HIGH-PERFORMANCE CLOUD DATA MANAGEMENT

by

Avrilia Floratou

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 07/09/2013

The dissertation is approved by the following members of the Final Oral Committee:
Jignesh M. Patel, Professor, Computer Sciences
Jeffrey F. Naughton, Professor, Computer Sciences
AnHai Doan, Associate Professor, Computer Sciences
David J. DeWitt, Emeritus Professor, Computer Sciences
Jonathan Eckhardt, Associate Professor, Management and Human Resources

To my parents, Yorgos and Dionysia

Acknowledgments

The development of this thesis is the result of the support, help and advice that I have received from many different people. Without them this dissertation wouldn't have been possible.

Firstly, I'd like to thank my parents, Yorgos and Dionysia and my sister, Revekka for being so supportive and caring all these years. This thesis is the result of the great interest that my parents have shown in my education since I was a child, and of the countless efforts they have been making, to inspire me to work hard, to set goals and try to achieve them and to also accept failure. Words cannot describe how thankful I am to them.

Here at Wisconsin, I've had the great pleasure to work with my advisor Jignesh Patel. Jignesh is very passionate about his work and knows how to transmit his enthusiasm to his students. He allows the students to explore their ideas and at the same time he is always willing to guide them and provide advice. I'm also deeply grateful to him for being so supportive, honest and patient with me when I was facing problems or when I was simply complaining about minor things. Jignesh is an advisor that never stops caring about his students. Graduate school would have been much harder without his endless help and support. I feel very fortunate to have been one of his students.

I'm also extremely thankful to David DeWitt. David gave me the opportunity to work in the Microsoft Jim Gray Systems Lab and to also collaborate with him and other Microsoft employees. It has been a great honor for me to have been mentored by him. Along with being a very influential computer scientist, David is also one of the most caring persons I've met and one of the few persons that truly show consideration for all the people that work with them.

Jeff Naughton, AnHai Doan and Chris Ré deserve special thanks for providing useful feedback in my work and for helping me improve my presentation skills. I'm also very thankful to Jon Eckhardt for taking the time to be a member of my defense committee.

This dissertation wouldn't have been possible without the help I received from Sandeep Tata. Sandeep is definitely a very remarkable person. He spent lots of time time guiding me while working with him during my internship at IBM, and also encouraged me and helped me during the following years. He taught me how to work more efficiently and also how to make the best out of collaboration. I feel very lucky to have met him.

I'd also like to thank all the employees at the Microsoft Jim Gray Systems Lab for having always been willing to answer my questions and for the very useful discussions I had with many of them. I'd like to especially thank Nikhil Teletia for working with me in parts of my thesis. It was very helpful to have someone complaining with me when the results didn't seem correct or when the machines were failing. Rimma Nehme deserves special thanks not only for the discussions we had together about research, but also for sharing her experiences about having a family and working in computer science at the same time and for the pleasant evenings I have spent with her family.

I'm very grateful to all my friends and colleagues in the database group for sharing their experiences about research with me and making it easy to look forward to go to the lab every morning. Craig, Jae, Ian, Park, Spyros and Yinan are some of my colleagues who made graduate school easier. I'd also like to thank Jessie for the many lunches we have shared together and for allowing me to practice on her car before getting my driver's license.

I'm very thankful to Willis for the great dinners we've had together, for patiently giving me a ride after work almost every day, and for making me laugh with his great sense of humor. Work wouldn't have been so much fun without him.

Many other people I met in Madison, have also contributed in making my stay enjoyable. I'd like to thank Charalambos, Mina, Yorgos, Loizos, Aris, Ioanna, Joanne, Katerina, Nikos and Dimitris for the many pleasant evenings we have spent together and for being so nice to me. I'm very thankful to Haris for answering all the questions I had about graduate school, life in Madison and California and for having always been willing to help me when needed.

Yannis Ioannidis deserves special thanks for having been a great advisor during my undegraduate studies in Greece, for encouraging me to pursue a Ph.D in computer science in the United States and for still being interested in me and my work. Without his encouragement, I would have never decided to leave from Greece.

The cold winters of Madison would have been unbearable without the cheerful chats I had with my two best friends from Greece. I'm very thankful to Dimitra and Maria for having always been available to talk for hours every weekend over the last 5 years. I will never forget the birthday cake surprise they made to me with Willis' help.

Last but not least, I'd like to thank George for having always been by my side during the last 6 years even though we have been living apart. I'm looking forward to spending a lifetime with him.

Contents

List of Tables

List of Figures

Abstract

Over the past decade, as industry in nearly every sector of the economy has moved to a data-driven world, there has been an explosion in the volume of data and the need for richer and more flexible data processing tools and environments. Processing and analyzing large volumes of data has become easier over the last few years, and today there are many distributed data processing tools and cloud computing platforms that can be readily used to analyze large databases. In other words, users now have access to a variety of open-source tools for "big data" management, tailored for different types of workloads. Most of these systems can be easily deployed to large clusters and can scale to thousands of nodes. Moreover, users do not need to purchase expensive hardware to deploy their applications. There are a variety of cloud computing platforms, adjusted for different types of workloads and hardware requirements, which further simplify the process of deploying, managing and controlling a large-scale application. At the same time, the migration to a cloud-based environment can potentially reduce the end-to-end cost for the user. As a result, users are increasingly leaning towards cloud environments for data-centric applications.

Given the increasing number of available data management solutions and the current computing trend towards cloud-based systems, the work in this dissertation focuses on studying the design of different types of data processing systems that can be deployed in cloud-based environments, identifying its effects on workload performance, and proposing new ways to improve it.

This thesis has two main parts. In the first part of this dissertation, we study the design choices made by the developers of a new class of systems, called NoSQL systems, which have emerged over the last few years. These systems are competing with the traditional and well-studied relational data processing systems (SQL systems) in almost every type of data processing application. We identify some of the performance bottlenecks in these systems and propose new ways to improve them. The second part of this thesis focuses on cloud-based Database-as-a-Service (DaaS) environments. In these environments users execute their workloads on a relational database management system that is deployed in the cloud. DaaS customers typically have certain performance and availability expectations when running their workloads. In this work, we investigate the database placement mechanisms which will allow to satisfy both the performance and the availability requirements of the users and at the same time will reduce the total operating cost for the DaaS provider.

Chapter 1

Introduction

Over the past decade, there has been an explosion in the volume of data and the need for richer and more flexible data processing tools and environments. Users now have access to a variety of open-source tools for "big data" management as well as to an increasing number of available cloud computing platforms, which simplify the process of managing and deploying a data processing application. The available data processing systems are optimized for a variety of workloads, namely analytical and/or transactional workloads, and can address a variety of users' needs . Similarly, the existing cloud computing platforms which are adjusted for different types of hardware and software requirements, can address almost all users' expectations. However, as the volume of data that needs to be processed continuously increases and as more and more databases and applications migrate to cloud-based systems, it becomes challenging for both the cloud service providers and the data processing systems designers to satisfy the performance needs of the users, and at the same time continue to scale their systems to larger cluster sizes.

Given the current computing trend towards cloud-based systems, this thesis focuses on **studying the design of different types of data processing systems that can be deployed in cloud-based environments, iden-**

tify its effects on workload performance and proposing new ways to improve it. More specifically, the work presented in this thesis can be divided into the following two areas.

1.1 Big Data Platforms

In this part of the dissertation, we study the design choices made by the developers of a new class of systems, called NoSQL systems, which have emerged over the last few years. These systems are competing with the traditional and well-studied relational data processing systems (SQL systems) in almost every type of data processing application. The newer NoSQL systems have made different design choices than the traditional RDBMSs and constitute a tempting choice when deploying large-scale data applications, mainly due to the fact that they are open-source, they can easily be deployed to large clusters of commodity hardware and they incorporate mechanisms to deal with hardware/software failures. This part of the thesis focuses on evaluating and analyzing the performance of SQL and NoSQL alternatives on different types of workloads and identifying the drawbacks and the advantages of each of this class of systems. More specifically, we evaluate these systems on two major types of applications, namely decision support analysis and interactive data-serving.

In interactive data-serving environments, consumer-facing artifacts must be computed on-the-fly from a database. An example of such an application is a multi-player game in which the objects to be displayed in the next screen must be assembled on-the-fly. In such environments, newer NoSQL document systems are popular alternatives to using an RDBMS. Our study reveals, that RDBMSs can still provide better performance, in terms of both latency and throughput, for this type of workload. This finding comes in contrast with the widely held belief that relational databases might be too heavy weight for this type of workload, where the requests consist of a single simple operation and do not require the complex transactional semantics that RDBMSs can handle.

At the other end of the big data application spectrum are analytical decision support workloads that are characterized by complex queries on massive amounts of data. Once again, (parallel) RDBMSs were largely the only solution for these applications just a few years ago, but now they face competition from another new class of NoSQL systems – namely, systems based on the MapReduce paradigm [48]. Our analysis reveals that although, the NoSQL system we tested, has greatly improved over the last few years, its performance still lags behind the traditional RDBMS. More specifically, we identify drawbacks in the storage layer, lack of cost-based optimization techniques that would greatly improve the performance of some class of queries, and failure to efficiently use mechanisms that are already incorporated in the NoSQL system, which, if used properly, have the potential to significantly improve performance. These findings are presented in detail in Chapter 2 of this dissertation.

As identified in this study, an inefficient data storage layout can severely impact the performance of data-analysis workloads in MapReduce-based systems. Again, the well-studied RDBMSs have faced similar problems and have tackled them by introducing columnar storage layouts. These layouts have proven to be very efficient for queries that access a few attributes of a dataset. Chapter 3, presents our work on incorporating columnar storage in MapReduce. Incorporating column-oriented storage in a MapReduce implementation such as Hadoop [3], presents unique challenges that can lead to new design choices. More specifically, in this chapter, we investigate the tradeoffs between different data serialization formats, the overheads of deserialization and decompression, and the importance of data colocation. Moreover, we introduce a dictionary-based compression technique which in combination with a novel columnar skip-list storage layout can improve MapReduce's performance by an order of magnitude.

1.2 Database-as-a-Service Environments

The last part of this thesis focuses on cloud-based Database-as-a-Service (DaaS) environments only. In these environments users execute their workloads on a relational database management system (RDBMS) that is deployed in the cloud. DaaS customers typically have certain performance and availability expectations when running their workloads and they also want to minimize the cost incurred when running their workloads. Guaranteeing such requirements is a challenging task for the DaaS providers. Failure to satisfy the users' needs directly translates into financial losses for the service provider.

One way to increase data availability is through database replication. Assigning replicas with different resource requirements to machines while meeting the replication constraints and satisfying the performance SLOs is a challenging task. The problem becomes even more complicated when the impact of the replica placement algorithm to the data center's operating cost is taken into consideration. Typically, DaaS providers aim to satisfy the performance and availability SLOs while minimizing the total operating cost. In general, aggressively packing replicas on each machine reduces the operating costs but degrades performance for the tenants, and vice versa. In this thesis, we present a framework for designing and evaluating *online* replica placement algorithms for multi-tenant DaaS environments. We also design a number of replica placement algorithms and evaluate them using multiple objective criteria that are established by our framework. We find that an algorithm called RkC, has a unique combination of being simple to implement, has good load balancing properties both in the initial replica placement and in dealing with load-related changes following a node failure, has low initial hardware provisioning cost, and is able to guarantee tenant performance SLOs without requiring global knowledge of every machine's current load. Chapter 4 discusses in detail our proposed techniques.

Regarding, the cost of running a workload in a Database-as-a-Service environment, this cost depends on the pricing model of the cloud service. DaaS pricing models are typically “pay-as-you-go” in which the customer is charged based on resource usage such as CPU and memory utilization. Thus, customers considering different DaaS options must take into account how the performance and efficiency of the DaaS will ultimately impact their monthly bill. In our work, we show that the current DaaS model can produce unpleasant surprises – for example, the case study that we present in this paper illustrates a scenario in which a DaaS service powered by a DBMS that has a lower hourly rate actually costs more to the end user than a DaaS service that is powered by another DBMS that charges a higher hourly rate. In Chapter 5, we present our evaluation on the DaaS pricing models using two different database systems and a variety of workloads. We also propose Benchmark-as-a-Service, which is a method for the end-user to get an accurate estimate of the true costs that will be incurred without worrying about the nuances of how the DaaS operates.

1.3 Outline

The remainder of this dissertation is organized as follows: Chapter 2 presents our experimental study on comparing SQL and NoSQL systems on different types of workloads. Chapter 3 discusses our work on incorporating column-oriented storage in the MapReduce framework. Chapter 4 presents our work on developing replica placement algorithms for Database-as-a-Service environments. Chapter 5 presents our evaluation of the current DaaS pricing models. Finally, Chapter 6 concludes this dissertation and points to some directions for future work.

Chapter 2

A Comparison of Data Processing Systems for Cloud Environments

2.1 Introduction

The database community is currently at an unprecedented and exciting inflection point. On one hand, the need for data processing products has never been higher than the current level, and on the other hand, the number of new data management solutions that are available has exploded over the past decade. For over four decades, data management typically meant relational data processing, and relational database management systems (RDBMSs) became commonplace in any serious data processing environment. Over the past decade, as industry in nearly every sector of the economy has moved to a data-driven world, there has been an explosion in the volume of data, and the need for richer and more flexible data processing tools.

RDBMSs are no longer the only viable alternative for data-driven applications. First, consider applications in interactive data-serving environments, where consumer-facing artifacts must be computed on-the-fly from a database. Examples of applications in this class include social networks where a consumer-facing web page must be assembled on-the-fly, or a multi-player game in which the objects to be displayed in the next screen must be assembled on-the-fly. Just a few years ago, the standard way to run such applications was to use an RDBMS for the data management component. Now, in such environments, newer NoSQL document systems, such as MongoDB [11], CouchDB [2], Riak [18], etc., are popular alternatives to using an RDBMS. These new NoSQL systems are often designed to have a simpler key-value based data model (in contrast to the relational data model), and are designed to work seamlessly in cluster environments. Thus, many of these systems have in-built “sharding” or partitioning primitives that split large data sets across multiple nodes and keep the shards balanced as new records and/or nodes are added to the system.

This new interactive data-serving domain is largely characterized by queries that read or update a very small amount of the entire dataset. In some sense, one can think of this class of applications as the “new OLTP” domain, bearing resemblance to the traditional OLTP world in which the workload largely consists of short “bullet” queries.

At the other end of the big data application spectrum are analytical decision support workloads that are characterized by complex queries on massive amounts of data. The need for these analytical data processing systems has also been growing rapidly. Once again, (parallel) RDBMSs were largely the only solution for these applications just a few years ago, but now they face competition from another new class of NoSQL systems - namely, systems based on the MapReduce paradigm such as Hive on Hadoop. These NoSQL systems are tailored for large-scale analytics, and are designed specifically to run on clusters of commodity hardware. They assume that hardware/software failures are common, and incorporate mechanisms to deal with such failures. These systems typically also scale easily when adding or removing nodes to an operational cluster.

The question that we ask in this chapter is: How does the performance of RDBMs solutions compare to the NoSQL systems for the two classes of workloads described above, namely interactive data-serving environments and decision support systems (DSS)? While [79] examined some aspect of this question, it focused only on a small number of simple DSS queries (selected join and aggregate queries). Furthermore, it only considered MapReduce (MR) as an alternative to RDBMSs, and did not consider a more sophisticated MR query processing system like Hive [5]. Furthermore, it has been several years since that effort, and the NoSQL systems have evolved significantly since that time. So, it is interesting to ask how the performance of the NoSQL systems compares to that of parallel RDBMSs today.

In this thesis, we present results comparing SQL Server and MongoDB using the YCSB benchmark [43] to characterize how these two SQL and NoSQL systems compare on interactive data-serving environments. We also present results comparing Hive and a parallel version of SQL Server, called PDW, using the TPC-H DSS benchmark [21]. Our results show that the SQL systems currently still have significant performance advantages over both classes of NoSQL systems, but these NoSQL systems are fairly competitive in many cases. The SQL systems will need to continue to keep up their performance advantages and potentially also need to expand their functionality (for example, supporting automatic sharding and a more flexible data model such as JSON) to continue to be competitive.

On a cautionary note, we acknowledge that the evaluation in this chapter only considers one data point/system in each class that is considered in this work, namely a) NoSQL interactive data-serving systems (we use MongoDB), b) MapReduce-based DSS systems (Hive), and c) RDBMS systems (SQL Server and SQL Server PDW). We understand that using other systems in each of these classes may produce different comparative results, and we hope future studies will expand this work to include other systems.

In this thesis, we have (arguably) taken one representative and leading system in each class, and benchmarked these systems against each other to gather an initial understanding of the emerging big data landscape.

Finally, we note that while we have used some common benchmarks in this thesis, the results presented are not audited or official results, and, in fact, were not run in a way that meets all of the benchmark requirements. The results are shown for the sole purpose of providing relative comparisons for this work, and should not be compared to official benchmark results.

2.2 Background

In this section, we describe some background about the different data processing systems that we examine in this thesis.

2.2.1 Parallel Data Warehouse (PDW)

SQL Server PDW [10] is a classic shared-nothing parallel database system from Microsoft that is built on top of SQL Server. PDW consists of multiple compute nodes, a single control node and other administrative service nodes. Each compute node is a separate server running SQL Server. The data is horizontally partitioned across the compute nodes. The control node is responsible for handling the user query and generating an optimized plan of parallel operations. The control node distributes the parallel operations to the compute nodes where the actual data resides. A special module running on each compute node called the Data Movement Service (DMS) is responsible for shuffling data between compute nodes as necessary to execute relational operations in parallel. When the compute nodes are finished, the control node handles post-processing and re-integration of results sets for delivery back to the users.

