TMP() {
  FOR $T IN DATASET LINEITEM
  WHERE
  ($T.L_SHIPMODE = 'AIR' OR $T.L_SHIPMODE = 'AIR REG')
  AND $T.L_SHIPINSTRUCT = 'DELIVER IN PERSON'
  RETURN {
    "LPKEY": $T.L_PARTKEY,
    "QUANTITY": $T.L_QUANTITY,
    "EXTNDPRICE": $T.L_EXTENDEDPRICE,
    "DISCOUNT": $T.L_DISCOUNT
  }
}

SUM(
  FOR $L IN Q19_TMP()
  FOR $P IN DATASET PART
  WHERE $P.P_PARTKEY = $L.LPKEY
    AND ( (
        $P.P_BRAND = 'BRAND#12'
        AND REG-EXP($P.P_CONTAINER,'SM CASE|SM BOX|SM PACK|SM PKG')
        AND $L.QUANTITY >= 1 AND $L.QUANTITY <= 11
        AND $P.P_SIZE >= 1 AND $P.P_SIZE <= 5
      ) OR (
        $P.P_BRAND = 'BRAND#23'
        AND REG-EXP($P.P_CONTAINER, 'MED BAG|MED BOX|MED PKG|MED PACK')
        AND $L.QUANTITY >= 10 AND $L.QUANTITY <= 20
        AND $P.P_SIZE >= 1 AND $P.P_SIZE <= 10
      ) OR (
        $P.P_BRAND = 'BRAND#34'
        AND REG-EXP($P.P_CONTAINER, 'LG CASE|LG BOX|LG PACK|LG PKG')
        AND $L.QUANTITY >= 20 AND $L.QUANTITY <= 30
        AND $P.P_SIZE >= 1 AND $P.P_SIZE <= 15
      )
    )
  RETURN $L.EXTNDPRICE * (1 - $L.DISCOUNT)
)
```

Listing 4.37: TPC-H Query 19 (in AQL)

map and four reduce steps to process this query. It first calculates the AVG value (using one map and one reduce) and broadcasts it. Another map step finds the qualified CUSTOMER tuples, and using a map and reduce step a group-by on ORDERS based on O_CUSTKEY is calculated. Then Hive performs an outer merge join on the qualified CUSTOMERS and the grouped ORDERS. The greater-than predicate is also applied in this step. The final aggregation happens on the output of this join and generates the results. System-X, however, decides to use the index on O_CUSTKEY on the ORDERS table to evaluate a sub-query for finding customers with no orders; the greater-than predicate evaluation happens in a subsequent nested loop join. As shown in Figure 4.16, AsterixDB first applies the less-than predicate and finds the customers with above the average balances by performing a nested loop join and then it evaluates the anti-join sub-query as the last step.

By breaking the query down into pieces and running each one, we realized that the anti-join (finding customers with no orders) takes the most time. While both Spark and AsterixDB use outer joins for this part, System-X chooses to use the secondary index on the foreign key column. In the SF=150 case, out of 22.5 million customers, more than 6.3 million of them pass the predicate on C_PHONE, and more than 2.1 million of these customers have no orders. Moving up to SF=450, we find a total of 67.5 million customers where almost 19 million of them pass the filter on C_PHONE. Out of these customers, almost 6.3 million have no orders. Therefore, as the data grows, the number of qualifying customers for this predicate also grows proportionally. Because of the access method that System-X uses to find them (index lookups rather than an outer join), it needs to do more and more random I/Os, proportional to the data size, as the cardinality scales up.