2.2.2 Hive

Hive [5] is an open-source data warehouse built on top of Hadoop [3]. It provides a structured data model for data that is stored in the Hadoop Distributed Filesystem (HDFS), and a SQL-like declarative query language called HiveQL. Hive converts HiveQL queries to a directed acyclic graph of MapReduce jobs, and thus saves the user from having to write the more complex MapReduce jobs directly.

Data organization in Hive is similar to that found in relational datab ses. Starting from a coarser granularity, data is stored in databases, tables, partitions and buckets. More details about the data layout in Hive are provided in Section 2.3. Finally, Hive has support for multiple data storage formats including text files, sequence files, and RCFiles [60]. Users can also create custom storage formats as well as serializers/deserializers, and plug them into the system.

2.2.3 MongoDB

MongoDB [11] is a popular open-source NoSQL database. Some of its features are a document-oriented storage layer, indexing in the form of B-trees, auto-sharding and asynchronous replication of data between servers. In MongoDB data is stored in *collections* and each collection contains *documents*. Collections and documents are loosely analogous to tables and records, respectively, found in relational databases. Each document is serialized using BSON. MongoDB does not require a rigid schema for the documents. Specifically, documents in the same collection can have different structures.

Another important feature of MongoDB is its support for auto-sharding. With sharding, data is partitioned amongst multiple nodes in an order-preserving manner. Sharding is similar to the horizontal partitioning technique that is used in parallel database systems. This feature enables horizontal scaling across multiple nodes. When some nodes contain a disproportionate amount of data compared to the other nodes in the cluster, MongoDB redistributes the data automatically so that the load is equally distributed across the nodes/shards.

Finally, MongoDB supports failover via replica sets, which is its mechanism for implementing asynchronous master/slave replication. A replica set consists of two or more nodes that are copies of each other. More information about the semantics of replica sets can be found in [12]. In the following sections, we use the name **Mongo-AS** (MongoDB with auto-sharding) when referring to the original MongoDB implementation.

2.2.4 Client-side Sharded SQL Server and MongoDB

For our experiments, we created a SQL Server implementation (**SQL-CS**) that uses client-side hashing to determine the home node/shard for each record by modifying the client-side application that runs the YCSB benchmark. We implemented this client-side sharding so that we could compare MongoDB(-AS) with SQL Server in a cluster environment. We also took the client-side sharding code and implemented it on top of MongoDB. This implementation of client-side sharding on MongoDB is denoted as **MongoDB-CS**, allowing us to compare MongoDB-AS with MongoDB-CS (and SQL-CS).

We note that both SQL-CS and Mongo-CS do not support some of the features that are supported by Mongo-AS. First, whereas Mongo-AS uses a form of range partitioning to distribute the records across the shards, the Mongo-CS and SQL-CS implementations both use hash partitioning. Another difference is that the Mongo-CS implementation does not use any of the routing (mongos), configuration (config db), and balancer processes that are part of Mongo-AS. As a result, load balancing cannot happen automatically as in Mongo-AS, where the auto-sharding mechanism aims to continually balance the load across all the nodes in the cluster. However, Mongo-CS makes use of the basic “mongo” process, which is responsible for processing the client’s requests. Finally, Mongo-CS and SQL-CS do not support automatic failover. We note that these features listed above were not the key subject of performance testing in the benchmark (YCSB) that we use in this thesis.

On the flip side, we also note that SQL Server has many features that are not supported in MongoDB. For example, MongoDB has a flexible data model that makes it far easier to deal with schema changes. MongoDB also supports read/write atomic operations on single data entities, whereas SQL Server provides full ACID semantics and multiple isolation levels. SQL Server also has better manageability and performance analysis tools (e.g. database tuning advisor).

2.3 Evaluation

In this section, we present an experimental evaluation of a RDBMS and a NoSQL system on a DSS and a “modern” OLTP workload. More specifically, we use TPC-H [21] to evaluate Microsoft’s Parallel Data Warehouse and Hive. We also compare MongoDB (Mongo-AS) with both client-side sharded Microsoft SQL Server (SQL-CS) and MongoDB (Mongo-CS) implementations, using the YCSB benchmark. The following sections present details about the hardware and the software configuration that is used in our experiments.

2.3.1 Hardware Configuration

All experiments were run on a cluster of 16 nodes connected by 1Gbit HP Procurve 2510G 48 (J9280A) Ethernet switch. Each node has dual Intel Xeon L5630 quad-core processors running at 2.13 GHz, 32 GB of main memory, and 10 SAS 10K RPM 300GB hard drives. One of the hard drives is always reserved for the operating system.

When evaluating PDW and Hive, we used eight disks to store the data. These disks were organized as one RAID 0 volume when the system was running Hive, and configured as separate logical volumes when running PDW. The log data for each PDW node was stored on a separate hard disk.

For Hive, we used one extra node to run the namenode and the jobtracker processes only. PDW needs two extra nodes, used as a control node and as a landing node respectively. The landing node is responsible for data loading and does not participate in query execution. All the extra nodes were connected to the same Ethernet switch that is used by the remaining 16 nodes in the cluster. The operating system was Windows Server 2008 R2 when running PDW, and Ubuntu 11.04 when running Hive.

For the YCSB benchmark experiments eight nodes were used as servers (running SQL or Mongo) and eight were used to run the client benchmark. Similar to the DSS experimental setting, eight disks were used to store the data for the OLTP experiments. These disks were configured as RAID 0 when running MongoDB, and were treated as separate logical volumes when running SQL Server. For the Mongo-AS experiments, we used one extra node as the“config” server. The“config” server keeps metadata about the cluster's state. The operating system was, in both cases, Windows Server 2008 R2.

2.3.2 Software Configuration

In this section, we describe the software configuration for each system that we tested.

Hive and Hadoop

We used Hive version 0.7.1 running on Hadoop version 0.20.203. We configured Hadoop to run 8 map tasks and 8 reduce tasks per node (a total of 128 map slots and 128 reduce slots). The maximum JVM heap size was set to 2GB per task. We used a 256 MB HDFS block size, and the HDFS replication factor was set to 3. The TPC-H Hive scripts are available online [19]. However, since Hive now supports features that were not available when these scripts were written, we modified the scripts in the following ways:

1. Instead of using text files to store the data, we used the RCFile format [60]. The RCFile layout is considered to be faster than a row-store format (e.g. text file, sequence file) since it can eliminate some I/O operations [60]. All the TPC-H base tables are stored in compressed (GZIP) RCFile format. Some TPC-H queries were split manually (by the Hive team) into smaller sub-queries, since HiveQL is not expressive enough to support the full SQL-92 specification; the output of these intermediate queries is also stored in the RCFile format.

2. We enabled the map-side aggregation, map-side join and bucketed map-join features of Hive, which usually improves performance by avoiding executing the reduce phase of a MapReduce job.

3. We set the number of reducers for each MapReduce job to the total number of reduce slots in the cluster (128 reducers). We found that this setting significantly improves the performance of Hive when running the TPC-H benchmark, since all the reducers can now complete in one reduce round.

Finally, all the results produced by the map tasks are compressed using LZO, according to the suggestions of the Hive team [19] for appropriate setting of Hive when running TPC-H.

PDW

For our experiments we used a pre-release version (November 2011) of PDW AU3. Each compute node runs SQL Server 2008 configured to use a maximum of 24GB of memory for its buffer pool. Each compute node contains 8 horizontal data partitions (a total of 128 partitions across the cluster).

SQL Server PDW is only sold as an appliance. In an appliance each node is configured much larger amounts of memory and storage and the nodes are interconnected using Infiniband, and not Ethernet. Hence, the results a customer would see would be much faster than what we report below for an appliance with a similar number of nodes. Since, we wanted to avoid an apples-to-oranges comparison between PDW and Hive we used exactly the same hardware for both systems.

MongoDB (Mongo-AS)

We used MongoDB version 1.8.2. MongoDB supports auto-sharding so that it can scale horizontally across multiple nodes. In our configuration, the data is spread across 128 shards. We ran 16 “mongo” processes on each one of our 8 server machines. Each “mongod” process is responsible for one shard.

In MongoDB (version 1.8.2) any number of concurrent read operations are allowed, but a write operation can block all other operations. That is because MongoDB uses a global lock for writes (there is one such lock per “mongod” process) . Consequently, we chose to run 16 processes per server node instead of one. In this way, we can exploit the fact that our nodes have 16 cores (hyper-threaded) and, at the same time, increase the concurrency when the workload contains inserts or updates. Our single node experiments have shown that running 16 processes per machine has better performance than running one or eight processes when using the YCSB benchmark. Except for the “config db” and “mongod” processes, we launched 8 “mongos” processes, one at each server machine. The “mongos” process is responsible for routing client requests to the appropriate “mongod” instance. All the clients that run on the same client node connect to the same "mongos" process. Since we have 8 client nodes, there is a "1-1" correspondence between the "mongos" processes and the client nodes.

MongoDB supports failover by using a form of asynchronous master/slave replication, called replica sets. For our experiments, we did not create any replica sets.

2.3.3 Traditional DSS Workload: Hive vs. PDW

In this section we describe the DSS TPC-H workload and various parameters related to this workload for both PDW and Hive.

Workload Description

We used TPC-H at four scale factors (250 GB, 1000 GB, 4000 GB,16000 GB) to evaluate the performance of PDW and Hive. These four scale factors represent cases where different portions of the TPC-H tables fit in main memory. We noticed that the TPC-H generator does not produce correct results at the 16000 scale factor (this scale factor cannot be reported in the official benchmark results). More specifically, the values generated for the `partkey` and `custkey` fields in the `mk_order` function are negative numbers. These numbers are produced using the RANDOM function, which overflows at the 16TB scale. Hence, we modified the generator code to use a 64-bit random number generator (RANDOM64). For all the scale factors, we executed the 22 TPC-H queries that are included in the benchmark, sequentially. We didn't execute the two TPC-H refresh functions, because the Hive version that we used, does not support deletes and inserts into existing tables or partitions (the newer Hive versions 0.8.0 and 0.8.1 do support INSERT INTO statements).

Data Layout

A Hive table can contain *partitions* and/or *buckets*. In Hive, each partition corresponds to one HDFS directory and contains all the records of the table that have the same value on the partitioning attributes. Selection queries on the partitioning columns can benefit from this layout since only the necessary partitions are scanned instead of the whole table.

A Hive table can also consist of a number of buckets. A bucket is stored in a file within the partition's or table's directory depending on whether the table is a partitioned table or not. The user provides a bucketing column as well as the number of buckets that should be created for the table. Hive determines the bucket number for each row of the table by hashing on the value of the bucketing column. Each bucket may contain rows with different values on the bucketing column. During a join, if the tables involved are bucketed on the join column, and the buckets are a multiple of each other, the buckets can be joined with each other in a map-side join only.

As seen in Table 2.1, a Hive table can contain both partitions and buckets (e.g. Customer table in Table 2.1). In this case the table consists of a set of directories (one for each partition). Each directory contains a set of files, each one corresponding to one bucket. Hive tables can also be only partitioned or only bucketed (e.g. Lineitem table in Table 2.1). In the first case, selection queries on the partitioning columns can benefit from the layout. However, join queries cannot benefit unless there is a predicate on the partitioning columns on at least one of the tables. Bucketed tables can help improve the performance of joins but cannot improve selection queries even if there is a selection predicate on the bucketing column.

In PDW, a table can be either horizontally partitioned or replicated across all the nodes. When partitioned, the records are distributed to the partitions using a hash function on a partition column. Table 2.1 summarizes the data layouts for Hive and PDW.

	Hive		**PDW**	
Table	**Partition Column**	**Buckets**	**Partition Column**	**Replication**
Customer	c_nationkey	8 buckets per partition on c_custkey	c_custkey	No
Lineitem	-	512 buckets on l_orderkey	l_orderkey	No
Nation	-	-	-	Yes
Orders	-	512 buckets on o_orderkey	o_orderkey	No
Part	-	8 buckets on p_partkey	p_partkey	No
Partsupp	-	8 buckets on ps_partkey	ps_partkey	No
Region	-	-	-	Yes
Supplier	s_nationkey	8 buckets per partition on s_suppkey	s_suppkey	No

Table 2.1: Data Layout in Hive and PDW.

As can be seen in Table 2.1, the bucket columns used in Hive are the same as those used to horizontally partition the PDW tables. Each bucket is also sorted on the corresponding bucket column. The PDW tables consist of 128 partitions (8 data "distribution" per node).

Previous work [79] has shown that one of the major reasons why relational databases outperform Hadoop on some workloads is their inherent indexing support. For our experiments we decided not to use any type of index for the PDW tables (including primary key indices). The reason behind this decision is that the Hive version we used does not support automatic generation of query plans that consider the available indices. Instead, the user has to rewrite the query so that it takes into account the appropriate indices. This process quickly becomes complicated with complex queries like those in the TPC-H benchmark since the user has to manually produce the "optimal" query plan. The newer versions of Hive, have improved their index support and there is an ongoing effort on seamlessly integrating indexing in Hive (e.g. [6, 7]). As part of future work, we plan on comparing the performance of PDW with Hive, once Hive's optimizer starts considering indices.

Data Preparation and Load Times

In this section, we describe the data preparation steps for each system. We also present data loading time for each system.

For Hive, we generated the TPC-H dataset in parallel across the 16 nodes of our cluster using the TPC "dbgen" program. All the data is stored on a separate hard disk that is not used to store HDFS data files. Before starting the loading process, we created one Hive table for each TPC-H table. In the table definition, we provide the schema of the table, the partitioning and bucketing columns (if applicable) and the storage format (RCFile).

Data is loaded into Hive using two phases. First, the TPC-H data files are loaded on each node in parallel, directly into HDFS, as plain text using the HDFS command-line utility that copies data from the local filesystem to HDFS. For each TPC-H table, an external Hive table is then created. The table points to a directory in HDFS that contains all the relevant data for this table. In the next phase, the data is converted from the text format into the compressed RCFile format. This conversion is done using a Hive query that selects all the tuples of each external table and inserts them into the corresponding Hive table.

When loading into PDW, the TPC-H data is generated on the landing node. Before loading the data, the necessary TPC-H tables are created by using the CREATE TABLE statement and specifying the schema and distribution of the tables (replicated or hash-distributed). The generated

	Load Time (in minutes)			
	250 GB	**1 TB**	**4 TB**	**16 TB**
HIVE	38	125	519	2512
PDW	79	313	1180	4712

Table 2.2: Load Times for Hive and PDW.

data is loaded using the "dwloader" utility of PDW, which splits the text files that are generated at the landing node, into multiple chunks. These chunks are then loaded to the 16 compute nodes of the cluster in parallel. Table 2.2 presents the data loading times for both systems.

Experimental Evaluation

In this section we present an analysis of the performance and scalability aspects of Hive and PDW when running TPC-H.

Table 2.3 presents the running time of the queries in PDW and Hive for each TPC-H query for each of the four scale factors. For each scale factor, the table contains the speedup of PDW over Hive. The table also contains the arithmetic and geometric mean of the response times at all scale factors. The values of AM-9 and GM-9 correspond to the arithmetic and geometric mean of all the queries but Query 9, since Query 9 did not complete in Hive at the 16TB scale factor due to lack of disk space.

Performance Analysis

Figures 2.1 and 2.2 present an overview of the performance results for the TPC-H queries at the four tested scale factors for both Hive and PDW (the detailed numbers are in shown in Table 2.3). Figure 2.1 shows the normalized arithmetic mean of the response times for the TPC-H queries, and Figure 2.2 shows the normalized geometric mean of the response times; the numbers plotted in the figures are normalized to the response

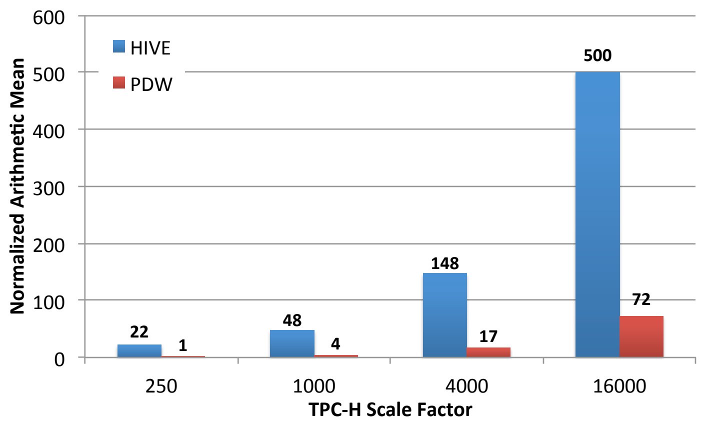


Figure 2.1: TPC-H Performance on HIVE and PDW based on the arithmetic mean at different scale factors (normalized to PDW at Scale Factor = 250).

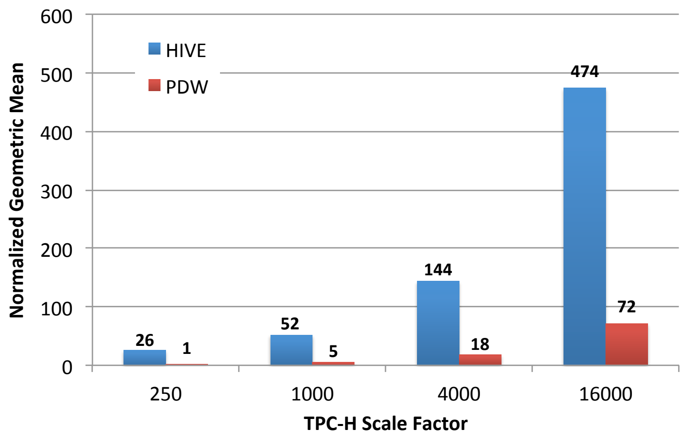


Figure 2.2: TPC-H Performance on HIVE and PDW based on the geometric mean at different scale factors (normalized to PDW at Scale Factor = 250).

times for PDW at scale factor 250. These numbers were computed based on the AM-9 and GM-9 values.