```
SELECT
  CNTRYCODE,
  COUNT(*) AS NUMCUST,
  SUM(C_ACCTBAL) AS TOTACCTBAL
FROM
  (
    SELECT
      SUBSTR(C_PHONE , 1 , 2) AS CNTRYCODE,
      C_ACCTBAL
    FROM CUSTOMER
    WHERE
      SUBSTR(C_PHONE,1,2) IN ('13', '31', '23', '29', '30', '18', '17') AND
      C_ACCTBAL >
      (
        SELECT AVG(C_ACCTBAL)
        FROM CUSTOMER
        WHERE
         C_ACCTBAL > 0.00 AND
         SUBSTR(C_PHONE,1,2) IN ('13', '31', '23', '29', '30', '18', '17')
      )
      AND NOT EXISTS
      (
        SELECT *
        FROM ORDERS
        WHERE
         O_CUSTKEY = C_CUSTKEY
      )
  )
GROUP BY
        CNTRYCODE
ORDER BY
        CNTRYCODE;
```

Listing 4.38: TPC-H Query 22

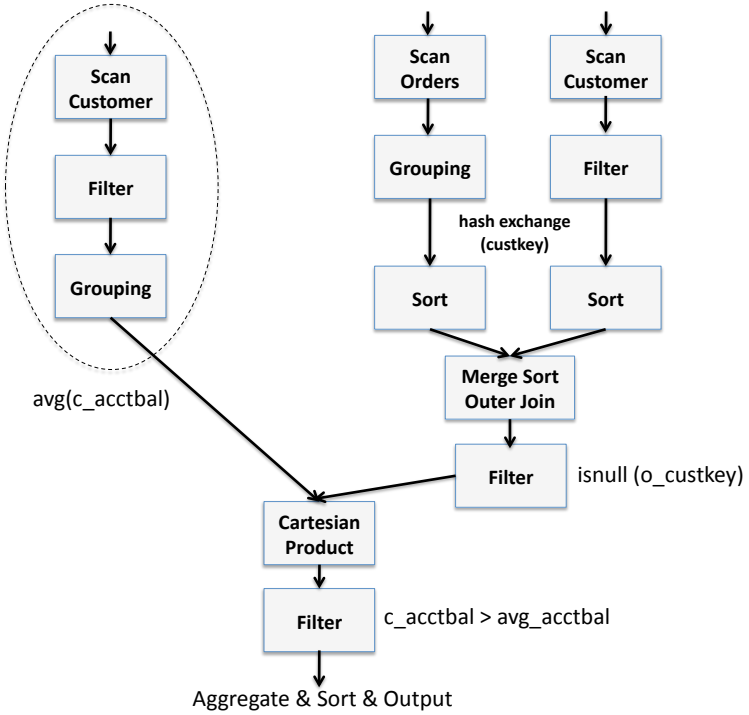

Figure 4.14: Query 22 plan - Spark

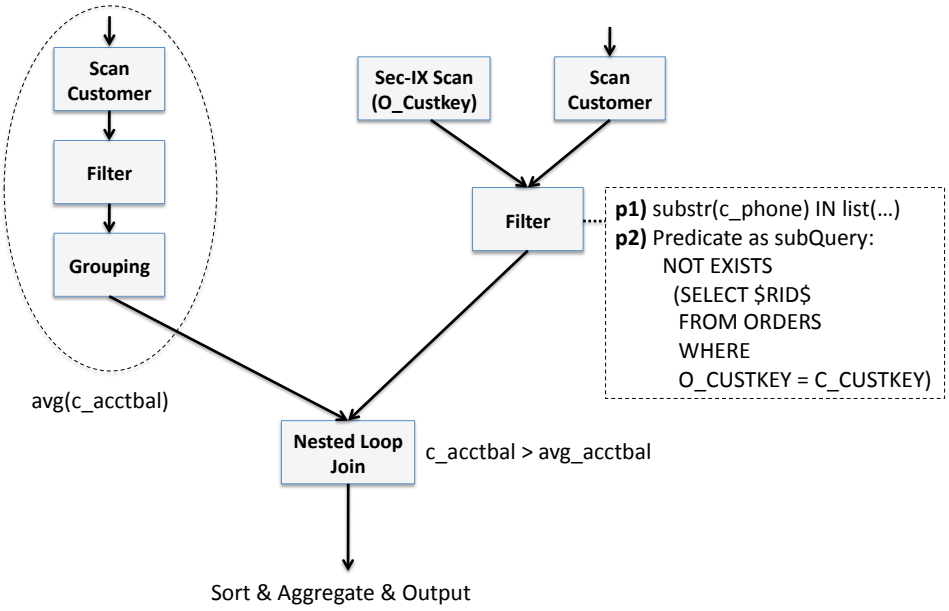

Figure 4.15: Query 22 plan - System-X

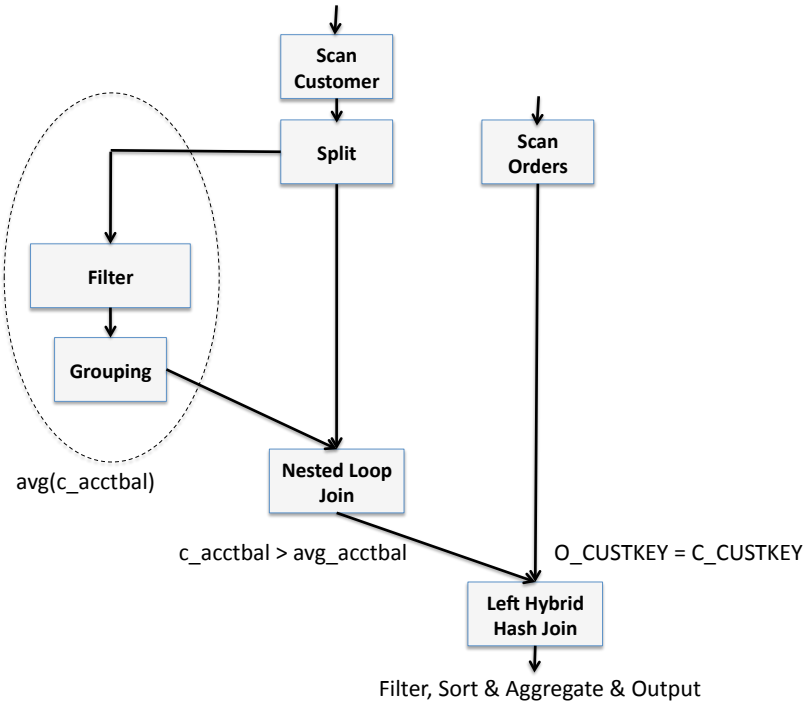

Figure 4.16: Query 22 plan - AsterixDB

4.8 Discussion

In this section, we summarize the main lessons that we have learned from our TPC-H based performance evaluation of Big Data analytics platforms. These lessons are not entirely new, however, they do provide an excellent insight into the state of reality in Big Data analytics platforms as of now.

L1. Importance of cost-based optimization

As Big Data analytics queries tend to be complex, the quality of the physical plans for processing them plays a significant role in the overall performance. Section 4.7 showed that the systems considered here come up with different query plans for most of our selected queries (for example Q10 and Q22). In practice, finding the optimal plan is a complicated task, as a complex query may have many different possible execution paths. Having a comprehensive set of rewrite rules, an accurate cost model, statistics about the data, and

reasonable estimates of the sizes of intermediate results are crucial factors in generating an efficient plan. From this perspective, relational databases still have significant advantages over modern Big Data systems. However, query optimization in new Big Data platforms is getting richer fast, as they are learning from DBMSs and also focusing on tackling various performance issues that have arisen with Big Data.

L2. Importance of storage format and I/O parallelism

Our experiments clearly showed the impact of optimized storage formats on Big Data platforms performance. Spark SQL showed a significant performance improvement when switching from the text format to Parquet. Both of the runtime engines in Hive (MapReduce and Tez) also showed performance advantages with ORC as compared to text data. A part of these performance gains comes from the reduced storage sizes with ORC and Parquet. Another key reason, however, is the inherent optimizations of these newer storage formats (such as using columnar model with built-in auxiliary information) when combined with format-specific readers. Such a combination can reduce the overhead of I/O by avoiding reading unneeded data for a query. The developers of recent formats for Spark and Hive have put considerable effort to implement such features and pairing them with other optimizations such as predicate push-down so that I/O becomes less of a bottleneck in their overall performance.

L3. Importance of dataflow processing

Our experiments re-confirmed the performance advantages that arise from using a pipelined query execution model when dealing with complex queries whose processing consists of several stages. Comparing Hive's results on Tez vs. MapReduce shows the overall impact of reduced scheduling overhead and data materialization. Both Spark and Hyracks [49] (used by AsterixDB) are execution engines that are built from scratch to run dataflow jobs efficiently. While MapReduce is excellent for one-pass computations, especially computations

whose running times require failure-tolerance, it suffers from performance-related issues for more complex jobs due to its limited (stylized) two-step programming model and its spilling intermediate results to disk.

4.9 Conclusion

In this chapter, we reported on the performance evaluation of Big Data systems for OLAP-style workloads. We used the data and read-only queries from the TPC-H benchmark to evaluate variants of four Big Data platforms: Hive, Spark SQL, AsterixDB, and System-X (a parallel RDBMS). We also used several different storage formats for Spark SQL (text and Parquet tables) and Hive (text and ORC files). For Hive, we considered both MapReduce and Tez as the underlying execution engine. Our results showed that no system or storage format gives the best performance for all of the queries. Moreover, different systems showed different scale-up behaviors. We also showed how the columnar storage formats (ORC and Parquet) improve performance in many cases. To better understand the performance differences between the systems, we analyzed a selected subset of the queries in more detail to study the impact of some of the systems' optimizations on performance.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

Emerging Big Data technologies, along with the increasing *volume*, *velocity*, and *variety* of data in Big Data applications, have created challenges in Big Data benchmarking. In this dissertation, we looked at the performance evaluation of Big Data systems from several different angles.

Modern Big Data systems can be divided into two major classes: NoSQL request-serving systems and Big Data analytics platforms. There are inherent differences between the workloads that each group is trying to serve. The first part of the thesis looked at the performance evaluation of NoSQL Key Value stores for the OLTP class of workloads. The experiments there used mixed workloads of short queries consisting of lookups, updates, and range scans, with a focus on range queries with different selectivities. Key Value stores of both hash-partitioning and range-partitioning types were included in the study. Moreover, we proposed different techniques to support range queries on top of a hash-partitioned Key Value store and studied the tradeoffs that exist in picking different schemes versus their performance.

In the second part of the thesis, we conducted a performance evaluation of Big Data systems based on their features and functionality. We proposed a new micro-benchmark, called BigFUN (for Big Data FUNctionality), which consists of configurable and extensible data and workload generator components. The workload includes both read-only and data modification operations, and each operation aims at evaluating the level of support that a system has for some core, fundamental functionality along with its performance. This part of the thesis reported on the performance of four Big Data systems using BigFUN along with a discussion of the results.

The last part of the thesis looked at the performance of the second major class of Big Data systems, namely Big Data analytics platforms, for analytics queries. We used the TPC-H benchmark for this purpose and explored the performance of four representative Big Data systems drawn from this class.

Significant differences in the architectures and design decisions of Big Data systems, along with their rapid growth have created important challenges in the task of benchmarking them. Our efforts in this thesis showed the importance of creating and utilizing standard benchmarks for evaluating and comparing Big Data platforms systematically. A mature performance evaluation effort should be comprehensive enough in terms of considering different characteristics of Big Data and the variety of the workloads to make sure that the benchmarking process does not simply concentrate on the "sweet spots" of the performance curve for a specific system. There are evident unmet needs in terms of defining the correct set of performance metrics and workload characteristics which have to be addressed to achieve this goal. We also want to emphasize on the importance of including an evaluation of the functional richness of a Big Data system within the benchmarking process. This is very important since Big Data applications tend to evolve and their needs tend to become more complicated. There is an increasing interest in using a "unified" system to serve a wide range of data processing needs rather than utilizing multiple systems. In fact, many Big

Data systems with a limited functionality are now being extended with new features in order to serve different types of applications efficiently. The "one size fits a bunch" conjecture that we briefly discussed in Chapter 3 mainly argues about this need, and one of our goals in this dissertation has been converging the tasks of performance evaluation and functionality evaluation to explore the feasibility and merits of this idea.

5.2 Future work

Further along the path of this thesis, we believe there are several potential extensions and challenging directions to be explored:

- **Adding new Big Data platforms to the performance evaluation tasks:** As mentioned in Chapter 1, new systems and solutions have been proposed in each class of Big Data systems that we considered. One potential extension to this work would be to use the benchmarks that we have considered to evaluate new systems and compare the results to the ones that we obtained.

- **Extending the data and workloads:** While we did our best in designing the schema and workload for our micro-benchmarks, there are a number of potential extensions that can be considered. Examples include:

    1. Adding more data diversity and applying a richer set of value distributions in the data generation process;
    2. Considering workloads with a focus on throughput in order to find the saturation point in a system's performance curve when dealing with many concurrent users;
    3. Involving more complex concurrent mixes of simple/short queries, analytical queries, and updates;

4. Involving concurrent mixes of queries with different levels of resource-intensiveness (including CPU, I/O, memory, and network intensities).

- **Exploring domain-centric Big Data benchmarks**: As Big Data applications are rapidly growing and becoming more complex, there is an increasing need for serving workloads which involve graph processing, machine learning, and scientific computing tasks. There is currently a lack of mature benchmarks that include rich data and operations along with generally agreed upon metrics for comparing Big Data systems on these dimensions. Creating such standard benchmarks is a challenging, but unavoidable, future task for the Big Data community.

Bibliography

[1] Apache Drill. https://drill.apache.org.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache Hive. http://hive.apache.org/.

[4] Apache Spark. http://spark.apache.org.

[5] Apache Tez. https://tez.apache.org.

[6] BSON. http://bsonspec.org/.

[7] Forbes Report. http://onforb.es/1nwCOsb/.

[8] GridMix. https://hadoop.apache.org/docs/r1.2.1/gridmix.html.

[9] HAWQ. http://hawq.incubator.apache.org.

[10] Hive on Tez. https://cwiki.apache.org/confluence/display/Hive/Hive+on+Tez.

[11] HiveContext in Spark. http://spark.apache.org/docs/latest/sql-programming-guide.html.

[12] Impala. http://impala.io/.

[13] Kryo Serializer. https://github.com/EsotericSoftware/kryo.

[14] ORC. http://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC.

[15] ORC specification. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC#LanguageManualORC-orc-spec.

[16] Parquet. https://cwiki.apache.org/confluence/display/Hive/Parquet.

[17] Parquet specification. https://parquet.apache.org/documentation/latest/.

[18] PigMix. https://cwiki.apache.org/confluence/display/PIG/PigMix.

[19] Presto. http://prestodb.io.

[20] Running TPC-H queries on Hive. https://issues.apache.org/jira/browse/HIVE-600.

[21] Snappy. https://google.github.io/snappy/.

[22] Spark JIRA issue. https://issues.apache.org/jira/browse/SPARK-10309.

[23] Spatial index internals in MongoDB. https://docs.mongodb.org/manual/core/geospatial-indexes/.

[24] TPC. http://www.tpc.org.

[25] TPC-DS. http://www.tpc.org/tpcds/.

[26] TPC-H. http://www.tpc.org/tpch.

[27] TPC-H queries in AQL. https://github.com/apache/incubator-asterixdb/tree/master/asterix-benchmarks/src/main/resources/tpc-h/queries.

[28] Trevni. https://avro.apache.org/docs/1.7.7/trevni/spec.html.

[29] WBDB. http://clds.ucsd.edu/bdbc/workshops.

[30] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[31] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[32] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. SIGMOD '09. ACM, 2009.

[33] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed B-tree. In *Proc. VLDB Endow.*, volume 1, pages 598–609, 2008.

[34] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[35] A. A. Alamoudi, R. Grover, M. J. Carey, and V. R. Borkar. External data access and indexing in asterixdb. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, pages 3–12, 2015.

[36] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *PVLDB*, 2014.

[37] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage Management in AsterixDB. *PVLDB*, 2014.

[38] Amazon S3. https://aws.amazon.com/s3/.

[39] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Peer-to-Peer Computing*, pages 33–40, 2002.

[40] Apache CouchDB. http://couchdb.apache.org/.

[41] Apache HDFS. http://hadoop.apache.org/hdfs/.

[42] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[43] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A Database Benchmark Based on the Facebook Social Graph. In *SIGMOD*, 2013.

[44] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancing and locality in range-queriable data structures. PODC '04, pages 115–124. ACM, 2004.

[45] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.

[46] C. Baru, M. Bhandarkar, R. Nambiar, M. Poess, and T. Rabl. Benchmarking Big Data Systems and the Big Data Top100 list. *Big Data*, 1(1), 2013.

[47] Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.htm.

[48] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 9:1–9:6. ACM, 2009.

[49] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE*, 2011.

[50] V. Borkar, M. Carey, D. Lychagin, T. Westmann, D. Engovatov, and N. Onose. Query Processing in the Aqualogic Data Services Platform. In *VLDB*, 2006.

[51] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD Conference*, pages 251–264, 2008.

[52] M. J. Carey. BDMS Performance Evaluation: Practices, Pitfalls, and Possibilities. In *TPCTC*, 2012.

[53] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *SIGMOD*, 1993.

[54] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke, and D. N. Shah. The BUCKY Object-Relational Benchmark. In *SIGMOD*, 1997.

[55] M. J. Carey, L. Ling, M. Nicola, and L. Shao. EXRT: Towards a Simple Benchmark for XML Readiness Testing. In *TPCTC*. Springer, 2011.

[56] Cassandra. `http://cassandra.apache.org/`.

[57] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[58] R. Cattell and J. Skeen. Object Operations Benchmark. *ACM Trans. Database Syst.*, 17(1), 1992.

[59] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, (2), 2008.

[60] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, 2008.

[61] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[62] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In *The Benchmark Handbook*. 1991.

[63] A. Edwards and G. Davies. Understanding Analytic Workloads - Meeting the complex processing demands of advanced analytics. 2011.

[64] A. Floratou, U. F. Minhas, and F. Özcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment*, 7(12):1295–1306, 2014.

[65] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for MapReduce. *Proceedings of the VLDB Endowment*, 4(7):419–429, 2011.

[66] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the Elephants Handle the NoSQL Onslaught? *PVLDB*, 2012.

[67] P. Ganesan, M. Bawa, and H. Garcia-molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *In VLDB*, pages 444–455, 2004.

[68] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multidimensional Queries in P2P Systems. In *WebDB*, 2004.

[69] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD*, 2013.

[70] Google App Engine. `https://cloud.google.com/appengine/`.

[71] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Rec.*, 23:243–252, May 1994.

[72] R. Grover and M. Carey. Data Ingestion in AsterixDB. *EDBT*, 2015.

[73] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *CIDR*, 2003.

[74] R. Han and X. Lu. On Big Data Benchmarking. *CoRR*, abs/1402.5194, 2014.

[75] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazons highly available key-value store. In *In Proc. SOSP*, pages 205–220, 2007.

[76] HBase. `http://hbase.apache.org/`.

[77] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.

[78] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *In VLDB*, pages 661–672, 2005.

[79] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[80] D. Lomet. Replicated indexes for distributed data. DIS '96, pages 108–119. IEEE Computer Society, 1996.

[81] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[82] MongoDB. `http://www.mongodb.org/`.

[83] MySQL. `https://www.mysql.com/`.

[84] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[85] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *SoCC*, 2011.

[86] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[87] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *EDBT*, pages 131–148, 2006.

[88] Project Voldemort. `http://project-voldemort.com/`.

[89] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. *PVLDB*, 2012.

[90] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: prefix hash tree. PODC '04. ACM, 2004.

[91] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, 2015.

[92] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A Peer-to-peer Framework for Caching Range Queries. In *ICDE*, pages 165–176, 2004.

[93] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.

[94] T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. In *CCGRID*, 2006.

[95] T. Schütt, F. Schintke, and A. Reinefeld. Range queries on structured overlay networks. In *Computer Communications*, volume 31, 2008.

[96] O. Serlin. The history of debitcredit and the tpc.

[97] Y. Shi, X. Meng, J. Zhao, X. Hu, B. Liu, and H. Wang. Benchmarking cloud-based data management systems. In *Proceedings of the second international workshop on Cloud data management*, CloudDB '10, pages 47–54. ACM, 2010.

[98] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[99] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[100] H. T. Vo, C. Chen, and B. C. Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. In *PVLDB*, volume 3, pages 506–517, 2010.

[101] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, and S. Zhang. Bigdatabench: A Big Data Benchmark Suite from Internet Services. In *IEEE HPCA*, 2014.

[102] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[103] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

Appendix A

TPC-H Queries in AQL

This appendix includes the AQL version of all 22 TPC-H queries used in our experiments in Chapter 4.

```
SET IMPORT-PRIVATE-FUNCTIONS 'TRUE';

FOR $L IN DATASET('LINEITEM')
WHERE $L.L_SHIPDATE /*+ SKIP-INDEX */ <= '1998-09-02'
/*+ HASH*/
GROUP BY $L_RETURNFLAG := $L.L_RETURNFLAG,
         $L_LINESTATUS := $L.L_LINESTATUS
  WITH $L
ORDER BY $L_RETURNFLAG, $L_LINESTATUS
RETURN {
  "L_RETURNFLAG": $L_RETURNFLAG,
  "L_LINESTATUS": $L_LINESTATUS,
  "SUM_QTY": SUM(FOR $I IN $L RETURN $I.L_QUANTITY),
  "SUM_BASE_PRICE": SUM(FOR $I IN $L RETURN $I.L_EXTENDEDPRICE),
  "SUM_DISC_PRICE": SUM(FOR $I IN $L RETURN $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT)),
  "SUM_CHARGE":
        SUM(FOR $I IN $L
        RETURN $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT) * (1 + $I.L_TAX)),
  "AVE_QTY": AVG(FOR $I IN $L RETURN $I.L_QUANTITY),
  "AVE_PRICE": AVG(FOR $I IN $L RETURN $I.L_EXTENDEDPRICE),
  "AVE_DISC": AVG(FOR $I IN $L RETURN $I.L_DISCOUNT),
  "COUNT_ORDER": COUNT($L)
}
```

Listing A.39: TPC-H Query 1 (in AQL)

```
DECLARE FUNCTION TMP1() {
  FOR $P IN DATASET('PART')
  FOR $PSSRN IN (
    FOR $PS IN DATASET('PARTSUPP')
    FOR $SRN IN (
      FOR $S IN DATASET('SUPPLIER')
      FOR $RN IN (
        FOR $R IN DATASET('REGION')
        FOR $N IN DATASET('NATION')
        WHERE $N.N_REGIONKEY = $R.R_REGIONKEY AND $R.R_NAME = 'EUROPE'
        RETURN {
          "N_NATIONKEY": $N.N_NATIONKEY,
          "N_NAME": $N.N_NAME
        }
      )
      WHERE $S.S_NATIONKEY = $RN.N_NATIONKEY
      RETURN {
        "S_SUPPKEY": $S.S_SUPPKEY,
        "N_NAME": $RN.N_NAME,
        "S_NAME": $S.S_NAME,
        "S_ACCTBAL": $S.S_ACCTBAL,
        "S_ADDRESS": $S.S_ADDRESS,
        "S_PHONE": $S.S_PHONE,
        "S_COMMENT": $S.S_COMMENT
      }
    )
    WHERE $SRN.S_SUPPKEY = $PS.PS_SUPPKEY
    RETURN {
      "N_NAME": $SRN.N_NAME,
      "P_PARTKEY": $PS.PS_PARTKEY,
      "PS_SUPPLYCOST": $PS.PS_SUPPLYCOST,
      "S_NAME": $SRN.S_NAME,
      "S_ACCTBAL": $SRN.S_ACCTBAL,
      "S_ADDRESS":  $SRN.S_ADDRESS,
      "S_PHONE":  $SRN.S_PHONE,
      "S_COMMENT":  $SRN.S_COMMENT
    }
  )
  WHERE $P.P_PARTKEY = $PSSRN.P_PARTKEY AND LIKE($P.P_TYPE, '%BRASS') AND $P.P_SIZE = 15
  RETURN {
    "S_ACCTBAL": $PSSRN.S_ACCTBAL,
    "S_NAME": $PSSRN.S_NAME,
    "N_NAME": $PSSRN.N_NAME,
    "P_PARTKEY": $P.P_PARTKEY,
    "PS_SUPPLYCOST": $PSSRN.PS_SUPPLYCOST,
    "P_MFGR": $P.P_MFGR,
    "S_ADDRESS":  $PSSRN.S_ADDRESS,
    "S_PHONE":  $PSSRN.S_PHONE,
    "S_COMMENT":  $PSSRN.S_COMMENT
  }
}
```

Listing A.40: TPC-H Query 2 (in AQL) - Part 1