As shown in the figures, PDW has a significantly lower (normalized) arithmetic and geometric mean. Moreover, PDW is always faster than

Query	SF = 250 GB			SF = 1000 GB			SF = 4000 GB			SF = 16000 GB		
	Time (secs)			Time (secs)			Time (secs)			Time (secs)		
	HIVE	PWD	Speedup	HIVE	PWD	Speedup	HIVE	PWD	Speedup	HIVE	PWD	Speedup
Q1	207	54	3.8	443	212	2.1	1376	864	1.6	5357	3607	1.5
Q2	411	7	58.7	530	25	21.2	1081	115	9.4	3191	495	6.4
Q3	508	32	15.9	1125	112	10.0	3789	606	6.3	11644	2572	4.5
Q4	367	8	45.9	855	54	15.8	2120	187	11.3	6508	629	10.3
Q5	536	33	16.2	1686	80	21.1	5481	253	21.7	19812	1060	18.7
Q6	79	5	15.7	166	41	4.0	537	142	3.8	2131	526	4.1
Q7	1007	19	53.0	2447	80	30.6	7694	240	32.1	24887	955	26.1
Q8	967	9	107.4	2003	89	22.5	6150	238	25.8	18112	814	22.3
Q9	2033	207	9.8	7243	844	8.6	27522	3962	6.9	–	15494	–
Q10	489	14	35.0	1107	67	16.5	2958	265	11.2	13195	981	13.5
Q11	242	3	80.8	258	18	14.3	695	99	7.0	1964	302	6.5
Q12	253	5	50.6	490	44	11.1	1597	192	8.3	5123	631	8.1
Q13	392	51	7.7	629	190	3.3	1428	772	1.8	4577	3061	1.5
Q14	154	7	22.0	353	64	5.5	769	164	4.7	2556	640	4.0
Q15	444	21	21.1	585	99	5.9	1145	377	3.0	2768	1397	2.0
Q16	460	36	12.8	654	71	9.2	1732	223	7.8	5695	549	10.4
Q17	654	93	7.0	1717	406	4.2	6334	1679	3.8	25662	6757	3.8
Q18	786	20	39.3	2249	103	21.8	8264	482	17.1	25964	2880	9.0
Q19	376	16	23.5	1069	73	14.6	4005	272	14.7	17644	958	18.4
Q20	606	20	30.3	1296	101	12.8	2461	425	5.8	11041	1611	6.9
Q21	1431	31	46.1	3217	138	23.3	13071	927	14.1	40748	4736	8.6
Q22	908	19	47.8	1145	71	16.1	1744	255	6.8	3402	1270	2.7
AM	605	32	34.1	1421	136	13.4	4634	579	10.2	–	2360	–
GM	474	19	25.2	971	89	10.9	2727	352	7.7	–	1368	–
AM-9	537	24	35.3	1144	102	13.6	3544	418	10.4	11999	1735	9.0
GM-9	442	17	26.3	882	80	11.0	2443	314	7.8	8062	1219	6.6

Table 2.3: Performance of Hive and PDW on TPC-H at four scale factors.

Hive for all TPC-H queries and at all scale factors (see Table 2.3). The average speedup of PDW over Hive is greater for small datasets (34.1X for the 250 GB scale factor). This behavior can be attributed to two factors: a) PDW can better exploit the property that, for small scale factors, most of the data fits in memory, and b) As we will discuss below, Hive has high overheads for small datasets.

In this section, we analyze two TPC-H queries in which PDW significantly outperformed Hive at all scale factors, to gather some insights into the performance differences.

Query 5

As shown in Table 2.3, Query 5 is approximately 19 times faster on PDW than Hive on the 16TB scale. This query joins six tables *customer*, *orders*, *lineitem*, *supplier*, *nation* and *region*) and then performs an aggregation. The plans produced by the PDW and the Hive query optimizers are as follows:

PDW: PDW first shuffles the orders table on *o_custkey*. The shuffle is completed after approximately 258 seconds. Then, PDW performs a join between the *customer*, *orders*, *nation* and *region* tables. This join can be performed locally on each PDW node since the *nation* and *region* tables are replicated across all the nodes of the cluster and the customer table is hash partitioned on the *c_custkey* attribute. The output of the join is shuffled on the *o_orderkey* attribute. The join and shuffle phases run for approximately 86 seconds. The table produced by the previous operations is locally joined with the *lineitem* table, which is partitioned on the *l_orderkey* attribute and then shuffled on the *l_suppkey* attribute. This shuffle and join phase runs for 665 seconds. Then, the resulting table is joined with the *supplier* table (locally). During this join, a partial aggregation on the *n_name* attribute is performed. Finally, all the local tables produced at each PDW node are globally aggregated on the *n_name* attribute to produce the final result. The partial aggregation and the global aggregation operations complete after 40 seconds.

Hive: Hive first performs a map-side join between the *nation* and the *region* tables. A hash table is created on the resulting table and then a map-side join is executed with the *supplier* table. Then, a common join is executed between the table produced and the *lineitem* table. The common join is a MapReduce job that scans the two tables in the map phase, repartitions them over the shuffle phase on the join attribute, and finally performs the join in the reduce phase. This join runs for about 14880 seconds at the 16TB scale dataset. The map and shuffle phases run for approximately 12480 seconds. The output of this operation (TMP table) is then joined with the orders table using the common join mechanism. The running time of this join is 4140 seconds. Then, Hive performs a common join between the *customer* table and the output of the previous operation. During this operation the results are partially aggregated on the *n_name* attribute. This join runs for about 720 seconds. Finally, Hive launches two map-reduce jobs to perform the global aggregation as well as the order-by part of the query.

The reasons why Hive is slower than PDW when running Query 5 are described below.

First, the RCFile format is not a very efficient storage layout. We noticed that the read bandwidth when reading data from the RCFile is very low. For example, during the join between the *lineitem* table and the second temporary table that is created, the read bandwidth during the map phase is approximately 70 MB/sec and the map tasks were CPU-bound (the 8 disks used to hold the database can deliver, in aggregate, almost 800 MB/sec of I/O when accessed sequentially. Tests using the *testdfsio* benchmark showed that in our setup, HDFS delivers approximately 400 MB/sec of read sequential bandwidth).

Another important reason for PDW's improved performance over Hive is that PDW repartitions the intermediate tables so that the subsequent join operations in the query plan can be executed locally. This repartitioning step is generated because the PDW optimizer computes a query plan, and splits the query into sub-queries using cost-based methods that minimize network transfers. As a result, large base tables, like *lineitem*, are not shuffled. Hive on the other hand, does not use any cost-based model to optimize query execution. The order of the joins is determined by the way the user (in this case the Hive developers) wrote the query. This approach results in missing opportunities to optimize joins. For example, since the join order is determined by the way the query is written, the table produced by joining the *nation*, *region* and *supplier* table has to be joined with the *lineitem* table. The *lineitem* table is not “bucketed” on an attribute related to the *supplier* table. As a result, the join is executed using the expensive common join mechanism that repartitions both tables and joins them in the reduce phase. The running time of this task is higher than the total running time of the PDW query at the 16TB scale factor. Another, example is the join between the TMP table and the *orders* table. Notice that the TMP table is produced by a join operation on the *lineitem* table, which is bucketed on the *l_orderkey* attribute. However, the TMP table is not bucketed at all. As a result the join between the TMP table and the *orders* table cannot proceed as a bucketed map-side join and the common join mechanism is used.

Query 19

Query 19 joins two tables (*lineitem*, *part*) and performs an aggregation on the output. This query contains a complex AND/OR selection predicate that involves both tables. This query was approximately 18 times faster in PDW at the 16TB scale factor. The plans produced by PDW and Hive are as follows:

PDW: PDW first replicates the part table at all the nodes of the cluster. This process is completed after 51 seconds. Then, it joins the *lineitem* table with the *part* table at each node, applies the selection predicate and performs a local aggregation operation; these three operations runs for approximately 906 seconds. Finally, it performs a global aggregation of all the results produced by the previous stage.

Hive: Hive performs a common join between the *lineitem* and the *part* table. At the 16TB scale, this join operation runs for about 17540 seconds. The map and shuffle phases run for 14220 seconds. During the reduce phase, a partial aggregation is also performed. Then, Hive launches one

SF = 250 GB	SF = 1 TB	SF = 4 TB	SF = 16 TB
148 secs	339 secs	1258 secs	5220 secs

Table 2.4: Total time for the map phase for Query 1.

more MapReduce job that performs the global aggregation. This job runs for about 25 seconds at the 16 TB scale factor.

As with query 5, PDW tries to avoid network transfers by. For this reason, it replicates the small table (*part*), and then performs the join with the *lineitem* table locally. Hive, on the other hand, redistributes both the *part* and the *lineitem* tables and then performs the join in the reduce phase of the MapReduce job. Hive could have performed a map-side join instead of a common join, but it doesn't make that choice, probably because a hash table on the *part* table wouldn't fit in the memory assigned to each map task.

Similar arguments hold for other queries where PDW significantly outperforms Hive (e.g. Q7, Q8).

Scalability Analysis

As shown in Table 2.3, Hive scales well as the dataset size increases. In this section we analyze some queries where Hive scales sub-linearly when the dataset size increases by a factor of 4.

Query 1

Query 1 scans the lineitem table and performs an aggregation followed by an order-by clause. The bulk of the time in this query is spent in the map phase of the MapReduce job that scans the *lineitem* table. The map tasks scan parts of the *lineitem* table and perform a map-side aggregation. Table 2.4 shows the total time spent in the map phase at each scale factor.

As shown in Table 2.4, when the dataset's size increases from 250 GB to 1TB, the map phase time increases by 2.3X. When the dataset increases from 1TB to 4TB the map phase time grows by a factor of 3.7. This factor becomes 4 when the dataset increases from 4TB to 16TB. The reason for this behavior is as follows:

The *lineitem* table contains 512 buckets based on the *l_orderkey* attribute, and it consists of 512 HDFS files (one file per bucket). We noticed that only 128 files out of the 512 contain data. The remaining 384 files are empty. According to the TPC-H specification, the *l_orderkey* attribute is sparsely populated (only the first 8 of every 32 keys are used). Hive uses hash partitioning to determine the bucket number that corresponds to each row. A hash function that assumes uniform distributions could have created this uneven distribution of data in buckets.

For the 250 GB dataset, 512 map tasks are launched (one per file). The map tasks that process non-empty files finish in approximately 75 seconds. The map tasks that process the empty files finish in 6 seconds. The total number of map tasks that can be run simultaneously on the cluster is 128 (128 map slots). Ideally, the first 128 map tasks would process non-empty files and complete in 75 seconds, and then 3 rounds would be needed to process the remaining empty files for a total time of 93 seconds. However, the total time is 148 seconds. This behavior happens because in the first round of map task allocation, both empty and non-empty files are processed. As a result, there is at least one map slot that processes two non-empty files, which then increases the total running time of the map phase.

For the larger dataset sizes, more than one map tasks process the non-empty files, and as a result, the ratio of the map tasks that do "useful" work over those that process empty files increases. For example, at the 1TB scale factor, 768 map tasks are launched (384 for the empty files and 384 for the non-empty files). As the datasets get bigger, the overhead introduced

	SF = 250 GB	**SF = 1 TB**	**SF = 4 TB**	**SF = 16 TB**
Sub-query 1	85 secs	104 secs	169 secs	263 secs
Sub-query 2	38 secs	51 secs	51 secs	63 secs
Sub-query 3	109 secs	236 secs	658 secs	2234 secs
Sub-query 4	654 secs	735 secs	797 secs	813 secs

Table 2.5: Time breakdown for Query 22.

by the empty files reduces.

Query 22

Query 22 consists of four sub-queries in Hive. The average time spent in each sub-query for all scale factors is shown in Table 2.5. Sub-query 1 scans the customer table, applies a selection predicate, and finally stores the output in a temporary table. The query consists of two MapReduce jobs at the first three scale factors and of one MapReduce job at the 16TB scale factor.

The first job executes the query (in a map-only phase), and outputs the result to a set of files (one per map task). The second job is a filesystem-related job that runs for 50 seconds at all the first three scale factors. This job stores the result of the previous query across fewer files. In the first MapReduce job, 200 map tasks are launched at the first three scale factors (one per customer bucket) and 600 map tasks when SF = 16TB. This is because at the 16TB each customer bucket consists of 3 HDFS blocks. The job time is 34 seconds when SF = 250 GB, 47 seconds when SF = 1TB, 102 seconds for the SF = 4TB and 263 sec when SF = 16TB. The job's running time does not increase by a factor of 4 as the dataset size increases by 4X. If we take a more careful look at the map phase of the job, we notice that each map task processes approximately 9.4 MB, 37 MB, 148 MB and 256 MB of compressed data at each scale factor. Each map task runs for about

9 seconds when SF = 250 GB, and 12 seconds when SF = 1000 GB. Since each map task processes a small amount of data (in the order of a few MB), the map task time does not scale linearly with the dataset size as the overhead associated with starting a new map task dominates the map task's running time.

Sub-query 2 consists of one MapReduce job that scans the output of the previous query, performs an aggregation, and stores the result into another table. When SF = 250 GB, two map tasks are launched and they finish after 12 seconds. When SF = 1TB, three map tasks are launched to process a total of 735 MB of data, and each task finishes after 27 seconds. When SF = 4TB, twelve map tasks are launched to process a total of 3 GB of data, and each one finishes in 27 seconds. Finally, at the 16TB scale factor 600 map tasks are launched to process a total of approximately 12 GB (each map task processes up to 102 MB and runs for at most 15 seconds). Observe that the running time of this query is the same at SF = 1000 and SF = 4000. The reasons for this behavior are:

- The map task time is the same at both scale factors since each map task processes one HDFS block (256 MB).
- Since the available number of map slots is 128, the map tasks launched at these two scale factors (3 and 12 map tasks respectively) can be executed in one round.

Sub-query 3 scans the orders table, performs an aggregation and stores the output in a temporary table. The orders table consists of 512 buckets on the orderkey attribute. Similar to the lineitem table, only 128 files actually contain data. The scaling behavior of this query is similar to that of Query 1 presented above.

Sub-query 4 performs two joins. The first one is executed between the outputs of Sub-query 1 and Sub-query 3. Hive attempts to perform a map-side join at all scale factors. However, the join always fails after about

400 seconds due to Java heap errors (this time varies slightly across all scale factors). Then, a backup task is launched that executes the join using Hive's common join mechanism. The join completes in 110 seconds, 150 seconds, 196 seconds and 300 seconds at the 4 scale factors. The reason for this scaling behavior is the small amount of data processed per map task (similarly to Sub-query 1). Similar observations hold for the remaining MapReduce jobs of this query (the second join, the group-by part, and the order-by part).

Discussion

Based on our analysis above, we now summarize the reasons that result in PDW outperforming Hive. These reasons are:

1. Although the RCFile format is an efficient storage format, it has a high CPU overhead.
2. Cost-based optimization in PDW results in better join ordering. In Hive, hand-written sub-queries and absence of cost-based optimization results in missed opportunities for better join processing.
3. Partitioning attributes in PDW are crucial inputs to the optimizer as it tries to produce plans with joins that can be done locally, and hence have low network transfer costs. In Hive although tables are divided into buckets, this information is not fully exploited by the
4. PDW replicates small tables to force local joins. Hive supports the notion of map-side join, which is similar to the replication mechanism of PDW. In a map-side join, a hash table is built on the smaller table at the Hive master node. The hash table is distributed to all the nodes using Hadoop's distributed cache mechanism. Then, all the map tasks load the hash table in-memory, scan the large table and perform a map-side only join. One disadvantage of this approach

is that the hash table must fit in the memory assigned to each map task. As a result, there is a tradeoff between the number of map slots per node (which is determined by the user) and the memory available to each map task. This memory restriction is frequently the reason why this query plan fails. Another issue is that each new map task at a node has to load the hash table in memory from the local storage. (The hash table does not persist across map tasks on the same node). The authors in [66] present another alternative to map-side joins that avoids these issues. A related point is that although bucketing can improve performance by allowing bucketed map-side joins, the bucket should be small enough so that it can fit in the memory available to each map task. Having many small buckets at each table can help getting more bucketed map-side joins. However, when scanning tables that consist of many small buckets, the map task time can be dominated by the startup cost of the map task. Moreover, it's possible that the number of map tasks launched is high so multiple map rounds of short map tasks are needed to complete the scan (e.g. Sub-query 1 in Q22).

5. Finally, although Hive does not exploit bucketing as efficiently as partitioning is exploited by PDW, it is worth noting that unlike PDW, the buckets of two tables that are bucketed on the same attribute, are not guaranteed to be co- located on the filesystem (HDFS). Even if Hive is able to exploit the bucketing information more efficiently, absence of co-location would translate to network I/O, which in turn can significantly deteriorate performance [56].

Regarding scalability, Hive scales better than PDW (i.e. the scaling factors in the six right-most columns of Table 2.3 are lower for Hive) for the following reasons:

1. It has extra overheads for small datasets (e.g. overheads introduced

by empty data files, startup cost of map tasks).

2. For some queries, increasing the dataset size does not affect the query time, since there is enough available parallelism to process the data (e.g. enough available map slots). Sub-query 2 of Q22 is such an example.

3. Some tasks take the same amount of time at all scale factors (filesystem-related job, map-side join fails after the same amount of time).

2.3.4 "Modern" OLTP Workload: MongoDB vs. SQL Server

In this section, we compare the performance of MongoDB and SQL Server in a cluster environment, using the YCSB data-serving benchmark [43].

Workload Description

We used the YCSB benchmark, to evaluate our MongoDB implementation (Mongo-CS), the original MongoDB system (Mongo-AS), and our sharded SQL Server implementation (SQL-CS) on the "modern" OLTP workloads that represent the new class of cloud data-serving systems. The YCSB benchmark consists of five workloads that are summarized in Table 2.6. The YCSB paper [43] contains more details about the request distributions used by each workload. We have extended YCSB in the following two ways: First, we added support for multiple instances on many database servers, so as to measure the performance of client-sharded SQL Server (SQL-CS) and client-sharded MongoDB (Mongo-CS). Second, we added support for stored procedures in the YCSB JDBC driver.