```
DECLARE FUNCTION TMP2(){
  FOR $P IN DATASET('PART')
  FOR $PSSRN IN (
    FOR $PS IN DATASET('PARTSUPP')
    FOR $SRN IN (
      FOR $S IN DATASET('SUPPLIER')
      FOR $RN IN (
        FOR $R IN DATASET('REGION')
        FOR $N IN DATASET('NATION')
        WHERE $N.N_REGIONKEY = $R.R_REGIONKEY AND $R.R_NAME = 'EUROPE'
        RETURN {
          "N_NATIONKEY": $N.N_NATIONKEY,
          "N_NAME": $N.N_NAME
        }
      )
      WHERE $S.S_NATIONKEY = $RN.N_NATIONKEY
      RETURN {
        "S_SUPPKEY": $S.S_SUPPKEY,
        "N_NAME": $RN.N_NAME,
        "S_NAME": $S.S_NAME,
        "S_ACCTBAL": $S.S_ACCTBAL,
        "S_ADDRESS": $S.S_ADDRESS,
        "S_PHONE": $S.S_PHONE,
        "S_COMMENT": $S.S_COMMENT
      }
    )
    WHERE $SRN.S_SUPPKEY = $PS.PS_SUPPKEY
    RETURN {
      "N_NAME": $SRN.N_NAME,
      "P_PARTKEY": $PS.PS_PARTKEY,
      "PS_SUPPLYCOST": $PS.PS_SUPPLYCOST,
      "S_NAME": $SRN.S_NAME,
      "S_ACCTBAL": $SRN.S_ACCTBAL,
      "S_ADDRESS":  $SRN.S_ADDRESS,
      "S_PHONE":  $SRN.S_PHONE,
      "S_COMMENT":  $SRN.S_COMMENT
    }
  )
  WHERE $P.P_PARTKEY = $PSSRN.P_PARTKEY AND LIKE($P.P_TYPE, '%BRASS') AND $P.P_SIZE = 15
  /*+ HASH*/
  GROUP BY $P_PARTKEY := $PSSRN.P_PARTKEY WITH $PSSRN
  RETURN {
    "P_PARTKEY": $P_PARTKEY,
    "PS_MIN_SUPPLYCOST": MIN(FOR $I IN $PSSRN RETURN $I.PS_SUPPLYCOST)
  }
}

FOR $T2 IN TMP2()
FOR $T1 IN TMP1()
WHERE $T1.P_PARTKEY = $T2.P_PARTKEY AND $T1.PS_SUPPLYCOST = $T2.PS_MIN_SUPPLYCOST
ORDER BY $T1.S_ACCTBAL DESC, $T1.N_NAME, $T1.S_NAME, $T1.P_PARTKEY
LIMIT 100
RETURN
{
  "S_ACCTBAL": $T1.S_ACCTBAL,
  "S_NAME": $T1.S_NAME,
  "N_NAME": $T1.N_NAME,
  "P_PARTKEY": $T1.P_PARTKEY,
  "P_MFGR": $T1.P_MFGR,
  "S_ADDRESS": $T1.S_ADDRESS,
  "S_PHONE": $T1.S_PHONE,
  "S_COMMENT": $T1.S_COMMENT
}
```

Listing A.40: TPC-H Query 2 (in AQL) - Part 2

```
FOR $C IN DATASET('CUSTOMER')
FOR $O IN DATASET('ORDERS')
WHERE
  $C.C_MKTSEGMENT = 'BUILDING' AND $C.C_CUSTKEY = $O.O_CUSTKEY
FOR $L IN DATASET('LINEITEM')
WHERE
  $L.L_ORDERKEY = $O.O_ORDERKEY AND
  $O.O_ORDERDATE < '1995-03-15' AND $L.L_SHIPDATE > '1995-03-15'
/*+ HASH*/
GROUP BY $L_ORDERKEY := $L.L_ORDERKEY,
        $O_ORDERDATE := $O.O_ORDERDATE,
        $O_SHIPPRIORITY := $O.O_SHIPPRIORITY
  WITH $L
LET $REVENUE := SUM (
  FOR $I IN $L
  RETURN
    $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT)
)
ORDER BY $REVENUE DESC, $O_ORDERDATE
LIMIT 10
RETURN {
  "L_ORDERKEY": $L_ORDERKEY,
  "REVENUE": $REVENUE,
  "O_ORDERDATE": $O_ORDERDATE,
  "O_SHIPPRIORITY": $O_SHIPPRIORITY
}
```

Listing A.41: TPC-H Query 3 (in AQL)

```
DECLARE FUNCTION TMP()
{
  FOR $L IN DATASET('LINEITEM')
  WHERE $L.L_COMMITDATE < $L.L_RECEIPTDATE
  DISTINCT BY $L.L_ORDERKEY
  RETURN { "O_ORDERKEY": $L.L_ORDERKEY }
}

FOR $O IN DATASET('ORDERS')
FOR $T IN TMP()
WHERE $O.O_ORDERKEY = $T.O_ORDERKEY AND
  $O.O_ORDERDATE >= '1993-07-01' AND $O.O_ORDERDATE < '1993-10-01'
GROUP BY $O_ORDERPRIORITY := $O.O_ORDERPRIORITY WITH $O
ORDER BY $O_ORDERPRIORITY
RETURN {
  "ORDER_PRIORITY": $O_ORDERPRIORITY,
  "COUNT": COUNT($O)
}
```

Listing A.42: TPC-H Query 4 (in AQL)

```
FOR $C IN DATASET('CUSTOMER')
FOR $O1 IN (
  FOR $O IN DATASET('ORDERS')
  FOR $L1 IN (
    FOR $L IN DATASET('LINEITEM')
    FOR $S1 IN (
      FOR $S IN DATASET('SUPPLIER')
      FOR $N1 IN (
        FOR $N IN DATASET('NATION')
        FOR $R IN DATASET('REGION')
        WHERE $N.N_REGIONKEY = $R.R_REGIONKEY
          AND $R.R_NAME = 'ASIA'
        RETURN {
          "N_NAME": $N.N_NAME,
          "N_NATIONKEY": $N.N_NATIONKEY
        }
      )
      WHERE $S.S_NATIONKEY = $N1.N_NATIONKEY
      RETURN {
        "N_NAME": $N1.N_NAME,
        "S_SUPPKEY": $S.S_SUPPKEY,
        "S_NATIONKEY": $S.S_NATIONKEY
      }
    )
    WHERE $L.L_SUPPKEY = $S1.S_SUPPKEY
    RETURN {
      "N_NAME": $S1.N_NAME,
      "L_EXTENDEDPRICE": $L.L_EXTENDEDPRICE,
      "L_DISCOUNT": $L.L_DISCOUNT,
      "L_ORDERKEY": $L.L_ORDERKEY,
      "S_NATIONKEY": $S1.S_NATIONKEY
    }
  )
  WHERE $L1.L_ORDERKEY = $O.O_ORDERKEY AND
        $O.O_ORDERDATE >= '1994-01-01' AND $O.O_ORDERDATE < '1995-01-01'
  RETURN {
    "N_NAME": $L1.N_NAME,
    "L_EXTENDEDPRICE": $L1.L_EXTENDEDPRICE,
    "L_DISCOUNT": $L1.L_DISCOUNT,
    "S_NATIONKEY": $L1.S_NATIONKEY,
    "O_CUSTKEY": $O.O_CUSTKEY
  }
)
WHERE $C.C_NATIONKEY = $O1.S_NATIONKEY AND $C.C_CUSTKEY = $O1.O_CUSTKEY
/*+ HASH*/
GROUP BY $N_NAME := $O1.N_NAME WITH $O1
LET $REVENUE := SUM (
  FOR $I IN $O1
  RETURN
    $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT)
)
ORDER BY $REVENUE DESC
RETURN {
  "N_NAME": $N_NAME,
  "REVENUE": $REVENUE
}
```

Listing A.43: TPC-H Query 5 (in AQL)

```
LET $REVENUE := SUM(
  FOR $L IN DATASET('LINEITEM')
  WHERE $L.L_SHIPDATE >= '1994-01-01'
    AND $L.L_SHIPDATE < '1995-01-01'
    AND $L.L_DISCOUNT >= 0.05 AND $L.L_DISCOUNT <= 0.07
    AND $L.L_QUANTITY < 24
  RETURN $L.L_EXTENDEDPRICE * $L.L_DISCOUNT
)
RETURN {
  "REVENUE": $REVENUE
}
```

Listing A.44: TPC-H Query 6 (in AQL)