We ran the YCSB benchmark on a database that consists of 640 million records (80M records per node). The dataset size per node is approximately 2.5 larger than the available main memory at each server machine. Each

Workload	Operations
A - Update heavy	Read: 50%, Update: 50%
B - Read heavy	Read: 95%, Update: 5%
C - Read only	Read: 100%
D - Read latest	Read: 95%, Append: 5%
E - Short ranges	Scan: 95%, Append: 5%

Table 2.6: Time breakdown for Query 22.

record in the database is 1024 bytes long and consists of one 24-byte key and 10 extra fields of 100 bytes each. All the fields as well as the key are stored as strings. Each key is generated by an integer, by using the string representation of the integer prefixing it with a sequence of âĽ˘0′, so that the total length of the key is 24 bytes. The data has an index on the record key, both in SQL Server and MongoDB. No other indexes were built in these systems. The record key is also used as the shard key for Mongo-AS.

We ran Mongo-AS and Mongo-CS in “safe” mode. This means that, after each write request, the client waits for a response from the server. This message shows that the server received the request and applied the write. However, there is no guarantee that the data was actually written to disk. We decided to not enable the “fsync” parameter that MongoDB provides, and as a result we do not wait for the writes to be flushed to disk before the response message is sent back to the client (this choice was made to improve the write performance in MongoDB).

While SQL Server supports ACID transaction semantics (at the default READ COMMITTED isolation level), the MongoDB experiments were run without durability support. The version of MongoDB that we used supports durability via write-ahead journaling. The journal is flushed to disk every 100 ms. This 100 ms delay means that the redo log by itself does not fully support durability, unless a commit acknowledgement is provided. For our experiments, we elected to run MongoDB without logging so that it doesn't pay any additional performance penalty.

In our setting, each client node runs 100 client threads for a total of 800 client threads (recall we have 8 machines dedicated to the clients). The five YCSB workloads are run sequentially, and before every run the main memory is flushed. After executing workloads D and E, which contain insertions and alter the record keys, the database is dropped and reloaded. Each read request reads all the record fields, and each update request updates only one field. Each scan request reads at most 1,000 records from the database. Finally, each append request inserts a new record in the database whose key has the next greater value than that of the last inserted key.

Each workload is run for 30 minutes. The values of latency and throughput reported are the average values over the last 10 minutes of execution, measured every 10 second interval. In the figures below, we also report the standard error across these 60 measurements.

Data Preparation

During the load phase, we used 8 client nodes, each running 16 client threads (as there are 16 hyper-threaded cores on each node). These threads are responsible for generating the correct keys and loading the data across the 8 server nodes.

Mongo-AS can automatically split and migrate data chunks across the shards by using a "balancer" process that takes care of load balancing. However, since the range and distribution of keys to be inserted are known in advance, we manually defined the boundaries for all of the initially empty chunks and spread them across the 128 shards of the cluster. Then, we started loading the data. In this way, the high cost of chunk migration across the shards is minimized. This technique is described in the MongoDB documentation [14]. This process resulted in an even distribution of the chunks across all the shards. The loading time with this strategy was 114 minutes.

The loading time for SQL-CS and Mongo-CS was 146 and 45 minutes respectively. The SQL-CS load time is higher than that of the Mongo-CS system because a bulk insert method was not used to load the data. Instead, every insertion was a separate transaction issued at the database.

Experimental Evaluation

The YCSB benchmark focuses on the latency of requests when the data-serving system is under load. However, as the load increases on a given system, the latency of requests typically increases since there is more contention for resources. In practice, the cloud service providers decide on an acceptable latency, and then provision enough servers to achieve the desired throughput. The YCSB benchmark aims to describe the tradeoffs between throughput and latency for each system by measuring latency as throughput is increased, until the point at which the system is saturated and throughput stops increasing. To run the benchmark, the (benchmark) user provides a target throughput as an input parameter, and the system returns the average latency as well as the actual throughput that is achieved. The user stops increasing the target throughput when the actual throughput that is achieved is lower than the target value.

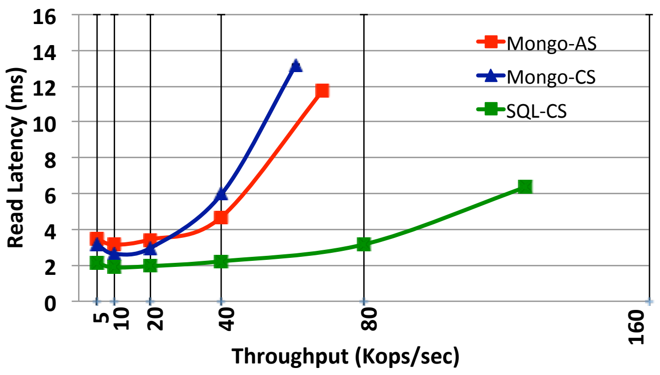
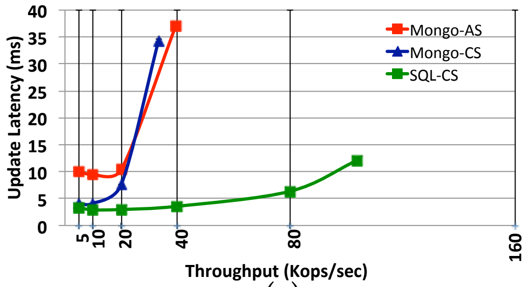


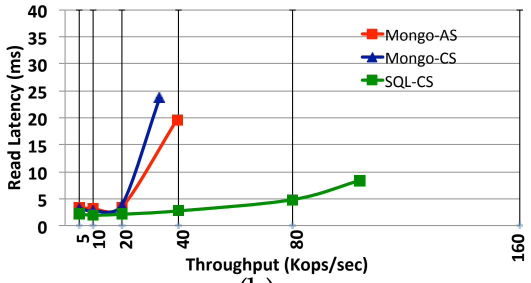
Figure 2.3: Workload C: 100% reads.

Figure 2.3 shows the latency vs. throughput curve for the "Read-Only" workload (Workload C). The label on the x-axis shows the target throughput values provided by the user. Each data point corresponds to a pair of the actual throughput achieved and the average read latency for that throughput.

As shown in Figure 2.3, SQL-CS is able to achieve the highest throughput (125,457 ops/sec) with an average read latency of 6.4 ms. Mongo-AS and Mongo-CS were not able to reach the 80,000 ops/sec of target throughput and peaked at 68,533 and 60,907 ops/sec respectively. The average read latency values at the highest throughput achieved for Mongo-AS and Mongo-CS are 11.8 ms and 13.2 ms respectively. Moreover, SQL-CS has lower latency than the other systems for all the target throughputs. This workload is disk-bound is all the systems at the highest achievable throughput. However, the latency of each read request is higher with the Mongo-AS and the Mongo-CS systems compared to SQL-CS. We noticed, that SQL Server reads 8KB from disk for each request that leads to a buffer pool miss, whereas Mongo-AS and Mongo-CS read on average 32 KB from disk for each read request. Since the I/O activity pattern in this workload is largely random access, Mongo-AS and Mongo-CS waste disk bandwidth by reading in data that is not needed.



(a)



(b)

Figure 2.4: Workload B: 95% reads, 5% updates.

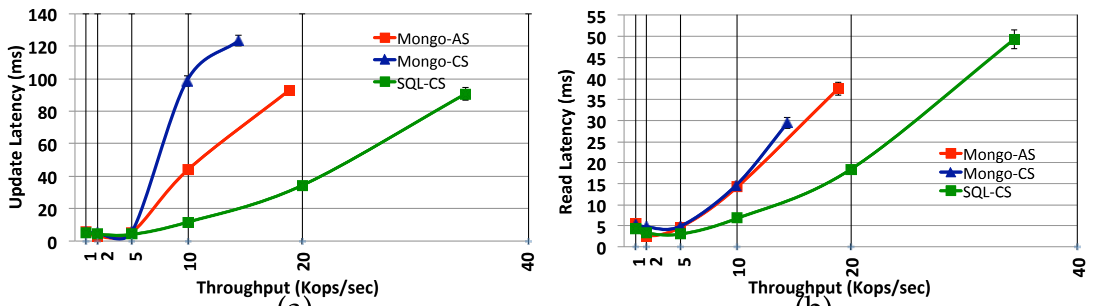
Figure 2.4 presents the latency vs. throughput curves for the "Read-

Heavy" workload (Workload B). The workload consists of 95% reads and 5% updates. The left-hand curve presents latency results for the update operation as the target throughput increases, and the right-hand curve presents latency results for the read operation.

The Mongo-DB systems cannot achieve the 40,000 ops/sec throughput target. Moreover, the update and read latencies increase abruptly (up to 24 ms. for read requests and 37 ms. for update requests) when the throughput increases from 20,000 to 40,000 ops/sec. However, SQL-CS is able to achieve 103,789 ops/sec with an update latency of 12 ms, and a read latency of 8.4 ms. This workload is disk-bound in each of these three systems. We noticed that each system achieves the same number of operations/sec as in Workload C. However, during checkpointing in SQL-CS, or when the MongoDB systems are flushing data to disk, the throughput decreases. For example when checkpointing was not happening, SQL-CS is able to reach on average about 15,000 ops/sec per server node (similar to workload C), but during the checkpointing interval the throughput decreases to 7,000-8,000 ops/sec. This is the reason why the maximum throughput achieved is lower in Workload B than with Workload C.

Figure 2.5 describes the "Update-Heavy" workload (Workload A). This workload is similar to Workload B, but it contains a significantly larger fraction of updates (50% updates compared to 5% updates for Workload B).

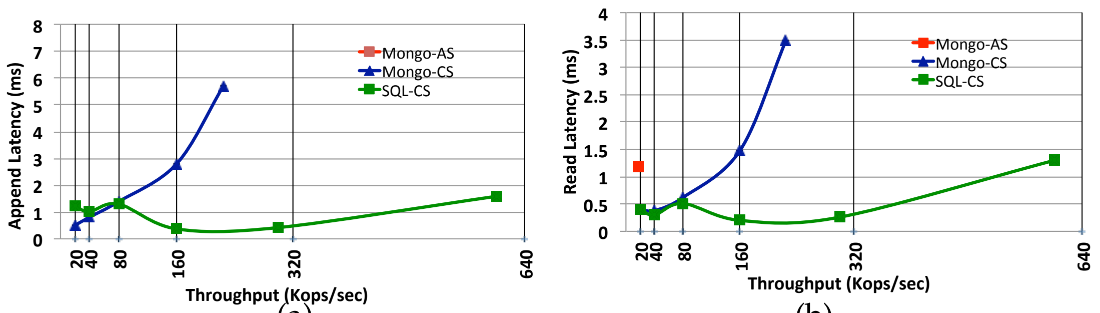
Using the mongostat tool [13], we observed that the percentage of time that was spent at the global lock ranges from 25%-45% at each one of the 128 "mongod" instances. This percentage ranges from 4%-12% when running Workload B, which contains a smaller fraction of updates. Similarly, the increased locking activity in SQL-CS is the reason why both the read and update latencies are higher than those in Workload B. To verify this hypothesis, we reran the same workload but now using the "read uncommitted" isolation level and measured the read and update



(a) (b)

Figure 2.5: Workload A: 50% reads, 50% updates.

latencies. When the target throughput was 40,000 ops/sec, the average update latency was 69 ms. and the average read latency was 15 ms. The read latency is significantly lower now, compared to that of the previous experiment where the "read committed" isolation level was used. This can be attributed to the fact that the read operations are not blocked by the write operations and thus the waiting time is reduced.



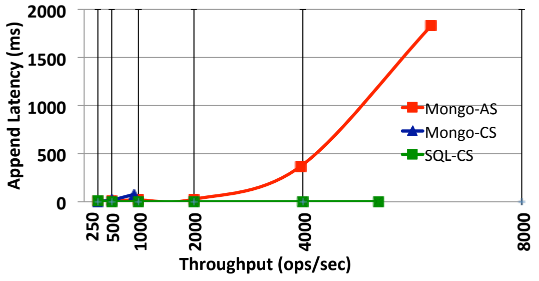
(a) (b)

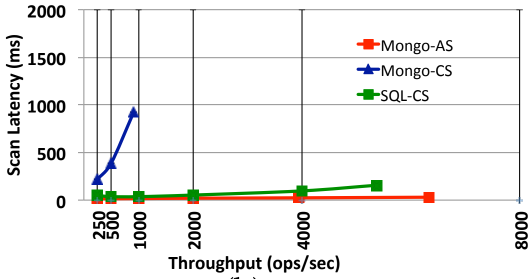
Figure 2.6: Workload D: 95% reads, 5% appends.

Figure 2.6 shows the append latency and the read latency vs. throughput curves for Workload D. The read request distribution for Workload D is "Read Latest". This means that there is a high probability that a read request will read the latest item that was just inserted into the database.

We observed that in SQL-CS, 99.5% of the requests are to pages that are in the buffer pool. This means that the majority of the read requests do not hit the disk. Consequently, the read latencies for SQL-CS are in the order of a few microseconds. During the execution of this workload, SQL-CS is CPU-bound. SQL-CS has higher latencies for the low target throughput values (up to 80,000 ops/sec) compared to the greater throughput values (160,000 ops/sec and 320,000 ops/sec). This behavior happens because at the low throughput values, memory is not fully filled with useful data until after the 30-minute interval. For example, when the target throughput is 20,000 ops/sec, only 19.2 GB of the main memory is filled (of the 32 GB that is available) and as a result many read requests still incur a disk I/O. Mongo-CS has high read and append latencies when it hits the highest achievable throughput (224,271 ops/sec) compared to SQL-CS. Interestingly, this workload is neither CPU-bound nor disk-bound (in MongoDB). Mongo-AS has a very high append latency (320 ms) for this workload when the target throughput is at 20,000 ops/sec, which is why this point does not appear in the graph in Figure 2.6. Moreover, Mongo-AS crashes when running this workload when the target throughput is set to a value greater than 20,000 ops/sec. After running the system with the debugger enabled, we observed that at some point the client machines wait for a response message from the server after an append request, but this message never arrives due to socket exceptions. For this reason, the clients stopped sending new requests to the servers and the throughput went down to 0 ops/sec.

Finally, Figure 2.7 shows the performance of the three systems on the "Short Ranges" workload (Workload E). All three systems are disk-bound when the servers hit their maximum throughput. As shown in the figure, Mongo-AS achieves the highest throughput (6,337 ops/sec) and has the lowest scan latency (30.4 ms). This behavior can be attributed to the fact that Mongo-AS uses range partitioning to distribute the data





(a) (b)

Figure 2.7: Workload E: 95% scans, 5% appends.

chunks across the servers, whereas both SQL-CS and Mongo-CS use hash partitioning. That means that Mongo-AS can determine, based on the range requested, which partitions contain the data and scan only those (typically one partition for each short range query) whereas SQL-CS and Mongo-CS need to scan as many partitions as needed until the appropriate records are found. However, Mongo-AS has a very high append latency (about 1832 ms) compared to SQL-CS (about 2 ms).

2.3.5 Discussion

In this section we compared a SQL system (PDW, SQL-CS) and a representative NoSQL system (Hive and MongoDB) on a DSS and an OLTP workload.

Our evaluation has shown that although NoSQL systems have significantly evolved over the past years, their performance still lags behind that of the relational database systems. On the one hand, the parallel database system (PDW) was approximately 9X faster than the MapReduce-based data warehouse (Hive) when running TPC-H at a 16TB scale, even when indexing was not used in PDW. The robust and mature cost-based optimization and sophisticated query evaluation techniques that are employed by the relational database system allow it to produce and run more efficient plans than the NoSQL system. The MapReduce-based systems could adopt these techniques to improve their performance.

Furthermore, SQL-CS was able to achieve higher throughput than the MongoDB for the same number of clients, and it had lower latency across for almost every single test of the YCSB benchmark. Interestingly, this is the case even when the NoSQL system did not provide any form of durability. This finding comes in contrast with the widely held belief that relational databases might be too heavy weight for this type of workload, where the requests consist of a single simple operation and do not require the complex transactional semantics that RDBMSs can handle.

2.4 Summary

Today there are a number of popular alternatives to using relational data processing systems, for both DSS workloads and the Web 2.0 data-serving workloads. While there are many complex factors that go into the choice of the system that gets deployed for specific data processing tasks (e.g., integration of the data processing system with an overall solutions stack, manageability, open-source vs. closed-source, etc.), one crucial aspect that is often a factor in choosing a data processing system is the performance of the system. In this chapter, we examined this performance aspect of NoSQL and SQL systems using two benchmarks - the TPC-H benchmark and the YCSB benchmark. Our results find that the SQL systems continue to provide a significant performance advantage over their NoSQL counterparts, but the NoSQL alternatives are competitive in some cases. The NoSQL and SQL systems also have different focuses on non-performance related features, such as data models (the NoSQL systems tend to have more flexible data models), support for auto-sharding and automatic load balancing and different consistency models. It is likely that in the future these systems will start to converge on the functionality aspects.

Chapter 3

Column-Oriented Storage Techniques for MapReduce

3.1 Introduction

Over the last few years, there has been tremendous growth in the need for large scale data processing systems. These systems were once the province of parallel database management systems (DBMSs), but lately the MapReduce paradigm has gained substantial momentum. However, there is a growing sense that there are advantages to both paradigms, and that techniques which have been successfully used in one can be used to fix deficiencies in the other. Performance is one area in particular where parallel DBMSs currently enjoy an advantage over MapReduce as pointed by our study, presented in Chapter 2 and by previous studies [79].

Hadoop [3] is the popular open-source implementation of MapReduce. In this chapter, we describe how the column-oriented storage techniques found in many parallel DBMSs can be used to dramatically improve Hadoop's performance. Our work is motivated by observing the needs of real IBM corporate Hadoop users. These users are very familiar with parallel DBMS technology, but they still pick Hadoop for certain applications because of its ease of use, low cost scaling, fault tolerance on commodity hardware, and programming flexibility.

Corporate users tend to be thrilled by how quickly they can get things working in Hadoop. However, as they try to scale up their workloads, they often face a real and sometimes crippling pain point with respect to performance. For example, some users we worked with at a large consumer bank were trying to use Hadoop to process the logs from web applications. They started with raw log files in text format for a single web application. As logs from additional applications were added and the retention period for the logs grew to 90 days, the 20-node Hadoop cluster they started with could no longer generate reports in a reasonable amount of time. Critics of MapReduce would argue that such users would be better off with a parallel DBMS, but given Hadoop's current advantages for certain applications and the growing investment in Hadoop by many users, this is not a realistic option. The goal is to fix Hadoop, not replace it with a parallel DBMS.

We have observed a recurring pattern of performance issues in Hadoop that are related to: (a) the use of complex data types such as arrays, maps, and nested records, which are common in many MapReduce jobs (b) the ability to write arbitrary map and reduce functions in a programming language instead of using a declarative query language, and (c) Hadoop's choice of Java as its default programming language. These issues do not appear in a parallel DBMS; they are unique to the MapReduce paradigm and Hadoop in particular. The column-oriented storage techniques we describe are specifically designed to address these issues.

Besides column-oriented storage techniques, it should be clear that other DBMS techniques can also be leveraged to improve Hadoop's performance, such as efficient join algorithms and indexing [51, 63, 38, 52]. These techniques are beyond the scope of this work but should be complementary to the ones described here.

3.1.1 Our Contributions

We first present the design and implementation of a column-oriented, binary storage format for Hadoop. We describe how such a format interacts with the replication policy of the Hadoop Distributed File System (HDFS) and the necessary mechanisms to co-locate column data. We demonstrate through experiments that Hadoop can leverage this storage format without incurring a large penalty for reconstructing records from the constituent columns. Such a storage format assumes that the application is willing to pay a one-time loading cost to organize the input data in the appropriate column-oriented fashion. As argued in earlier papers [79], this is a reasonable assumption to make for datasets that are expected to be analyzed multiple times.

The use of text storage formats rather than binary storage formats in performance evaluations of MapReduce has been criticized [49] but never quantified. Other evaluations of Hadoop [79, 26] used text formats, so it has not been clear how much Hadoop can actually benefit from binary formats. We show that simply switching to a binary storage format can improve Hadoop's scan performance by 3x.

We identify performance challenges specific to complex data types in Hadoop, and describe a novel skip-list column format that enables lazy record construction. The lazy record construction we describe is inspired by the late materialization techniques used in column-oriented DBMSs [24]. We also examine techniques that allow lazy decompression in Hadoop. We show that compression techniques like LZO [9] may be too CPU intensive for many MapReduce jobs. Experiments on a real dataset show that lightweight dictionary compression schemes, which provide lower compression ratios than LZO but with less CPU overhead during decompression, may be a better alternative for complex data types. We show that these methods can result in speedups of up to 1.5x over an eager record construction strategy.

It is important to emphasize that our column-oriented techniques leverage extensibilty features that are already in Hadoop, so no modifications to the core of Hadoop are required. Moreover, our techniques do not require the use of a declarative query language and are designed to work with hand-coded MapReduce jobs. Applications using popular serialization frameworks like Avro [1], Thrift [20], or Protobufs [17] can benefit from our techniques with almost no modification. In aggregate, our techniques can improve the performance of MapReduce jobs in Hadoop by as much as two orders of magnitude.

3.2 Hadoop Background

We first provide some background on Hadoop along with a description of the extensibility points that were used to implement our column-oriented storage format.

Consider the example MapReduce job shown in Figure 3.1. This is a simplified version of a real Hadoop job that analyzes a collection of crawled documents and finds all the distinct "content-types" reported for URLs that contain the pattern "ibm.com/jp". We initially focus on the main program where the job is configured with an *InputFormat* and *OutputFormat*.

An *InputFormat* is an important abstraction and extensibility point in Hadoop. It is responsible for two main functions: first, to generate *splits* [1] of the data that can each be assigned to a map task; and second, to transform data on disk to the typed key and value pairs that are required by the map function. An *InputFormat* implements the following three methods:

[1] A split is the unit of scheduling and is a non-overlapping partition of the input data that is assigned to a map task.

```
class MyMapper {
  void map (NullWritable key, Record rec) {
    String url = (String) rec.get("url");
    if (url.contains("ibm.com/jp"))
      output.collect(null,
         rec.get("metadata").get("content-type"));
  }
}

class MyReducer {
  void reduce(NullWritable key, Iterator<Text> vals){
    HashSet<Text> distinctVals = new HashSet<Text>();
    for (Text t: vals)
      distinctVals.add(t);
    for (Text t: distinctVals)
     output.collect(null, key);
  }
}

main() {
  Job job = new Job();
  job.setMapperClass(MyMapper);
  job.setReducerClass(MyReducer);
  job.setInputFormat(SequenceFileInputFormat.class);
  SequenceFileInputFormat.addInputPath(job,"/data/jan");
  job.setOutputFormat(TextOutputFormat.class);
  TextOutputFormat.setOutputPath("/output/job1");
  JobRunner.submit(job);
}
```

Figure 3.1: Example MapReduce job.

addInputPath() is used to specify input data sets when configuring the job.

getSplits() is used by the Hadoop scheduler to get a list of splits for the job.

getRecordReader() is invoked by Hadoop to obtain an implementation of a *RecordReader*, which is used to read the key and value pairs from a given split.

Hadoop provides different *InputFormats* to consume data from text files, comma separated files, etc. For example, the MapReduce job in Figure 3.1 is configured to use a *SequenceFileInputFormat*. A *SequenceFile* stores key and value pairs in a standard, serialized binary format. The dual of an *InputFormat* in Hadoop is an *OutputFormat*, which is responsible for transforming the key-value pairs output by a MapReduce job to a disk format.

The key and value pairs consumed by map and reduce functions can be of any object type. In this thesis, we assume that *Record* objects are used for values. Serialization frameworks like Avro [1], Protocol Buffers [17], or Thrift [20] can be used to provide a record abstraction with methods to convert records to raw bytes, read them from disk, or pass them between map and reduce tasks. The attributes of a record are accessed using a Java *get(name)* method, which takes the name of an attribute as a parameter. Type casting is usually required to access an attribute.

3.2.1 Record Abstraction

In a MapReduce job, the type of keys and values supplied to the map function depends on the *InputFormat*. The programmer is responsible for supplying a map function that is compatible with the *InputFormat*. For instance, in the case of the job described in Figure 3.1, the programmer needs to know that the *SequenceFile* being read contains keys of type *NullWritable* and values of type *Record*.

In this thesis, we assume that MapReduce jobs are written using a generic class that provides a record abstraction. We use the *Record* interface supplied by the Avro serialization framework. *LazyRecord* and *EagerRecord* described in Section 3.5 are classes that implement this interface. *ColumnInputFormat* produces keys of type *NullWritable* and values of type *Record*.

Attributes are accessed using a *get(name)* method that takes the name of the attribute as a parameter. The return type for this method is *java.lang.Object*. As a result, type casting is required to access the field values. This *Record* class can be used for records that conform to any schema. Figure 3.1 illustrates the use of this record abstraction in a map function.

Other serialization frameworks have emerged in the open source that generate a binary representation for a record. Examples include Protocol Buffers [17] and Thrift [20]. These frameworks typically allow developers to specify a schema so that records can be serialized and deserialized efficiently. The schema language supports complex fields like arrays, maps, and nested records. While we used Avro, the principles we describe in this chapter are also applicable to Thrift and Protobufs.

Avro also supports the notion of a "specific" record. Given a schema, the Avro compiler can be used to produce a Java class containing specific *get* methods for each of the attributes with precise return types. For instance, one could generate a URLInfo class using the schema from Figure 3.2. The equivalent map function for Figure 3.1 would be simplified to:

```
map(NullWritable key, URLInfoRecord rec)
{
 if (rec.getUrl().contains.("ibm.com/jp"))
  output.collect(null,
    rec.get("metadata").get("content-type"))
}
```

Note that as of Version 1.3.3, by default, Avro supplies only a single *get()* method in its generated classes, much like the generic *Record* class. Extending the compiler to generate accessor methods with the appropriate return types is not difficult. Using generated classes provides a small per-

```
URLInfo {
  Utf8 url,
  Utf8 srcUrl,
  time fetchTime,
  Utf8[] inlink,
  Map<String, String> metadata,
  Map<String annotations,
  byte[] content
}
```

Figure 3.2: Example schema with complex types.

formance advantage for serialization and deserialization when compared to the *Record* class. A careful comparison with generated classes instead of the *Record* class is left as future work.

3.3 Challenges

This section discusses the performance challenges outlined in the introduction in more detail.

3.3.1 Complex Data Types

In large scale data analysis, it is often convenient to use complex types like arrays, maps, and nested records to model data. Recent studies [70, 42] have argued that it is better to use native, nested representations of complex types in read-mostly analysis. This is in contrast to flattening complex types into normalized relational tables.

Figure 3.2 shows an example schema with complex types. The example is taken from an actual intranet search application where documents are crawled and stored along with metadata, extracted annotations, and inlinks. The annotations and the metadata vary widely from page to page and are therefore stored using maps. Inlinks are stored in an array. The use of complex types causes two major problems: deserialization costs and the lack of effective column-oriented compression techniques.

3.3.2 Serialization and Deserialization

Serialization is the process of converting a data structure in memory into bytes that can be transmitted over the network or written out to disk. Deserialization is the inverse of this process.

The overhead of deserializing and creating the objects corresponding to a complex type can be substantial. Previous studies [42, 63] have noted the importance of paying attention to the cost of deserialization and object creation in Hadoop. Most column-oriented DBMSs are implemented in C++, allowing column data from disk to be directly accessed in memory as an array without any deserialization overhead [22]. For example, suppose we want to compute the sum of 1 million integers in a file. In C++, the integers can be read into a memory buffer, and an array pointer can be cast to the beginning of the buffer. Then the array's elements can be summed directly in a tight loop. Java, on the other hand, would require deserializing each integer from the memory buffer before summing it.

We conducted an experiment to measure this overhead. Our experiments showed that the CPU overhead of deserialization and object creation is so significant that it can quickly become a bottleneck in Hadoop. This overhead even affects simpler data types such as integers. For more details on this experiment, see Appendix 3.7.

3.3.3 Compression

Using compression results in lower I/O costs at the expense of higher CPU costs. In general, column-oriented storage formats tend to exhibit better compression ratios since data within a column tends to be more similar than data across columns. Previous studies have looked at compression in the context of an column-oriented DBMS [23]. However complex types, which are not as amenable to techniques like run-length compression, dictionary compression, offset encoding, etc., were not considered in those studies.

Lightweight compression schemes are critical in Hadoop. Anecdotal evidence points to poor results when using “heavy” compression schemes like ZLIB, which can achieve excellent compression ratios but incur substantial CPU overhead during decompression. LZO [9] is commonly used in Hadoop to provide reasonable compression ratios with low decompression overhead. In Section 3.5.3, we describe a lightweight dictionary compression scheme for our column-oriented storage format that works well with complex types and provides better performance than LZO.

3.3.4 Query Language vs. Programming API

In contrast to a DBMS, where a declarative query language is compiled into a set of runtime operators, the basic MapReduce framework provides only a programming API. Unfortunately, many of the advanced techniques used by column-oriented DBMSs are not feasible with hand-coded map and reduce functions – the programming task would be too difficult for a human. These techniques include operating on compressed data [23], the use of SIMD instructions [35], and late materialization [24]. Eventually some of these techniques may appear in declarative languages for Hadoop, such as Pig [16], Hive [5], or Jaql [8]. However, this work only focuses on hand-coded MapReduce jobs written against the programming API of Hadoop.

3.4 Column-Oriented Storage

We now describe the design and implementation of our column-oriented storage format and its interaction with Hadoop’s data replication and

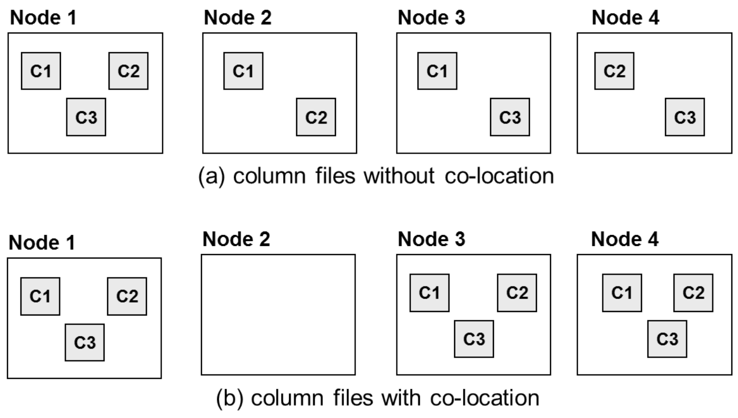

Node 1
Node 2
Node 3
Node 4
C1
C2
C3
(a) column files without co-location
(b) column files with co-location

Figure 3.3: Co-locating column files.

scheduling.

3.4.1 Replication and Co-location

A straightforward way to implement a column-oriented storage format in Hadoop is to store each column of the dataset in a separate file. This imposes two problems. First, how can we generate roughly equal sized splits so that a job can be effectively parallelized over the cluster? Second, how do we make sure that the corresponding values from different columns in the dataset are co-located on the same node running the map task?

The first problem can be solved by horizontally partitioning the dataset and storing each partition in a separate subdirectory. Each such subdirectory now serves as a split. The second problem is harder to solve. HDFS uses 3-way block-level replication to provide fault tolerances on commodity servers, but the default block placement policy does not provide any co-location guarantees.

Consider a dataset with three columns C1, C2, and C3. Assume the columns are stored in three different files, and for simplicity, also assume that each file occupies a single HDFS block. In practice, a large file could span many HDFS blocks The files of C1-C3 need to be accessed together as a split, but with Hadoop's default placement policy, they could be randomly spread over the cluster. Figure 3.3a illustrates what can happen with Hadoop's default placement policy. C1-C3 are co-located on Node 1 but not co-located on any other node. Suppose a map task is scheduled for the split consisting of C1-C3 but Node 1 is busy. In that case, Hadoop would schedule the map task on some other node, say Node 2, but performance would suffer, since C3 would have to be remotely accessed.

Recent work on the RCFile format [60] avoids these problems by resorting to a PAX [27] format instead of a true column-oriented format. RCFile takes the approach of packing each HDFS block with chunks called *row-groups*. Each row-group contains a special sync marker at the start, followed by a metadata region, and then a data region, with the data region laid out in a column-oriented fashion. The metadata describes the columns in the data region and their starting offsets, as well as the number of rows in the data region. Since all the columns are packed into a single row-group, and each row-group can function independently as a split, it avoids the two challenges that arise when storing columns separately.

RCFile has multiple drawbacks. Since the columns are all interleaved in a single HDFS block, efficient I/O elimination becomes difficult because of prefetching by HDFS and the local filesystem. Tuning the row-group size and the I/O transfer size correctly also becomes critical. With larger I/O transfer sizes like 1MB, records that contain more than 4 columns show very poor I/O elimination characteristics with the default RCFile settings. Finally, extra metadata needs to be written for each row group, leading to additional space overhead.

The next section describes an alternative to RCFile that uses separate files for each column and still avoids these problems. Experiments in Section 5.3 will show that this new format can significantly outperform RCFile.

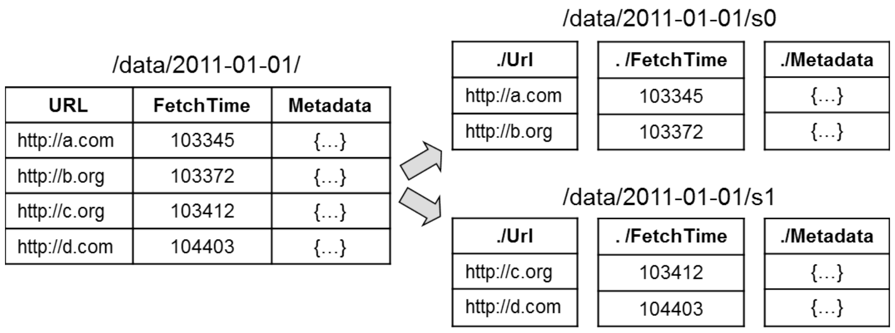


Figure 3.4: Data layout with COF.

3.4.2 The CIF Storage Format

We solved the problem of co-locating associated column files by implementing a new HDFS block placement policy. HDFS allows its placement policy to be changing by setting the property "dfs.block.replicator.classname" to point to the new class in the appropriate configuration file. This feature has been present since Hadoop 0.21.0 and does not require recompiling Hadoop or HDFS.

ColumnPlacementPolicy (CPP) is the class name of our column-oriented block placement policy. For simplicity, we will assume that each column file occupies a single HDFS block and describe CPP as though it works at the file level. In effect, CPP guarantees that the files corresponding to the different columns of a split are always co-located across replicas. Figure 3.3b shows how C1-C3 would be co-located across replicas using CPP. Subdirectories that store splits need to follow a specific naming convention for CPP to work. Files that do not follow this naming convention, are replicated using the default placement policy of HDFS.

We implemented the logic for our column-oriented storage format in two classes: the *ColumnInputFormat* (CIF) and the *ColumnOutputFormat* (COF). Data may arrive into Hadoop in any format. Once it is in HDFS, a parallel loader is used to load the data using COF.

Consider a example scenario involving the data described in Figure 3.2.