```
DECLARE FUNCTION Q7_VOLUME_SHIPPING_TMP() {
  FOR $N1 IN DATASET('NATION')
  FOR $N2 IN DATASET('NATION')
  WHERE ($N1.N_NAME='FRANCE' AND $N2.N_NAME='GERMANY') OR
        ($N1.N_NAME='GERMANY' AND $N2.N_NAME='FRANCE')
  RETURN {
    "SUPP_NATION": $N1.N_NAME,
    "CUST_NATION": $N2.N_NAME,
    "S_NATIONKEY": $N1.N_NATIONKEY,
    "C_NATIONKEY": $N2.N_NATIONKEY
  }
}

FOR $LOCS IN (
  FOR $LOC IN (
    FOR $LO IN (
      FOR $L IN DATASET('LINEITEM')
      FOR $O IN DATASET('ORDERS')
      WHERE $O.O_ORDERKEY = $L.L_ORDERKEY AND $L.L_SHIPDATE >= '1995-01-01'
        AND $L.L_SHIPDATE <= '1996-12-31'
      RETURN {
        "L_SHIPDATE": $L.L_SHIPDATE,
        "L_EXTENDEDPRICE": $L.L_EXTENDEDPRICE,
        "L_DISCOUNT": $L.L_DISCOUNT,
        "L_SUPPKEY": $L.L_SUPPKEY,
        "O_CUSTKEY": $O.O_CUSTKEY
      }
    )
    FOR $C IN DATASET('CUSTOMER')
    WHERE $C.C_CUSTKEY = $LO.O_CUSTKEY
    RETURN {
      "L_SHIPDATE": $LO.L_SHIPDATE,
      "L_EXTENDEDPRICE": $LO.L_EXTENDEDPRICE,
      "L_DISCOUNT": $LO.L_DISCOUNT,
      "L_SUPPKEY": $LO.L_SUPPKEY,
      "C_NATIONKEY": $C.C_NATIONKEY
    }
  )
  FOR $S IN DATASET('SUPPLIER')
  WHERE $S.S_SUPPKEY = $LOC.L_SUPPKEY
  RETURN {
    "L_SHIPDATE": $LOC.L_SHIPDATE,
    "L_EXTENDEDPRICE": $LOC.L_EXTENDEDPRICE,
    "L_DISCOUNT": $LOC.L_DISCOUNT,
    "C_NATIONKEY": $LOC.C_NATIONKEY,
    "S_NATIONKEY": $S.S_NATIONKEY
  }
)
FOR $T IN Q7_VOLUME_SHIPPING_TMP()
WHERE $LOCS.C_NATIONKEY = $T.C_NATIONKEY
  AND $LOCS.S_NATIONKEY = $T.S_NATIONKEY
LET $L_YEAR0 := GET-YEAR($LOCS.L_SHIPDATE)
GROUP BY $SUPP_NATION := $T.SUPP_NATION, $CUST_NATION := $T.CUST_NATION, $L_YEAR := $L_YEAR0
WITH $LOCS
LET $REVENUE := SUM(FOR $I IN $LOCS RETURN $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT))
ORDER BY $SUPP_NATION, $CUST_NATION, $L_YEAR
RETURN {
  "SUPP_NATION": $SUPP_NATION,
  "CUST_NATION": $CUST_NATION,
  "L_YEAR": $L_YEAR,
  "REVENUE": $REVENUE
}
```

Listing A.45: TPC-H Query 7 (in AQL)

```
FOR $T IN (
  FOR $SLNRCOP IN (
    FOR $S IN DATASET("SUPPLIER")
    FOR $LNRCOP IN (
      FOR $LNRCO IN (
        FOR $L IN DATASET('LINEITEM')
        FOR $NRCO IN (
          FOR $O IN DATASET('ORDERS')
          FOR $NRC IN (
            FOR $C IN DATASET('CUSTOMER')
            FOR $NR IN (
              FOR $N1 IN DATASET('NATION')
              FOR $R1 IN DATASET('REGION')
              WHERE $N1.N_REGIONKEY = $R1.R_REGIONKEY AND $R1.R_NAME = 'AMERICA'
              RETURN { "N_NATIONKEY": $N1.N_NATIONKEY }
            )
            WHERE $C.C_NATIONKEY = $NR.N_NATIONKEY
            RETURN { "C_CUSTKEY": $C.C_CUSTKEY }
          )
          WHERE $NRC.C_CUSTKEY = $O.O_CUSTKEY
          RETURN {
            "O_ORDERDATE" : $O.O_ORDERDATE,
            "O_ORDERKEY": $O.O_ORDERKEY
          }
        )
        WHERE $L.L_ORDERKEY = $NRCO.O_ORDERKEY
          AND $NRCO.O_ORDERDATE >= '1995-01-01'
          AND $NRCO.O_ORDERDATE <= '1996-12-31'
        RETURN {
          "O_ORDERDATE": $NRCO.O_ORDERDATE,
          "L_PARTKEY": $L.L_PARTKEY,
          "L_DISCOUNT": $L.L_DISCOUNT,
          "L_EXTENDEDPRICE": $L.L_EXTENDEDPRICE,
          "L_SUPPKEY": $L.L_SUPPKEY
        }
      )
      FOR $P IN DATASET("PART")
      WHERE $P.P_PARTKEY = $LNRCO.L_PARTKEY AND $P.P_TYPE = 'ECONOMY ANODIZED STEEL'
      RETURN {
        "O_ORDERDATE": $LNRCO.O_ORDERDATE,
        "L_DISCOUNT": $LNRCO.L_DISCOUNT,
        "L_EXTENDEDPRICE": $LNRCO.L_EXTENDEDPRICE,
        "L_SUPPKEY": $LNRCO.L_SUPPKEY
      }
    )
    WHERE $S.S_SUPPKEY = $LNRCOP.L_SUPPKEY
    RETURN {
      "O_ORDERDATE": $LNRCOP.O_ORDERDATE,
      "L_DISCOUNT": $LNRCOP.L_DISCOUNT,
      "L_EXTENDEDPRICE": $LNRCOP.L_EXTENDEDPRICE,
      "L_SUPPKEY": $LNRCOP.L_SUPPKEY,
      "S_NATIONKEY": $S.S_NATIONKEY
    }
  )
  FOR $N2 IN DATASET('NATION')
  WHERE $SLNRCOP.S_NATIONKEY = $N2.N_NATIONKEY
  LET $O_YEAR := GET-YEAR($SLNRCOP.O_ORDERDATE)
  RETURN {
    "YEAR": $O_YEAR,
    "REVENUE": $SLNRCOP.L_EXTENDEDPRICE *(1-$SLNRCOP.L_DISCOUNT),
    "S_NAME": $N2.N_NAME
  }
)
GROUP BY $YEAR := $T.YEAR WITH $T
ORDER BY $YEAR
RETURN {
  "YEAR": $YEAR,
  "MKT_SHARE": SUM(FOR $I IN $T
        RETURN SWITCH-CASE($I.S_NAME='BRAZIL', TRUE, $I.REVENUE, FALSE, 0.0))/
        SUM(FOR $I IN $T RETURN $I.REVENUE)
}
```


Listing A.46: TPC-H Query 8 (in AQL)

```
FOR $PROFIT IN (
  FOR $O IN DATASET('ORDERS')
  FOR $L3 IN (
    FOR $P IN DATASET('PART')
    FOR $L2 IN (
      FOR $PS IN DATASET('PARTSUPP')
      FOR $L1 IN (
        FOR $S1 IN (
          FOR $S IN DATASET('SUPPLIER')
          FOR $N IN DATASET('NATION')
          WHERE $N.N_NATIONKEY = $S.S_NATIONKEY
          RETURN {
            "S_SUPPKEY": $S.S_SUPPKEY,
            "N_NAME": $N.N_NAME
          }
        )
        FOR $L IN DATASET('LINEITEM')
        WHERE $S1.S_SUPPKEY = $L.L_SUPPKEY
        RETURN  {
          "L_SUPPKEY": $L.L_SUPPKEY,
          "L_EXTENDEDPRICE": $L.L_EXTENDEDPRICE,
          "L_DISCOUNT": $L.L_DISCOUNT,
          "L_QUANTITY": $L.L_QUANTITY,
          "L_PARTKEY": $L.L_PARTKEY,
          "L_ORDERKEY": $L.L_ORDERKEY,
          "N_NAME": $S1.N_NAME
        }
      )
      WHERE $PS.PS_SUPPKEY = $L1.L_SUPPKEY AND $PS.PS_PARTKEY = $L1.L_PARTKEY
      RETURN {
        "L_EXTENDEDPRICE": $L1.L_EXTENDEDPRICE,
        "L_DISCOUNT": $L1.L_DISCOUNT,
        "L_QUANTITY": $L1.L_QUANTITY,
        "L_PARTKEY": $L1.L_PARTKEY,
        "L_ORDERKEY": $L1.L_ORDERKEY,
        "N_NAME": $L1.N_NAME,
        "PS_SUPPLYCOST": $PS.PS_SUPPLYCOST
      }
    )
    WHERE CONTAINS($P.P_NAME, 'GREEN') AND $P.P_PARTKEY = $L2.L_PARTKEY
    RETURN {
      "L_EXTENDEDPRICE": $L2.L_EXTENDEDPRICE,
      "L_DISCOUNT": $L2.L_DISCOUNT,
      "L_QUANTITY": $L2.L_QUANTITY,
      "L_ORDERKEY": $L2.L_ORDERKEY,
      "N_NAME": $L2.N_NAME,
      "PS_SUPPLYCOST": $L2.PS_SUPPLYCOST
    }
  )
  WHERE $O.O_ORDERKEY = $L3.L_ORDERKEY
  LET $AMOUNT := $L3.L_EXTENDEDPRICE * (1 - $L3.L_DISCOUNT) -
$L3.PS_SUPPLYCOST * $L3.L_QUANTITY
  LET $O_YEAR := GET-YEAR($O.O_ORDERDATE)
  RETURN {
    "NATION": $L3.N_NAME,
    "O_YEAR": $O_YEAR,
    "AMOUNT": $AMOUNT
  }
)
GROUP BY $NATION := $PROFIT.NATION, $O_YEAR := $PROFIT.O_YEAR WITH $PROFIT
ORDER BY $NATION, $O_YEAR DESC
RETURN {
  "NATION": $NATION,
  "O_YEAR": $O_YEAR,
  "SUM_PROFIT": SUM( FOR $PR IN $PROFIT RETURN $PR.AMOUNT )
}
```

Listing A.47: TPC-H Query 9 (in AQL)

```
FOR $LOCN IN (
  FOR $L IN DATASET('LINEITEM')
  FOR $OCN IN (
    FOR $O IN DATASET('ORDERS')
    FOR $C IN DATASET('CUSTOMER')
    WHERE $C.C_CUSTKEY = $O.O_CUSTKEY AND $O.O_ORDERDATE >= '1993-10-01'
      AND $O.O_ORDERDATE < '1994-01-01'
    FOR $N IN DATASET('NATION')
    WHERE $C.C_NATIONKEY = $N.N_NATIONKEY
    RETURN {
      "C_CUSTKEY": $C.C_CUSTKEY,
      "C_NAME": $C.C_NAME,
      "C_ACCTBAL": $C.C_ACCTBAL,
      "N_NAME": $N.N_NAME,
      "C_ADDRESS": $C.C_ADDRESS,
      "C_PHONE": $C.C_PHONE,
      "C_COMMENT": $C.C_COMMENT,
      "O_ORDERKEY": $O.O_ORDERKEY
    }
  )
  WHERE $L.L_ORDERKEY = $OCN.O_ORDERKEY AND $L.L_RETURNFLAG = 'R'
  RETURN {
    "C_CUSTKEY": $OCN.C_CUSTKEY,
    "C_NAME": $OCN.C_NAME,
    "C_ACCTBAL": $OCN.C_ACCTBAL,
    "N_NAME": $OCN.N_NAME,
    "C_ADDRESS": $OCN.C_ADDRESS,
    "C_PHONE": $OCN.C_PHONE,
    "C_COMMENT": $OCN.C_COMMENT,
    "L_EXTENDEDPRICE": $L.L_EXTENDEDPRICE,
    "L_DISCOUNT": $L.L_DISCOUNT
  }
)
GROUP BY $C_CUSTKEY:=$LOCN.C_CUSTKEY,
    $C_NAME:=$LOCN.C_NAME,
    $C_ACCTBAL:=$LOCN.C_ACCTBAL, $C_PHONE:=$LOCN.C_PHONE,
    $N_NAME:=$LOCN.N_NAME, $C_ADDRESS:=$LOCN.C_ADDRESS, $C_COMMENT:=$LOCN.C_COMMENT
    WITH $LOCN
LET $REVENUE := SUM(FOR $I IN $LOCN RETURN $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT))
ORDER BY $REVENUE DESC
LIMIT 20
RETURN {
  "C_CUSTKEY": $C_CUSTKEY,
  "C_NAME": $C_NAME,
  "REVENUE": $REVENUE,
  "C_ACCTBAL": $C_ACCTBAL,
  "N_NAME": $N_NAME,
  "C_ADDRESS": $C_ADDRESS,
  "C_PHONE": $C_PHONE,
  "C_COMMENT": $C_COMMENT
}
```

Listing A.48: TPC-H Query 10 (in AQL)

```
LET $SUM := SUM (
  FOR $PS IN DATASET('PARTSUPP')
  FOR $SN IN (
    FOR $S IN DATASET('SUPPLIER')
    FOR $N IN DATASET('NATION')
    WHERE $S.S_NATIONKEY = $N.N_NATIONKEY
      AND $N.N_NAME = 'GERMANY'
    RETURN { "S_SUPPKEY": $S.S_SUPPKEY }
  )
  WHERE $PS.PS_SUPPKEY = $SN.S_SUPPKEY
  RETURN $PS.PS_SUPPLYCOST * $PS.PS_AVAILQTY
)
FOR $T1 IN (
  FOR $PS IN DATASET('PARTSUPP')
  FOR $SN IN (
    FOR $S IN DATASET('SUPPLIER')
    FOR $N IN DATASET('NATION')
    WHERE $S.S_NATIONKEY = $N.N_NATIONKEY
      AND $N.N_NAME = 'GERMANY'
    RETURN { "S_SUPPKEY": $S.S_SUPPKEY }
  )
  WHERE $PS.PS_SUPPKEY = $SN.S_SUPPKEY
  GROUP BY $PS_PARTKEY := $PS.PS_PARTKEY WITH $PS
  RETURN {
    "PS_PARTKEY": $PS_PARTKEY,
    "PART_VALUE": SUM(FOR $I IN $PS RETURN $I.PS_SUPPLYCOST * $I.PS_AVAILQTY)
  }
)
WHERE $T1.PART_VALUE > $SUM * (0.0001 / SF) [Check TPC-H specification]
ORDER BY $T1.PART_VALUE DESC
RETURN {
  "PARTKEY": $T1.PS_PARTKEY,
  "PART_VALUE": $T1.PART_VALUE
}
```

Listing A.49: TPC-H Query 11 (in AQL)

```
FOR $L IN DATASET('LINEITEM')
FOR $O IN DATASET('ORDERS')
WHERE $O.O_ORDERKEY = $L.L_ORDERKEY
  AND $L.L_COMMITDATE < $L.L_RECEIPTDATE
  AND $L.L_SHIPDATE < $L.L_COMMITDATE
  AND $L.L_RECEIPTDATE >= '1994-01-01'
  AND $L.L_RECEIPTDATE < '1995-01-01'
  AND ($L.L_SHIPMODE = 'MAIL' OR $L.L_SHIPMODE = 'SHIP')
GROUP BY $L_SHIPMODE := $L.L_SHIPMODE WITH $O
ORDER BY $L_SHIPMODE
RETURN {
  "L_SHIPMODE": $L_SHIPMODE,
  "HIGH_LINE_COUNT": SUM(
    FOR $I IN $O
    RETURN
      SWITCH-CASE($I.O_ORDERPRIORITY ='1-URGENT' OR $I.O_ORDERPRIORITY ='2-HIGH',
                  TRUE, 1, FALSE, 0)
  ),
  "LOW_LINE_COUNT": SUM(
    FOR $I IN $O
    RETURN SWITCH-CASE($I.O_ORDERPRIORITY ='1-URGENT' OR $I.O_ORDERPRIORITY ='2-HIGH',
                       TRUE, 0, FALSE, 1)
  )
}
```

Listing A.50: TPC-H Query 12 (in AQL)

```
SET IMPORT-PRIVATE-FUNCTIONS 'TRUE';

FOR $GCO IN (
  FOR $CO IN (
    FOR $C IN DATASET('CUSTOMER')
    RETURN {
      "C_CUSTKEY": $C.C_CUSTKEY,
      "O_ORDERKEY_COUNT": COUNT(
        FOR $O IN DATASET('ORDERS')
        WHERE  $C.C_CUSTKEY = $O.O_CUSTKEY AND NOT(LIKE($O.O_COMMENT,'%SPECIAL%REQUESTS%'))
        RETURN $O.O_ORDERKEY
      )
    }
  )
  GROUP BY $C_CUSTKEY := $CO.C_CUSTKEY WITH $CO
  RETURN {
    "C_CUSTKEY": $C_CUSTKEY,
    "C_COUNT": SUM(FOR $I IN $CO RETURN $I.O_ORDERKEY_COUNT)
  }
)
GROUP BY $C_COUNT := $GCO.C_COUNT WITH $GCO
LET $CUSTDIST := COUNT($GCO)
ORDER BY $CUSTDIST DESC, $C_COUNT DESC
RETURN {
  "C_COUNT": $C_COUNT,
  "CUSTDIST": $CUSTDIST
}
```

Listing A.51: TPC-H Query 13 (in AQL)

```
FOR $L IN DATASET('LINEITEM')
FOR $P IN DATASET('PART')
WHERE $L.L_PARTKEY = $P.P_PARTKEY
  AND $L.L_SHIPDATE >= '1995-09-01'
  AND $L.L_SHIPDATE < '1995-10-01'
LET $LP := {
    "L_EXTENDEDPRICE": $L.L_EXTENDEDPRICE,
    "L_DISCOUNT": $L.L_DISCOUNT,
    "P_TYPE": $P.P_TYPE
  }
GROUP BY $T:=1 WITH $LP
RETURN 100.00 * SUM(
  FOR $I IN $LP
  RETURN SWITCH-CASE(LIKE($I.P_TYPE, 'PROMO%'),
                     TRUE, $I.L_EXTENDEDPRICE*(1-$I.L_DISCOUNT),
                     FALSE, 0.0)
  ) / SUM(FOR $I IN $LP RETURN $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT)
)
```

Listing A.52: TPC-H Query 14 (in AQL)