Assume that crawled data arrives at regular intervals and that a day's worth of data has arrived and needs to be stored in "/data/2011-01-01". When a dataset is loaded into a subdirectory using COF, it breaks the dataset into smaller horizontal partitions. Each partition, referred to as a *split-directory*, is a subdirectory with a set of files, one per column in the dataset. An additional file describing the schema is also kept in each split-directory. Figure 3.4 shows the layout of data using COF, with split-directories s0 and s1.

When reading a dataset, CIF can actually assign one or more split-directories to a single split. The column files of a split-directory are scanned sequentially and the records are reassembled using values from corresponding positions in the files. Projections can be pushed into CIF by supplying it with a list of columns. This can be done while configuring a MapReduce job as follows:

```
ColumnInputFormat.setColumns(job, "url, metadata");
```

The record objects created by CIF are populated only with the fields that are selected. The files corresponding to the remaining columns are not scanned.

3.4.3 Discussion

A major advantage of CIF over RCFile is that adding a column to a dataset is not an expensive operation. This can be done by simply placing an additional file for the new column in each of the split-directories. With RCFile, adding a new column is a very expensive operation – the entire dataset has to be read and each block re-written.

Adding columns is well known to be an important feature. This is a particularly common operation when the dataset needs to be augmented with derived columns computed from the existing columns. We have also seen the need for this feature when a customer starts by extracting a set of columns from raw input files (such as logs) into organized storage for efficient querying. As business needs evolve, additional columns from the raw input files need to be moved to organized storage.

Experiments in Section 5.3 will show that CIF does not pay a performance penalty for this flexibility advantage over RCFile. In fact, CIF overcomes some drawbacks of RCFile with respect to metadata overheads, poor prefetching, and I/O elimination.

On the other hand, a potential disadvantage of CIF is that the available parallelism may be limited for smaller datasets. Maximum parallelism is achieved for a MapReduce job when the number of splits is at least equal to the number of map slots, say m. RCFile allows fine grained splits at the row-group level (4MB) when compared to split-directories in CIF (typically 64 MB). For RCFile, assuming that each HDFS block has r row-groups, maximum parallelism is available when the total dataset size is greater than m/r blocks. With CIF, this happens when there are at least m split-directories. If we choose split-directories containing c blocks worth of data in each directory (where c is the number of columns), full parallelism is available when the dataset size exceeds $m \times c$ blocks.

Assuming a typical cluster with 200 map slots and 64M blocks, a dataset with 10 columns would need to be at least 128GB in size before full parallelism is reached. Since we expect to deal with datasets in the terabyte range on Hadoop, we expect to be able to utilize all the available parallelism in a cluster with CIF. In practice even with RCFile, large row-groups are preferred since they minimize metadata overhead and improve I/O elimination (see Figure 3.9 in Appendix 3.7).

In summary, CIF offers flexibility and some performance benefits over RCFile. This advantage comes at the cost of needing to install a special block-placement policy for HDFS and potentially limiting the amount of parallelism for smaller datasets. We do not expect either of these considerations to be a problem for large deployments. A deeper analysis of other

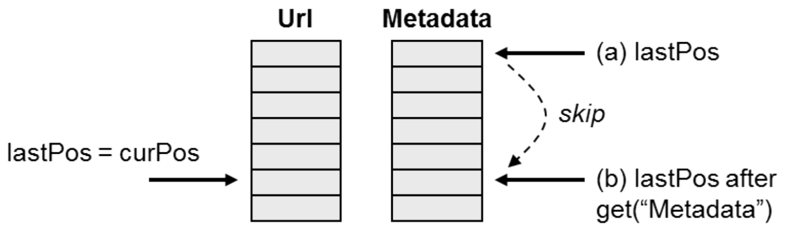


Figure 3.5: Lazy record construction.

considerations such as load-balancing and re-replication after failures are important avenues for future work.

3.5 Lazy Record Construction

In this section, we describe our lazy record construction technique, which is used to mitigate the deserialization overhead in Hadoop, as well as eliminate disk I/O. The basic idea behind lazy record construction is to deserialize only those columns of a record that are actually accessed in a map function. Consider the example MapReduce job in Figure 3.1 that was described earlier. The metadata column is accessed in the map function only for records where the URL column contains the pattern "ibm.com/jp". Using lazy record construction, we can avoid deserializing the metadata column for the records where the URL column does not contain this pattern.

3.5.1 Implementation

CIF can be configured to use one of two classes for materializing records, namely, *EagerRecord* or *LazyRecord*. Both of these classes implement the same *Record* interface. As a result, the map function code looks the same, regardless of which class is instantiated.

EagerRecord eagerly deserializes all the columns that are being scanned by CIF. *LazyRecord* is slightly more complicated. Internally, *LazyRecord* maintains a split-level *curPos* pointer, which keeps track of the current record the map function is working on in a split. It also maintains a *lastPos* pointer per column file, which keeps track of the last record that was actually read and deserialized for a particular column file. Both pointers are initialized to the first record of a split at the start of processing.

Each time *RecordReader* is asked to read the next record, it increments *curPos*. No bytes are actually read or deserialized until one of the *get()* methods is called on the resulting *Record* object. Consider the example in Figure 3.5. Since get("url") is called on every record, *lastPos* is always equal to *curPos* for the URL column. However, for the metadata column, *lastPos* may lag behind *curPos* if there are records where the URL column does not contain the pattern "ibm.com/jp". When the URL column contains this pattern and get("metadata") is called, *lastPos* skips ahead to *curPos* before the metadata column is deserialized.

Note that complex column types with variable lengths are the main reason both the split-level *curPos* and per column file *lastPos* pointers are needed for lazy record construction. Ostensibly, it might seem like just a *curPos* pointer per column file could be used without a *lastPos* pointer. However, in that case, each next record call would require all the columns to be deserialized for length information to update their corresponding *curPos* pointers. This in turn would defeat the purpose of lazy record construction.

3.5.2 Skip List Format

A skip list format [80] can be used within each column file to efficiently skip records. The skip list format used in CIF is shown in Figure 3.6. A column file contains two kinds of values, regular serialized values and *skip blocks*. Skip blocks contain information about byte offsets to enable

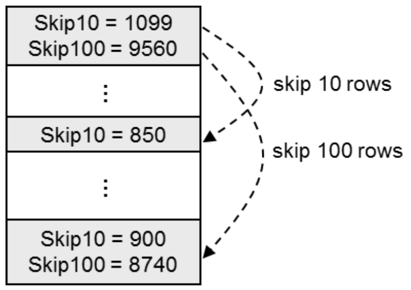


Figure 3.6: Skip list format for complex types.

skipping the next N records, where N is typically configured for 10, 100, and 1000 record skips.

Column files support a *skip()* method that is called by *LazyRecord* as skip(*curPos* - *lastPos*). If a column file is not formatted as as a skip list, each record is skipped individually, resulting in no deserialization or I/O savings. The cost for creating a skip list format is paid once at load time. Experiments in Section 3.7 show that the additional overhead incurred during loading is minimal.

3.5.3 Compression

We propose two schemes to compress columns of complex data types: compressed blocks, and dictionary compressed skip lists. Section 5.3 compares these two schemes. Both schemes are amenable to lazy decompression where portions of the data that are not accessed in the map function are not decompressed.

Compressed Blocks: This scheme uses a standard compression algorithm to compress a block of contiguous column values. Multiple compressed blocks may fit into a single HDFS block. The compressed block size is set at load time. It affects both the compression ratio and the decompression overhead. A header indicates the number of records in a compressed block and the block's size. This allows the block to be skipped if no values are accessed in it. However, when a value in the block is accessed, the entire block needs to be decompressed. LZO is generally chosen for the compression algorithm in favor of other strategies like ZLIB when low decompression overhead is more important than the compression ratio. We study both LZO and ZLIB in Section 5.3.

Dictionary Compressed Skip List: This scheme is tailored for map column types. It takes advantage of the fact that the keys used in maps are often strings that are drawn from a limited universe. Such strings are well suited for dictionary compression. We build a dictionary of keys for each block of map values and store the compressed keys in a map using a skip list format. This scheme often provides a worse compression ratio than LZO but compensates with lower CPU overhead for decompression. The main advantage of this scheme is that a value can be accessed without having to decompress an entire block of values.

3.6 Experiments

In this section, we present experimental results demonstrating that column-oriented storage techniques can be effectively exploited in Hadoop. We compare CIF with popular formats in use, namely text files (TXT), *SequenceFiles* (SEQ), and RCFile.

3.6.1 Experimental Setup

The experiments were run on a cluster with 42 nodes. Two nodes were reserved to run the Hadoop jobtracker and the namenode. The remaining 40 nodes were used for HDFS and MapReduce. Each node had 8 cores (via two quad-core 2.4Ghz sockets), 32GB of main memory, and five locally attached SATA disks. Datanodes spread their data across four of these disks. Hadoop version 0.21.0 was used, and was configured to run six mappers and one reducer per node.

3.6.2 Benefits of Column-Oriented Storage

The first experiment was a microbenchmark to verify that using CIF can indeed make scans faster compared to using SEQ and TXT. As in [63], these experiments were run on a single node of the cluster. Data was read using standard HDFS and *InputFormat* APIs. We present experiments using the full cluster in Section 3.6.3.

We used a synthetic dataset generated as follows: Each record consisted of 6 strings, 6 integers, and a map. The integers were randomly assigned values between 1 and 10000. Random strings of length between 20 and 40 were generated over readable ASCII characters. Each map consisted of 10 items, the keys were random strings of length 4, and the values were randomly chosen integers. The data was written out in each of the formats. For SEQ, *NullWritables* were used as the keys. A record containing the above fields was used as the value class. The total size of the dataset was 57GB in the SEQ format. The I/O transfer size – *io.file.buffer.size* – was set to 128K. This is a commonly configured value for many deployments. Repeating the experiment with 4KB and 1MB produced similar results and are omitted. The time to scan various projections of the dataset for each of the formats is shown in Figure 3.7. The filesystem cache was flushed before each experiment. For TXT and SEQ, the time to scan any projection was roughly the same and therefore only one value is reported.

Comparison with TXT: As shown in Figure 3.7, the scan time with SEQ was approximately 3x faster than TXT. This is because parsing each line of the text file quickly becomes CPU-bound while SEQ uses a binary format and does not need to run expensive parsing code for every record. This ratio could be even higher for more complex records. This is a straightforward way to dramatically improve on the naive use of text files in previous Hadoop performance studies [79, 26]. This confirms the criticism by Google engineers [49] where they argued these studies were flawed in comparing Hadoop on plain text files with a DBMS using binary

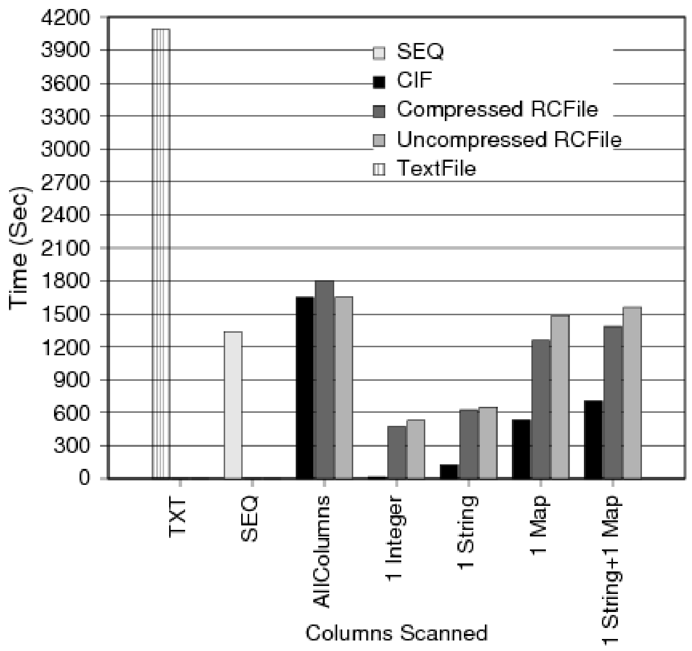


Figure 3.7: Microbenchmark comparing Text, SEQ, CIF, and RCFile.

formats.

Comparison with Sequence Files: When using CIF, the times for scanning a single integer, string, or map were 2.5x to 95x faster than SEQ. In each case, the speedup is directly attributable to the fact that CIF read much less data than SEQ. When scanning *all* the columns of the dataset CIF took about 25% longer than SEQ. This is because of the additional seeks that CIF incurred when gathering data from columns stored in different files. In all other cases, CIF is superior to both TXT and SEQ.

Comparison with RCFile: The uncompressed RCFile was approximately 69GB and the compressed RCFile was 43GB. The row-group size for RCFile was set to the recommended value of 4MB [60].

When a small number of columns were chosen from the dataset, CIF was more efficient than RCFile at eliminating unnecessary I/O. For the case of a single integer, CIF was nearly 38x faster than the uncompressed RCFile.

Measurements using *iostat* revealed that RCFile read 20x more bytes than CIF even when instructed to scan exactly one column. Additionally, it incurred more CPU overhead since it had to interpret the metadata blocks for approximately every 4MB of data.

When using a compressed RCFile, the running time improved. However, CIF was still faster in all cases. For instance, when a single integer was projected, CIF was 33x faster than the compressed RCFile. For the case where all the columns from the dataset were examined, CIF, compressed RCFile, and the uncompressed RCFile all had approximately the same performance. SEQ was 1.2x faster than the rest. Experiments in Section 3.7 show that CIF's advantage over RCFile holds for other values of the row-group size.

3.6.3 Comparison of Column Layouts

Next, we compared the performance of the different formats on a real dataset consisting of crawled pages for an intranet search application. This application currently uses Hadoop for its backend analytics. The data is acquired using Nutch [15] and the crawled data is stored in HDFS. The schema of the dataset used is described in Figure 3.2. Queries are frequently run against this data as the analytics pipeline is being constantly modified. Common tasks include determining if various pieces of metadata reported by a page are consistent. This includes fields like encoding, language, location, and other custom metadata.

We used the following MapReduce job for our experiment: determine all the distinct content-types reported by pages from IBM Japan i.e., URLs containing "ibm.com/jp". The content-type is an attribute that is part of the metadata map along with many other fields returned by an HTTP server in response to a request from the crawler. The code of the MapReduce job is very similar to the example in Figure 3.1. The content-type is one of the entries in the metadata column. The selectivity of the predicate

Layout	Data Read (GB)	Map Cost	Map Ratio	Total Time (sec)	Time Ratio
SEQ-uncomp	6400	339802	-	1482	-
SEQ-record	3008	196698	-	889	-
SEQ-block	2848	193338	-	886	-
SEQ-custom	3040	181056	1.0x	806	1.0x
RCFile	1113	168528	1.1x	761	1.1x
RCFile-comp	102	48389	3.7x	291	2.8x
CIF-ZLIB	36	3063	59.1x	77	10.4x
CIF	96	2978	60.8x	78	10.3x
CIF-LZO	54	2966	61.0x	79	10.2x
CIF-SL	75	2209	81.9x	70	11.5x
CIF-DCSL	61	1680	107.8x	63	12.8x

Table 3.1: Comparison of different storage formats. "Ratio" is the speedup vs. SEQ-custom, while "Map Cost" is the sum of all map task times, not wall-clock time.

on the URL was approximately 6%. We executed this job on a 6.4TB subset of the crawl dataset. The total amount of data per node was approximately 160GB.

With SEQ, we tried 4 variants: uncompressed (SEQ-uncomp), block-compressed (SEQ-block), record compressed (SEQ-record), and a custom format (SEQ-custom) which used an uncompressed sequence file, but compressed the content column using application specific code. We also included the time taken with RCFile with and without Zlib compression enabled (RCFile and RCFile-comp). For CIF, we laid out the metadata column in five different ways: default (CIF), CIF with skip lists (CIF-SL), CIF with block compression using LZO (CIF-LZO) and ZLIB (CIF-ZLIB), and CIF with dictionary compressed skip lists (CIF-DCSL). TXT is omitted from this experiment. *ColumnPlacementPolicy* (CPP) was used for all the CIF experiments.

Table 3.1 presents the time consumed along with the total bytes read from HDFS for each of the formats. The entries are presented in increasing order of performance. We report two numbers: the sum of the time consumed by all the map tasks in the job (map cost) and the overall job completion times. The map cost allows us to focus on the improvement in the resource usage during scans with different formats and isolate effects of scheduling delays between map and reduce which are present in wall clock times, especially with shorter jobs.

As shown, SEQ variants were generally the slowest since they also read the content field, which contains several KB of data for each record. SEQ-record and SEQ-block were both better than SEQ-uncomp by approximately 1.7x. SEQ-custom was the fastest by a small margin. The speedups for RCFile and CIF are computed with respect to SEQ-custom.

RCFile and RCFile-comp were better than SEQ-custom in map cost by 1.1x and 3.7x respectively. Both RCFile and RCFile-comp eliminate some of the I/O for unnecessary columns and read substantially less data than SEQ-custom.

CIF was 60.8x better than SEQ-custom in the map cost. This speedup is largely the result of 31.7x less data being read in CIF because of its column-oriented storage format. The rest of the difference is due to caching effects in the local filesystem improving the read performance of the smaller working set (96GB vs 3040GB across 40 nodes). CIF-ZLIB was slightly worse than CIF at 59.1x in spite of reading substantially less data – 36GB vs 96GB for CIF. CIF-LZO also read less data than CIF, but only slightly faster than CIF at 61.0x. This was largely because the CPU overhead of decompressing the data does not pay for the increased bandwidth available to the mappers. We also repeated the experiment with different compression block sizes and did not find a significant difference.

CIF-SL was 81.9x better than SEQ-custom and it also read fewer bytes of data. This speedup comes from the use of skip lists and the use of

LazyRecord to avoid deserializing the metadata column unless the URL contained "ibm.com/jp". CIF-DCSL was the best, providing a 107.8x improvement over SEQ-custom's map cost. The dictionary compression algorithm is extremely lightweight it also benefits from lazy record construction and the use of skip lists. Table 3.1 also shows the overall speedups. These are lower since the different formats do nothing to speed up the shuffle, sort, and reduce phases. These vary from 10.4 to 12.8x for the various column formats.