```
DECLARE FUNCTION REVENUE() {
  FOR $L IN DATASET('LINEITEM')
  WHERE $L.L_SHIPDATE >= '1996-01-01' AND $L.L_SHIPDATE < '1996-04-01'
  GROUP BY $L_SUPPKEY := $L.L_SUPPKEY WITH $L
  RETURN {
    "SUPPLIER_NO": $L_SUPPKEY,
    "TOTAL_REVENUE": SUM(FOR $I IN $L RETURN $I.L_EXTENDEDPRICE * (1 - $I.L_DISCOUNT))
  }
}

LET $M := MAX(
  FOR $R2 IN REVENUE()
  RETURN $R2.TOTAL_REVENUE
)

FOR $S IN DATASET('SUPPLIER')
FOR $R IN REVENUE()
WHERE $S.S_SUPPKEY = $R.SUPPLIER_NO AND
        $R.TOTAL_REVENUE<$M+0.000000001 AND
        $R.TOTAL_REVENUE>$M-0.000000001
RETURN {
  "S_SUPPKEY": $S.S_SUPPKEY,
  "S_NAME": $S.S_NAME,
  "S_ADDRESS": $S.S_ADDRESS,
  "S_PHONE": $S.S_PHONE,
  "TOTAL_REVENUE": $R.TOTAL_REVENUE
}
```

Listing A.53: TPC-H Query 15 (in AQL)

```
DECLARE FUNCTION TMP(){
  FOR $PSP IN (
    FOR $PS IN DATASET('PARTSUPP')
    FOR $P IN DATASET('PART')
    WHERE $P.P_PARTKEY = $PS.PS_PARTKEY AND $P.P_BRAND != 'BRAND#45'
      AND NOT(LIKE($P.P_TYPE, 'MEDIUM POLISHED%'))
    RETURN {
      "P_BRAND": $P.P_BRAND,
      "P_TYPE": $P.P_TYPE,
      "P_SIZE": $P.P_SIZE,
      "PS_SUPPKEY": $PS.PS_SUPPKEY
    }
  )
  FOR $S IN DATASET('SUPPLIER')
  WHERE $PSP.PS_SUPPKEY = $S.S_SUPPKEY AND NOT(LIKE($S.S_COMMENT, '%CUSTOMER%COMPLAINTS%'))
  RETURN {
   "P_BRAND": $PSP.P_BRAND,
   "P_TYPE": $PSP.P_TYPE,
   "P_SIZE": $PSP.P_SIZE,
   "PS_SUPPKEY": $PSP.PS_SUPPKEY
  }
}

FOR $T2 IN (
  FOR $T IN TMP()
  WHERE $T.P_SIZE = 49 OR $T.P_SIZE = 14 OR $T.P_SIZE = 23
    OR $T.P_SIZE = 45 OR $T.P_SIZE = 19 OR $T.P_SIZE = 3
    OR $T.P_SIZE = 36 OR $T.P_SIZE = 9
  GROUP BY $P_BRAND1:= $T.P_BRAND, $P_TYPE1 := $T.P_TYPE,
    $P_SIZE1:= $T.P_SIZE, $PS_SUPPKEY1:=$T.PS_SUPPKEY WITH $T
  RETURN {
    "P_BRAND": $P_BRAND1,
    "P_TYPE": $P_TYPE1,
    "P_SIZE": $P_SIZE1,
    "PS_SUPPKEY": $PS_SUPPKEY1
  }
)
GROUP BY $P_BRAND := $T2.P_BRAND, $P_TYPE := $T2.P_TYPE, $P_SIZE := $T2.P_SIZE WITH $T2
LET $SUPPLIER_CNT := COUNT(FOR $I IN $T2 RETURN $I.PS_SUPPKEY)
ORDER BY $SUPPLIER_CNT DESC, $P_BRAND, $P_TYPE, $P_SIZE
RETURN {
  "P_BRAND": $P_BRAND,
  "P_TYPE": $P_TYPE,
  "P_SIZE": $P_SIZE,
  "SUPPLIER_CNT": $SUPPLIER_CNT
}
```

Listing A.54: TPC-H Query 16 (in AQL)

```
DECLARE FUNCTION TMP(){
  FOR $L IN DATASET('LINEITEM')
  GROUP BY $L_PARTKEY := $L.L_PARTKEY WITH $L
  RETURN {
    "T_PARTKEY": $L_PARTKEY,
    "T_AVG_QUANTITY": 0.2 * AVG(FOR $I IN $L RETURN $I.L_QUANTITY)
  }
}

SUM(
  FOR $L IN DATASET('LINEITEM')
  FOR $P IN DATASET('PART')
  WHERE  $P.P_PARTKEY = $L.L_PARTKEY
    AND $P.P_BRAND = 'BRAND#23'
    AND $P.P_CONTAINER = 'MED BOX'
  FOR $T IN TMP()
  WHERE $L.L_PARTKEY = $T.T_PARTKEY
    AND $L.L_QUANTITY < $T.T_AVG_QUANTITY
  RETURN $L.L_EXTENDEDPRICE
)/7.0
```

Listing A.55: TPC-H Query 17 (in AQL)

```
FOR $C IN DATASET('CUSTOMER')
FOR $O IN DATASET('ORDERS')
WHERE $C.C_CUSTKEY = $O.O_CUSTKEY
FOR $T IN (
  FOR $L IN DATASET('LINEITEM')
  GROUP BY $L_ORDERKEY := $L.L_ORDERKEY WITH $L
  RETURN {
    "L_ORDERKEY": $L_ORDERKEY,
    "T_SUM_QUANTITY": SUM(FOR $I IN $L RETURN $I.L_QUANTITY)
  }
)
WHERE $O.O_ORDERKEY = $T.L_ORDERKEY AND $T.T_SUM_QUANTITY > 300
FOR $L IN DATASET('LINEITEM')
WHERE $L.L_ORDERKEY = $O.O_ORDERKEY
GROUP BY $C_NAME := $C.C_NAME, $C_CUSTKEY := $C.C_CUSTKEY, $O_ORDERKEY := $O.O_ORDERKEY,
         $O_ORDERDATE := $O.O_ORDERDATE, $O_TOTALPRICE := $O.O_TOTALPRICE WITH $L
ORDER BY $O_TOTALPRICE DESC, $O_ORDERDATE
LIMIT 100
RETURN {
  "C_NAME": $C_NAME,
  "C_CUSTKEY": $C_CUSTKEY,
  "O_ORDERKEY": $O_ORDERKEY,
  "O_ORDERDATE": $O_ORDERDATE,
  "O_TOTALPRICE": $O_TOTALPRICE,
  "SUM_QUANTITY": SUM(FOR $J IN $L RETURN $J.L_QUANTITY)
}
```

Listing A.56: TPC-H Query 18 (in AQL)

```
SET IMPORT-PRIVATE-FUNCTIONS 'TRUE';

DECLARE FUNCTION Q19_TMP() {
  FOR $T IN DATASET LINEITEM
  WHERE
  ($T.L_SHIPMODE = 'AIR' OR $T.L_SHIPMODE = 'AIR REG')
  AND $T.L_SHIPINSTRUCT = 'DELIVER IN PERSON'
  RETURN {
    "LPKEY": $T.L_PARTKEY,
    "QUANTITY": $T.L_QUANTITY,
    "EXTNDPRICE": $T.L_EXTENDEDPRICE,
    "DISCOUNT": $T.L_DISCOUNT
  }
}

SUM(
  FOR $L IN Q19_TMP()
  FOR $P IN DATASET PART
  WHERE $P.P_PARTKEY = $L.LPKEY
    AND ( (
        $P.P_BRAND = 'BRAND#12'
        AND REG-EXP($P.P_CONTAINER,'SM CASE|SM BOX|SM PACK|SM PKG')
        AND $L.QUANTITY >= 1 AND $L.QUANTITY <= 11
        AND $P.P_SIZE >= 1 AND $P.P_SIZE <= 5
      ) OR (
        $P.P_BRAND = 'BRAND#23'
        AND REG-EXP($P.P_CONTAINER, 'MED BAG|MED BOX|MED PKG|MED PACK')
        AND $L.QUANTITY >= 10 AND $L.QUANTITY <= 20
        AND $P.P_SIZE >= 1 AND $P.P_SIZE <= 10
      ) OR (
        $P.P_BRAND = 'BRAND#34'
        AND REG-EXP($P.P_CONTAINER, 'LG CASE|LG BOX|LG PACK|LG PKG')
        AND $L.QUANTITY >= 20 AND $L.QUANTITY <= 30
        AND $P.P_SIZE >= 1 AND $P.P_SIZE <= 15
      )
    )
  RETURN $L.EXTNDPRICE * (1 - $L.DISCOUNT)
)
```

Listing A.57: TPC-H Query 19 (in AQL)