3.6.4 Impact of Co-Location

To measure the impact of co-location, we re-executed the same MapReduce job as above but this time using CIF with the default HDFS block placement policy rather than with CPP. The map cost with CPP was 5.1x better than the map cost without CPP. CIF with CPP was faster because CPP ensured that no column files had to be remotely accessed.

3.7 Additional Experiments

3.7.1 Cost of Deserialization

We conducted a simple experiment to illustrate the overhead involved in deserializing simple and complex types. We created a dataset with 1 million records, each 1000 bytes wide. We filled a given fraction f of the 1000 bytes with integers. The remainder of the record was filled with a byte array. The integers require deserialization when the data is scanned. The byte array can be read into the record without any deserialization. We vary the fraction f from 0.0 to 1.0. We measure the time taken to scan this entire dataset in a simple Java program. The data was written to a local file and the filesystem cache was warmed before reading the data. As a result, the entire dataset was present in memory, and no-disk I/O was

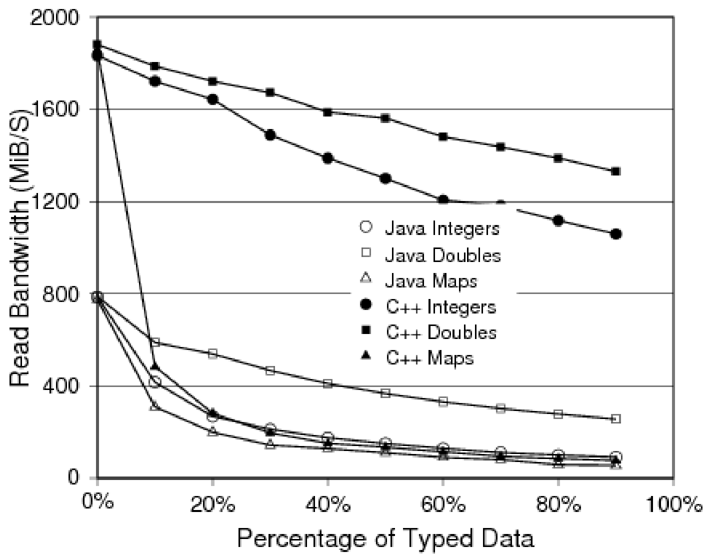


Figure 3.8: Microbenchmark examining overhead of serialization and object creation.

incurred in any of the cases. We also repeated the experiment in C++. The same single machine configuration that was described in Section 5.3 was used here.

Figure 3.8 shows the total read bandwidth measured while scanning this dataset as f was varied for different data types: integers, doubles, and maps. For the map implementations, we used *java.util.map* for Java and *std::map* for C++.

As shown, in each case, the read bandwidth drops as f increases. This is because deserializing typed data imposes a larger CPU overhead than reading a simply byte array. As explained in Section 3.3.2, Java suffers much more from this phenomenon than C++. The read bandwidth of the C++ program is substantially higher for integers and doubles than the Java program.

The case of deserializing maps is more interesting. Each map consisted of 4 entries, the keys were mutable strings and the values were integers. Since maps require new objects to be created, the total overhead of deseri-

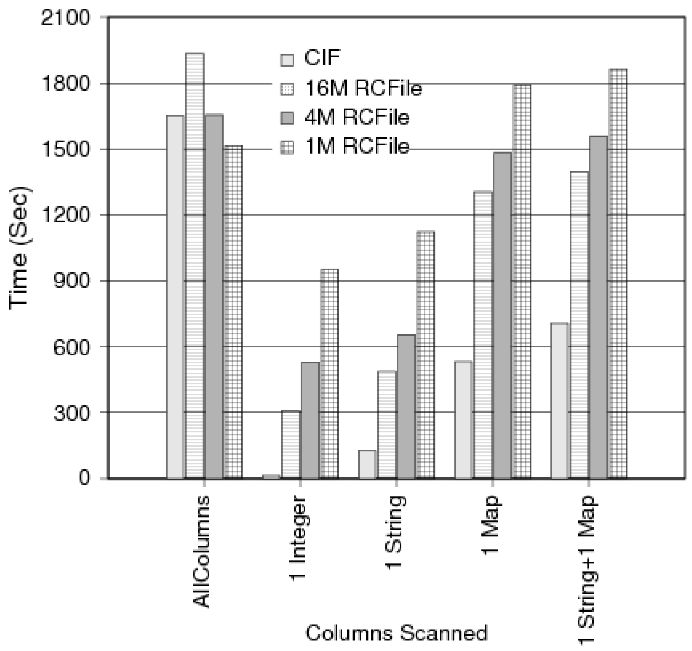


Figure 3.9: Tuning row-group size for RCFiles

alizing maps is substantially larger. In fact, as the figure shows, when f exceeds 60%, the rate at which maps are deserialized can be slower than the bandwidth of a typical SATA disk.

3.7.2 Tuning RCFile

We studied the impact of varying the row-group size of the RCFile on the scan tests described in Section 3.6.2. Using the same dataset as before, we varied the row-group sizes as 1MB, 4MB, and 16MB. The running times for scanning various projections are shown in Figure 3.9.

For the case where a single integer was scanned, CIF read a total of 415MB. RCFile read 16.5GB, 8.5GB, and 4.5 GB for the 1MB, 4MB, and 16MB row-group size settings respectivey. The larger row-group size clearly helped achieve better I/O elimination. However, a larger row-group size has an adverse impact on the benefits from lazy decompression

Layout	Time (min)
CIF	89
CIF-SL	93
RCFile	89

Table 3.2: Load times with synthetic data.

as described by the authors of RCFile [60]. By eliminating this additional tuning parameter, CIF is more robust and at the same time offers better performance than RCFile.

3.7.3 Load Times

We measured the time taken to convert the synthetic dataset used in Section 3.6.2 from SEQ to various formats. These are presented in Table 3.2. Observe that the overhead of adding skip lists to the CIF was fairly minor. We expect this cost to be representative. The additional overhead comes from the fact HDFS exposes an append-only API. While writing the output of a job, one can only append to the file. It is not possible to go back and alter any values. As a result, building skip lists requires double buffering the data so one can actually calculate the number of bytes for each skip pointer before writing the data to disk. With the current load algorithm, the largest skip is limited by the size of the main memory. Another observation from Table 3.2 is that converting to uncompressed RCFiles takes approximately the same amount of time as CIF. We do not expect the load utilities that convert data to CIF to be any worse than those that convert data to RCFile.

3.7.4 Varying Selectivity

Working with the same dataset as in Section 3.6.2, we measured the benefits of skip lists and lazy deserialization as the selectivity of the predicate in

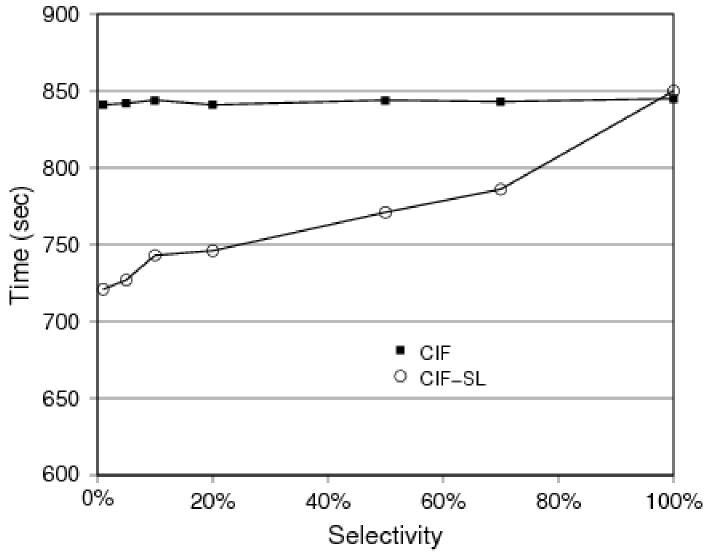


Figure 3.10: Benefits of lazy materialization and skip lists.

the map function was varied. We measured the time taken to aggregate the value in the map column under a given key for all the records where the string column satisfied a given pattern. We varied the selectivity of the predicate and measured the running time of the job. We compared the running time of CIF vs CIF-SL. The results are shown in Figure 3.10.

The figure shows that for highly selective queries, CIF-SL provides more savings by eliminating unnecessary deserialization and object creation. As the selectivity gets closer to 100% CIF-SL converges to the performance of CIF. The overhead for CIF-SL with respect to CIF at 100% selectivity is minor. The performance benefits of CIF-SL depend on the complex type, and the associated cost of deserializing it.

3.7.5 Varying Record Size

In this experiment, we compared the performance of CIF and RCFile as the number of columns in a record increases. We generated three datasets with 20, 40, and 80 columns per record. Each column contained a random

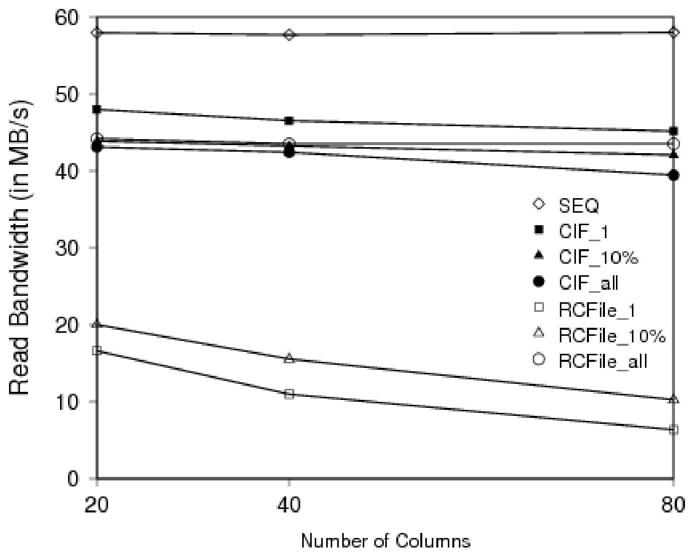


Figure 3.11: Comparison of CIF and RCFile as the number of columns in a record increases.

string of length 30. In each case, the total data size was approximately 60GB. We conducted three scan tests where we projected 1 column, 10% of the columns, or all the columns of the dataset. For the RCFile, we used 16MB as the row-group size. Figure 3.11 reports the read bandwidth measured for each of the scan tests.

The figure shows that when projecting a small number of columns, CIF performs better than RCFile in all cases. The overhead of CIF over SEQ when scanning all the columns of a dataset are greater as the number of columns in the dataset increases. This is consistent with previous research on the overheads of column oriented storage [70]. Another interesting observation is that as the number of columns in the dataset increases, the read bandwidth for reading a single column decreases for RCFile, while it remains relatively stable for CIF. With wider rows, the amount of data corresponding to a single column in a row group (16MB) decreases, and consequently the overheads associated with processing a row-group are amortized over fewer records.

3.8 Related Work

Our study presented in Chapter 2, and an older study [79] compared the performance of Hadoop with various parallel DBMSs to find that Hadoop suffers substantial performance penalties for query processing. Subsequent studies have tried to bring some of the technologies from DBMSs into MapReduce without affecting its advantages. HadoopDB [26] advocates using database nodes to do the actual work and relying on MapReduce only for scheduling and communication. A later study [51] points out many drawbacks of the HadoopDB approach and demonstrates, along with other independent efforts [63, 38, 52], how indexing can be incorporated into MapReduce in a substantially less disruptive manner. Incorporating optimization techniques from DBMSs for high-level languages like Pig has been suggested in [75]. Our focus in this thesis is on MapReduce programs written directly in Java. The database community has been interested in several other aspects of bridging the gap between MapReduce and DBMSs [61, 41].

This work draws on many of the techniques advocated in the literature for column oriented databases. The advantage of column-oriented storage for eliminating unnecessary I/O is well known. However, many of the advanced techniques used in a column-oriented runtime such as careful integration of compression with query execution [23, 69], late materialization [24], and use of SIMD instructions [35] and other organizing techniques [62] are challenging to adapt to a MapReduce environment without assuming a higher-level language and a special runtime for query execution.

A recent paper describes Dremel [70] – a column-oriented storage system at Google for large datasets involving nested types. The paper describes a technique for nested data that shreds the constituent fields into separate columns. It also provides algorithms for putting these complex types back together with only the portions requested by the query. Dremel uses a SQL-like language and a special runtime. In contrast to Dremel, we store complex types as a single column and do not shred it into separate columns. In addition to nested records, we also deal with maps, which is not a focus for Dremel. Our focus is on performance improvement in the context of Hadoop and Java. We believe our approach complements many of the techniques described in Dremel.

The Trojan Layouts [64] is another layout proposed for the MapReduce setting. This layout organizes data blocks using a PAX layout but colocates attributes together according to the workload and also uses different layouts for different data replicas. Our CIF layout is agnostic of the workload, doesn't use the PAX layout as a basis and keeps the same layout across all the data replicas.

3.9 Summary

The column-oriented storage techniques that have proven so successful in parallel DBMSs can also be used to dramatically improve the performance of a MapReduce system. However, translating these techniques to a MapReduce system such as Hadoop presents unique challenges because of different replication and scheduling constraints, the low-level MapReduce programming API, and the more common use of complex types in MapReduce jobs. In this chapter, we described a new column-oriented, binary storage format for Hadoop that is not only compatible with Hadoop's programming APIs but also requires no changes to the core of Hadoop. Our new storage format includes features such as lightweight compression and lazy record construction to avoid deserializing unwanted records. Experiments on real data were used to show that our column-oriented storage techniques can improve Hadoop's performance by up two orders of magnitude.

Chapter 4

Design and Evaluation of Online Replica Placement Algorithms

4.1 Introduction

As mission-critical workloads continue to move to cloud-based environments, there is an increased motivation for the Database-as-a-Service (DaaS) providers to provide performance and availability guarantees to their customers, typically in the form of Service Level Objectives (SLOs). From the DaaS provider's perspective, guaranteeing both the performance and the availability SLOs to all the tenants is a challenging task as simple methods that "safely" map tenant workload (so as to meet the performance SLOs) to nodes in the cluster leads to underprovisioning the hardware resources, which in turn increases the total operating cost for the DaaS provider. In multi-tenant environments where each tenant needs only a fraction of the resources of even a single node (e.g., in [32]), the degree of multi-tenant concurrency per node is high, which makes guaranteeing the performance SLOs challenging.

Another challenge is that the DaaS providers typically have an *estimate* of the workloads that they expect to serve and provision resources based on this estimate, but the *actual* workload characteristics may deviate from this estimate. An *online* data placement algorithm, as opposed to the offline placement techniques, such as [45], tackle exactly this situation – they find a placement for the replicas of a given tenant as soon as the tenant arrives at the system, and without assuming *a priori* knowledge of the workloads of the entire set of tenants. Thus the online placement techniques, should gracefully adjust to unexpected workload changes in case the actual observed workloads are different from the expected workloads.

In this thesis, we tackle the *online* replica placement problem. Our algorithms assume that the tenant's database is replicated a few times so that the availability SLOs are met. In the model that we consider in this chapter, there is a master/primary replica which drives the load on the slave/secondary replicas (e.g., as in [32]), by forwarding certain operations to them. One of our main concerns, is to examine how each replica placement algorithm distributes the load across all the machines. As reported in [32] having a balanced load typically helps in handling unexpected workload changes. This is because, a few aggressive tenants on a highly-loaded machine can make it difficult to continue meeting the performance SLOs of *all* the tenants.

In general, designing a data placement algorithm for multi-tenant DaaS environments is a challenging task, since the replica placement algorithm must: a) take into account the different performance requirements of all the tenants (performance SLOs), b) take into account the differences in the load between the replicas of the same tenant, c) not violate the replication constraints, d) aim to balance the load across all machines, e) minimize the total operating cost and f) gracefully adjust in unexpected workload changes. This work makes the following contributions:

- We formulate and systematically explore the *online* replica placement problem in multi-tenant DaaS environments which support both performance SLOs and the key mechanism (i.e., replication) that is used to meet availability SLOs.

- We develop a framework to address the problem above and present a set of evaluation criteria that examine the load balancing properties of different replica placement algorithms both during the initial placement and following a failure event. The framework also considers the impact on the total operating cost , as well as the adaptivity of the replica placement algorithm in case of unexpected workload changes. .

- We design and evaluate a number of algorithms for the replica placement problem, and find that an algorithm called **RkC** has very good load balancing properties, low cost, and adaptivity in changing workload characteristics under a variety of multi-tenant environments and also does not require global knowledge of the current load of all the machines, making the algorithm desirable in online distributed environments that lack of centralized control. Thus, this thesis makes the case for **RkC**.

To the best of our knowledge, this work is the first attempt to explore the online replica placement policies that do not require global knowledge of the load across the cluster, relate them with load balancing in DaaS settings, study their impact on the total operating cost and their adaptivity.

We note that studying the effects of different data placement mechanisms in cloud environments is a new and emerging area with many open problems. To the best of our knowledge, this work is the first attempt to explore the *online* replica placement policies, relate them with load balancing in DaaS settings, study their impact on the total operating cost and their adaptivity in changes in expected workload characteristics. In this thesis, we use the initial hardware provisioning cost as a proxy for the total operating cost. We recognize that other factors like the energy cost or related infrastructure costs are not taken into account, but these are often directly impacted by the number of machines provisioned, and considering these factors is an important direction for future work. In this thesis, we focus on workloads that exert a steady load on the cloud service (though we consider the impact of changing workload characteristics). Dealing with dynamic workload fluctuations is a rich area for future work, and potentially requires building workload characterization and prediction models (a rich area of research by itself) and adapting our framework to work with these models. We also note that our model for load balancing in case of failures is focused on the impact of the replica placement algorithm on a single node failure. Our work, shows that simple techniques that can be used during the initial replica placement process can actually help balance the load in case of single-node failures. Extending our model to incorporate multiple node failures as well as correlated failures, and studying what guarantees our replica placement algorithms can provide in these cases is an interesting path for future work. Finally this thesis, adresses homogeneous clusters. Extending our framework to accommodate heterogeneous clusters is part of future work.