```
FOR $T3 IN (
  FOR $T2 IN (
    FOR $L IN DATASET('LINEITEM')
    WHERE $L.L_SHIPDATE >= '1994-01-01' AND $L.L_SHIPDATE < '1995-01-01'
    GROUP BY $L_PARTKEY:=$L.L_PARTKEY, $L_SUPPKEY:=$L.L_SUPPKEY WITH $L
    RETURN {
      "L_PARTKEY": $L_PARTKEY,
      "L_SUPPKEY": $L_SUPPKEY,
      "SUM_QUANTITY": 0.5 * SUM(FOR $I IN $L RETURN $I.L_QUANTITY)
    }
  )
  FOR $PST1 IN (
    FOR $PS IN DATASET('PARTSUPP')
    FOR $T1 IN (
      FOR $P IN DATASET('PART')
      WHERE LIKE($P.P_NAME, 'FOREST%')
      DISTINCT BY $P.P_PARTKEY
      RETURN { "P_PARTKEY": $P.P_PARTKEY }
    )
    WHERE $PS.PS_PARTKEY = $T1.P_PARTKEY
    RETURN {
      "PS_SUPPKEY": $PS.PS_SUPPKEY,
      "PS_PARTKEY": $PS.PS_PARTKEY,
      "PS_AVAILQTY": $PS.PS_AVAILQTY
    }
  )
  WHERE $PST1.PS_PARTKEY = $T2.L_PARTKEY AND $PST1.PS_SUPPKEY = $T2.L_SUPPKEY
    AND $PST1.PS_AVAILQTY > $T2.SUM_QUANTITY
  DISTINCT BY $PST1.PS_SUPPKEY
  RETURN { "PS_SUPPKEY": $PST1.PS_SUPPKEY }
)
FOR $T4 IN (
  FOR $N IN DATASET('NATION')
  FOR $S IN DATASET('SUPPLIER')
  WHERE  $S.S_NATIONKEY = $N.N_NATIONKEY AND $N.N_NAME = 'CANADA'
  RETURN {
    "S_NAME": $S.S_NAME,
    "S_ADDRESS": $S.S_ADDRESS,
    "S_SUPPKEY": $S.S_SUPPKEY
  }
)
WHERE $T3.PS_SUPPKEY = $T4.S_SUPPKEY
ORDER BY $T4.S_NAME
RETURN {
  "S_NAME": $T4.S_NAME,
  "S_ADDRESS": $T4.S_ADDRESS
}
```

Listing A.58: TPC-H Query 20 (in AQL)

```
DECLARE FUNCTION TMP1() {
  FOR $L2 IN (
    FOR $L IN DATASET('LINEITEM')
    GROUP BY $L_ORDERKEY1 := $L.L_ORDERKEY, $L_SUPPKEY1 := $L.L_SUPPKEY WITH $L
    RETURN {
      "L_ORDERKEY": $L_ORDERKEY1,
      "L_SUPPKEY": $L_SUPPKEY1
    }
  )
  GROUP BY $L_ORDERKEY := $L2.L_ORDERKEY WITH $L2
  RETURN {
    "L_ORDERKEY": $L_ORDERKEY,
    "COUNT_SUPPKEY": COUNT(FOR $I IN $L2 RETURN $I.L_SUPPKEY),
    "MAX_SUPPKEY": MAX(FOR $I IN $L2 RETURN $I.L_SUPPKEY)
  }
}

DECLARE FUNCTION TMP2() {
  FOR $L2 IN (
    FOR $L IN DATASET('LINEITEM')
    GROUP BY $L_ORDERKEY1 := $L.L_ORDERKEY, $L_SUPPKEY1 := $L.L_SUPPKEY WITH $L
    RETURN {
      "L_ORDERKEY": $L_ORDERKEY1,
      "L_SUPPKEY": $L_SUPPKEY1,
      "COUNT": COUNT(FOR $I IN $L RETURN $I.L_SUPPKEY)
    }
  )
  FOR $L3 IN (
    FOR $L IN DATASET('LINEITEM')
    WHERE $L.L_RECEIPTDATE <= $L.L_COMMITDATE
    GROUP BY $L_ORDERKEY1 := $L.L_ORDERKEY, $L_SUPPKEY1 := $L.L_SUPPKEY WITH $L
    RETURN {
      "L_ORDERKEY": $L_ORDERKEY1,
      "L_SUPPKEY": $L_SUPPKEY1,
      "COUNT": COUNT(FOR $I IN $L RETURN $I.L_SUPPKEY)
    }
  )
  WHERE $L2.L_ORDERKEY = $L3.L_ORDERKEY
    AND $L2.L_SUPPKEY = $L3.L_SUPPKEY
    AND $L2.COUNT = $L3.COUNT
  GROUP BY $L_ORDERKEY := $L2.L_ORDERKEY WITH $L2
  RETURN {
    "L_ORDERKEY": $L_ORDERKEY,
    "COUNT_SUPPKEY": COUNT(FOR $I IN $L2 RETURN $I.L_SUPPKEY),
    "MAX_SUPPKEY": MAX(FOR $I IN $L2 RETURN $I.L_SUPPKEY)
  }
}
```

Listing A.59: TPC-H Query 21 (in AQL) - Part 1

```
FOR $T4 IN (
  FOR $T3 IN (
    FOR $L IN DATASET('LINEITEM')
    FOR $NS IN (
      FOR $N IN DATASET('NATION')
      FOR $S IN DATASET('SUPPLIER')
      WHERE $S.S_NATIONKEY = $N.N_NATIONKEY
        AND $N.N_NAME = 'SAUDI ARABIA'
      RETURN {
        "S_NAME": $S.S_NAME,
        "S_SUPPKEY": $S.S_SUPPKEY
      }
    )
    WHERE $NS.S_SUPPKEY = $L.L_SUPPKEY AND $L.L_RECEIPTDATE > $L.L_COMMITDATE
    FOR $O IN DATASET('ORDERS')
    WHERE $O.O_ORDERKEY = $L.L_ORDERKEY
      AND $O.O_ORDERSTATUS = 'F'
    FOR $T1 IN TMP1()
    WHERE $L.L_ORDERKEY = $T1.L_ORDERKEY
      AND $T1.COUNT_SUPPKEY > 1
    RETURN {
      "S_NAME": $NS.S_NAME,
      "L_ORDERKEY": $T1.L_ORDERKEY,
      "L_SUPPKEY": $L.L_SUPPKEY,
      "T1_COUNT_SUPPKEY": $T1.COUNT_SUPPKEY
    }
  )
  FOR $T2 IN TMP2()
  WHERE $T3.L_ORDERKEY = $T2.L_ORDERKEY
    AND $T2.COUNT_SUPPKEY = $T3.T1_COUNT_SUPPKEY - 1
  RETURN {
    "S_NAME": $T3.S_NAME,
    "L_SUPPKEY": $T3.L_SUPPKEY,
    "L_ORDERKEY": $T2.L_ORDERKEY,
    "COUNT_SUPPKEY": $T2.COUNT_SUPPKEY,
    "MAX_SUPPKEY": $T2.MAX_SUPPKEY
  }
)
GROUP BY $S_NAME := $T4.S_NAME WITH $T4
LET $NUMWAIT := COUNT($T4)
ORDER BY $NUMWAIT DESC, $S_NAME
LIMIT 100
RETURN {
  "S_NAME": $S_NAME,
  "NUMWAIT": $NUMWAIT
}
```

Listing A.59: TPC-H Query 21 (in AQL) - Part 2

```
DECLARE FUNCTION Q22_CUSTOMER_TMP() {
  FOR $C IN DATASET('CUSTOMER')
  LET $PHONE_SUBSTR := SUBSTRING($C.C_PHONE, 1, 2)
  WHERE $PHONE_SUBSTR = '13'
    OR $PHONE_SUBSTR = '31'
    OR $PHONE_SUBSTR = '23'
    OR $PHONE_SUBSTR = '29'
    OR $PHONE_SUBSTR = '30'
    OR $PHONE_SUBSTR = '18'
    OR $PHONE_SUBSTR = '17'
  RETURN {
    "C_ACCTBAL": $C.C_ACCTBAL,
    "C_CUSTKEY": $C.C_CUSTKEY,
    "CNTRYCODE": $PHONE_SUBSTR
  }
}
LET $AVG := AVG(
  FOR $C IN DATASET('CUSTOMER')
  LET $PHONE_SUBSTR := SUBSTRING($C.C_PHONE, 1, 2)
  WHERE $C.C_ACCTBAL > 0.00
    AND ($PHONE_SUBSTR = '13'
    OR $PHONE_SUBSTR = '31'
    OR $PHONE_SUBSTR = '23'
    OR $PHONE_SUBSTR = '29'
    OR $PHONE_SUBSTR = '30'
    OR $PHONE_SUBSTR = '18'
    OR $PHONE_SUBSTR = '17')
  RETURN $C.C_ACCTBAL
)
FOR $CT IN Q22_CUSTOMER_TMP()
WHERE $CT.C_ACCTBAL > $AVG
  AND COUNT(FOR $O IN DATASET('ORDERS') WHERE $CT.C_CUSTKEY = $O.O_CUSTKEY RETURN $O) = 0
GROUP BY $CNTRYCODE := $CT.CNTRYCODE WITH $CT
ORDER BY $CNTRYCODE
RETURN {
  "CNTRYCODE": $CNTRYCODE,
  "NUMCUST": COUNT($CT),
  "TOTACCTBAL": SUM(FOR $I IN $CT RETURN $I.C_ACCTBAL)
}
```

Listing A.60: TPC-H Query 22 (in AQL)