4.2 Replica Placement Framework

This section describes our framework for designing and evaluating replica placement algorithms in multi-tenant cloud environments where the tenants have different performance SLOs.

4.2.1 Replica Placement Problem Formulation

Our framework assumes that each tenant's data is replicated n times across the cluster. Every replica is assigned an initial role, either the *primary* role or the *secondary* role. The load of a given replica may vary depending on its role. For example, the *primary* replica which is assigned the primary role, may handle both the read and the write operations of a workload. The remaining $n-1$ replicas are the *secondary* replicas and handle only the write operations that synchronously receive from the primary replica. In this scenario, the secondary replicas are used as a backup when a primary replica fails to process the workload.

In our framework, the tenants are split into tenant classes, each one corresponding to a different performance SLO. For example, assume that a DaaS provider has tenants that have workloads that are like TPC-C scale factor 10, and that the tenants are split into two classes. The performance metric in this case, is transactions per second (tps). The tenants in the 100tps class are associated with a high performance SLO of 100 tps, whereas the tenants in the 10tps class are associated with a lower performance SLO of 10 tps. In our environment, each tenant is placed in a separate database (similar to SQL Azure [32]). The tenants that belong to the same tenant class are assigned to the same DBMS instance.

In our problem statement, we consider two phases of the data placement process, namely the *initial hardware provisioning phase* and the *replica placement phase*. In the initial hardware provisioning phase the DaaS providers have an *estimate* of the *expected* number of tenants and they need to know the minimum number of machines M, they should provision based on this estimate. Then, they typically provision an additional number of machines to account for hardware failures, future growth, unexpected mix of new tenants, etc. Regardless, the DaaS provider needs to know the value of M. In the following phase (replica placement phase), the DaaS providers need to find a placement for the replicas of each incoming tenant. Note that during the provisioning phase the actual tenant workloads are not known in advance and in practice they may differ from the expected workloads that the DaaS provider considers when determining the number of machines that need to be provisioned. Thus, in our setting, we do not assume *a priori* knowledge of the actual workloads.

Problem Statement: Given a set of tenant classes $C : \{c_1, ..., c_T\}$, a replication factor n, a fill factor $f \geqslant 1$, an *expected* workload $W_e = (N_e, C, R_e, n)$ comprised of N_e tenants with corresponding tenant ratios $R_e : \{r_{e1}, \dots ,$ $r_{eT}\}$, find:

Output 1: The minimum number of machines M that need to be provisioned in order to place the N_e primary replicas and the $(n-1) * N_e$ secondary replicas, in a way that guarantees the performance SLOs of all the tenants and does not violate the replication constraints (*h*ardware provisioning phase).

Let $MS : \{m_1, ..., m_{M*f}\}$ be the set of provisioned machines, $Q_{pi} :$ $\{p_1, ..., p_i\} \rightarrow \{m_k^1, \dots , m_k^i\}$ the assignment of the primary replica p_j of the j^{th} tenant to machine m_k^j, where $m_k^j \in MS, \forall j, 1 \leqslant j \leqslant i$, and Q_{si}: $\{s_1, ..., s_{i*(n-1)}\} \rightarrow \{m_z^1, \dots , m_z^{i*(n-1)}\}$ the assignment of the corresponding secondary replicas to the machines, $m_z^j \in MS, 1 \leqslant j \leqslant i * (n-1)$. Given the incoming $(i+1)^{th}$ tenant find:

Output 2: The mapping $Q_{p(i+1)}$: $\{p_1, ..., p_{(i+1)}\} \rightarrow \{m_k^1, \dots , m_k^{(i+1)}\}$ by adding the primary replica of the $(i+1)^{th}$ tenant to Q_{pi} (*r*eplica placement phase).

Output 3: The mapping $Q_{s(i+1)}$: $\{s_1, ..., s_{(i+1)*(n-1)}\} \rightarrow \{m_z^1, \dots , m_z^{(i+1)*(n-1)}\}$ by adding the secondary replicas of the $(i+1)^{th}$ tenant to Q_{si} (*r*eplica placement phase).

The fill factor f determines the additional number of machines that the service provider provisions to account for unexpected events. In this thesis, we do not change the existing mapping of the replicas of the first i tenants when placing the $(i+1)^{th}$ tenant, in order to avoid transferring data over the network. Exploring the additional opportunities that may arise by remapping a subset of the already placed tenants, similar to [83, 84], but in our setting, is part of future work. Note that since the actual workload is not known in advance, heuristics that sort the incoming tenants based on their load characteristics, before the tenants are placed on the machines (e.g., offline bin packing heuristics like FFD [65]) are not applicable in our framework. Our algorithms are *o*nline placement algorithms which assign the replicas of each tenant to machines, as soon as the tenant arrives at the system.

4.2.2 Evaluation Criteria

In this section we present four criteria to evaluate data placement algorithms in multi-tenant cloud environments. These criteria cover a variety of factors that the DaaS providers consider when they accommodate tenants with different performance SLOs and they address both the *initial hardware provisioning phase (phase 1)* and the *replica placement phase (phase 2)*.

Since the actual workload observed may be different from the expected workload, we define the *actual* workload as $W_a = (N_a, C, R_a, n)$ where C, n are specified in the problem statement, N_a is the number of tenants that have already arrived at the system and $R_a : \{r_{a1}, \dots, r_{aT}\}$ are the tenant ratios. Based on this definition, our evaluation criteria are the following:

C1: The number of machines used (phase 1)

During the initial provisioning phase, the DaaS providers typically have an estimate of the expected number of tenants and they provision the hardware resources based on this estimate. Typically, the DaaS providers aim to minimize their total operating cost. As a result, among all the replica placement algorithms that can accommodate the expected number of tenants and their associated performance requirements, the algorithm(s) that use the fewest machines are preferred. Thus, given a function $H(a(W_e))$ which returns the number of machines that are used by algorithm a to place the replicas of the expected workload W_e, the DaaS providers would prefer the algorithm a for which:

$$M = \underset{a}{\operatorname{argmin}} H(a(W_e)) \tag{4.1}$$

C2: Load distribution across all machines (phases 1 and 2)

As discussed in the Introduction, in DaaS environments the functionality of the primary and secondary replicas is often different. For example, in SQL Azure [32] the primary replicas process more operations than the corresponding secondary replicas. Since the primary replicas actually do more work than the secondary replicas, the DaaS providers prefer to host a mix of primary and secondary replicas in each server with the goal of balancing the load across all the machines [32]. The C2 criterion targets this property of the data placement algorithms, and focuses on the distribution of the load across all the available machines. Note that this criterion targets not only the replica placement phase but also the initial hardware provisioning phase. This is because, from all the possible algorithms used to place the replicas of the expected workload W_e, the ones that are able to balance the load are desirable.

Let $L_p(c_i)$ denote a function that returns the load generated by a primary replica of tenant class c_i, and $L_s(c_i)$ denote a function that returns the load generated by a secondary replica of tenant class c_i. If the number of primary replicas of tenant class c_i placed on machine m_j is p_{ij}, and the number of secondary replicas of tenant class c_i placed on that machine is s_{ij}, then for this machine, the load Lj can be defined as $L_j = \sum_{i=1}^{T} p_{ij} * L_p(c_i) + \sum_{i=1}^{T} s_{ij} * L_s(c_i)$. Given a function $X(a(W, M)) = \sigma_l$ that returns the standard deviation σ_l of the load values $[L_1 ... L_M]$ produced by algorithm a, on a set of M machines, then the algorithms that return uniform distributions of the load are desirable. That is, from the set of all replica placement algorithms, DaaS providers would prefer the algorithm a for which:

$$\sigma_l = \underset{a}{\operatorname{argmin}}\, X(a(W, M)) \tag{4.2}$$

C3: Load generated by a failed machine (phases 1 and 2)

The C3 criterion is related to load balancing in the presence of failures. When a machine that contains primary replicas is unavailable (for any reason including hardware failure on that node), a set of new primary replicas will be created for the database with primary copies on the failed node. This operation is executed by promoting an existing secondary replica that is placed on a different machine as the new primary. During this operation, it is desirable to not overload any machine by assigning to it a disproportionate number of new primary replicas. As discussed in [32], it is desirable to spread the load created by a failed machine across as many machines as possible. For example, an algorithm like Round Robin that assigns a primary replica on one machine and its corresponding $n-1$ secondary replicas on the $n-1$ neighboring machines is a poor choice with respect to criterion C3. If one machine fails, then all the primary replicas hosted on that machine will be re-assigned to only $n-1$ machines. Since the replication factor n is typically a small number (e.g., 3), the load will not be spread evenly across all the remaining machines in the cluster.

Let LF_{jv} be the load generated on machine m_j when the machine m_v fails. This load is created by promoting some of the secondary replicas on machine m_j to primary replicas. Given a function $Y(a(W, M, v)) = \sigma_{fv}$ that returns the standard deviation σ_{fv} of the load values $[LF_{1v}...LF_{Mv}]$ produced by algorithm a, on a set of M machines when m_v fails. Let d_a be the maximum difference between any two standard deviations produced by the function Y on a given set of machines for a fixed workload W and algorithm a. From the set of all replica placement algorithms, the preferred algorithm a is one for which:

$$d_a = \underset{a}{\operatorname{argmin}}\,[max(|Y(a(W, M, i)) - Y(a(W, M, j))|)] \qquad \forall i, j \; 1 \leqslant i, j \leqslant M \tag{4.3}$$

Note that due to this objective, the well-known heuristics used for the online bin packing problem (e.g. first fit, best fit etc.) are not directly applicable in our setting. This is because these heuristics take into account the load of the given machine (e.g., best fit) and/or the ordering of the machines (e.g., first fit). Criterion C3, on the other hand, needs heuristics that consider the relationship between the machine used to place a given secondary replica, the location of its corresponding primary replica and the locations of the primary replicas which have their corresponding secondaries on that machine. In Section 4.4, we present the additional heuristics needed to account for this objective. Criterion C3 does not guarantee that after the load of a failed machine is redistributed across the cluster, all the tenants on the remaining machines will meet the performance SLOs. It guarantees though, that the initial data placement is good enough so that the failure-handling algorithms will be able to spread the load as uniformly as possible without migrating replicas.

C4: The number of tenants served when $W_a \neq W_e$ (phase 2)

In case the actual workload W_a observed, is different from the expected workload W_e, the service providers would like the replica placement algorithm to gracefully adapt to the change. This translates into being able to pack as many tenants as possible on the provisioned number of machines M*f. Given a function $Z(a(W_a, M * f)) = N$ that returns the number of tenants from W_a that can be placed on M*f machines using algorithm a, the preferred algorithm a is one for which:

$$N = \underset{a}{\operatorname{argmax}}\, Z(a(W_a, M * f)) \tag{4.4}$$

4.2.3 Workload Characterization

Tenant placement requires workload characterization so that the impact on existing tenants when a new tenant is placed on a machine, is quantified. This problem has been studied in the context of different database environments [45, 68, 53, 55, 83]. Some of the approaches proposed in the literature for workload characterization in multi-tenant database environments include the use of machine learning and prediction models [55, 53, 83], empirical models [45] or direct benchmarking [68]. In our framework, we also need to characterize the hardware that is used to build each node in the cluster, so that we know what mixes of primaries and secondaries can be "safely" (i.e., meet the performance SLOs) placed on each node. Given tenants with different SLOs and replicas with different loads, we need to characterize the performance that can be delivered by each machine to each tenant class c_i. For this purpose, we use a machine characterizing function. As in [68], we directly benchmarked the tenants' to generate the machine characterizing function. However, machine learning techniques or prediction models could also be used, especially in cases where multiple tenant classes are involved. The general methodology for generating the characterizing function using benchmarking when multiple tenant classes are involved can be found in Section 4.3. The methodology is an extension of the one presented in [68] that also accounts for the existence of replicas.

Figure 4.1 shows an example of a machine characterizing function for a workload comprised of 1GB TPC-C tenants with 10 warehouses. TPC-C is a well known database benchmark and has been used before to study DaaS environments [46, 68]. All the tenants belong to one tenant class whose performance SLO is 10 tps. This characterizing function presents how the number of primary replicas and the number of secondary replicas can be varied on a machine so that this mix does not prevent any tenant from achieving its 10tps performance target. As a performance metric, we use the throughput of the *new-order* transactions, as is done for reporting TPC-C results (more specifically, the average transactions per second over a time interval of 900 seconds). The measurements are taken on a server with 16GB of main memory and two Intel E5410 quad-cores running at 2.33GHz. The node runs Microsoft SQL Server 2012, and each tenant has its own database file and its own log file. All the data files are spread across five 136 GB 10K RPM SAS drives. The log files are stored on a separate hard disk. The details on how to generate the machine characterizing function are presented in Section 4.3.

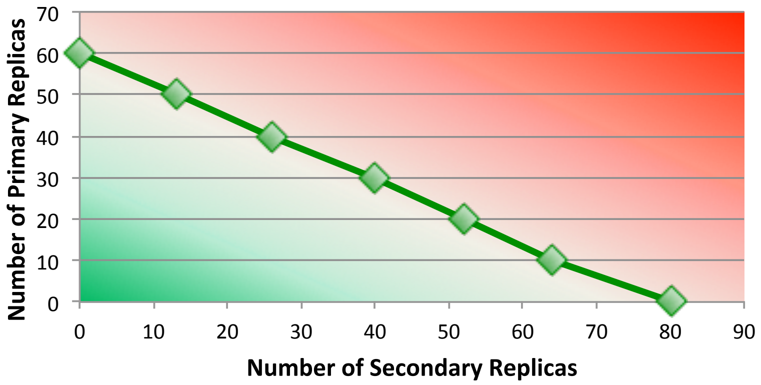


Figure 4.1: Machine Characterizing Function for the 10tps Tenant Class.

As shown, in Figure 4.1 the machine characterizing function for the 10 tps tenant class is almost linear and can be described by the following equation: $4 * p_1 + 3 * s_1 - 240 = 0$. Each diamond point on this line represents an actual benchmark test that we ran. At each of these data points the machine is disk-bound. Having defined the machine characterizing function, the next question is to find acceptable operating zones that deliver the required performance to each tenant. This zone is the one below or on the frontier defined by the machine characterizing function, which is defined by the following inequality: $4 * p_{1j} + 3 * s_{1j} - 240 \leqslant 0$. We can generalize this example to incorporate multiple tenant classes for which the machine characterizing function is multi-linear.

Definition 1. *For a given machine m_j, T tenant classes, $\overrightarrow{p} = [p_{1j} \ldots \; p_{Tj}]$ and $\overrightarrow{s} = [s_{1j} \ldots \; s_{Tj}]$ where p_{ij} represents the number of primaries of class c_i scheduled on machine m_j and s_{ij} represents the number of secondaries of class c_i scheduled on machine m_j, $1 \leqslant i \leqslant T$, $1 \leqslant j \leqslant M$, and a machine characterizing function that can be described by a multi-linear equation of the form:*

$$a_1 * p_1 + b_1 * s_1 + \ldots + a_T * p_T + b_T * s_T + d = 0 \tag{4.5}$$

where $a_i, b_i > 0$, $\forall i \; 1 \leqslant i \leqslant T$ and $d < 0$, all the primary and secondary replicas scheduled on the machine deliver the required performance if $a_1 * p_{1j} + b_1 * s_{1j} + \ldots + a_T * p_{Tj} + b_T * s_{Tj} + d \leqslant 0$.

In this chapter, we only focus on multi-linear characterizing functions. Previous work on characterizing main-memory database workloads has shown that the resource models in these environments are typically linear [83]. For this reason, linear models have been used for tenant placement in main-memory clusters [84]. The authors of [45] also observe additivity of CPU and memory consumption in disk-based systems. Modeling shared disk I/O is more challenging [45, 68]. However, our real experiments using TPC-C on the 10tps and 100tps tenant classes, on a disk-based system have also produced almost linear functions. Relaxing this assumption and considering the impact of other functions, to capture a larger variety of systems is an important direction for future work.

The algorithms and the techniques described in this paper can also be applied on non-linear machine characterizing functions, given that a multi-linear function can be drawn within the area that is below the frontier of the given machine characterizing function. In this case, the algorithms that are designed to work well for criteria C2 and C3 on multi-linear characterizing functions would continue to produce good results with respect to these criteria. Using this technique, the load would still be balanced. However, the minimum number of machines used (criterion C1) could vary with different performance characterizing functions. An interesting part of future work, is to study the opportunities that arise when non-linear characterizing functions are encountered.

Another interesting point to note is that if the machine characteristizing function is not linear, then there are other interesting aspect of systems research, including understanding why the function is not linear (e.g. is some resource the bottleneck when shared across tenants), and leads to questions about whether there are changes to the underlying software that could be made to produce a more "efficient" machine characterizing function. In fact, this line of thinking leads to a number of interesting research questions in its own right, including asking what is the best hardware one can buy for a given budget (e.g. should the machine have more memory, or fewer disks but more flash storage, etc.). These problems are beyond the scope of this initial work.

4.3 Obtaining the machine characterizing function using benchmarking

In the following paragraphs, we describe our methodology to obtain Figure 4.1 of Section 4.2 using benchmarking.

We followed the approach presented in [68], to deploy multiple tenants on the same machine and placed each tenant in a separate database. In [68] all the tenants' databases that belong to the same tenant class are placed on the same DBMS instance. This work, though, does not consider the existence of multiple replicas of the same database. In our setting, each tenant has one primary and one secondary replica ($n = 2$). The primary replica handles both the read and the write operations of the workload and synchronously forwards all the write operations to the secondary replica. Since the load handled by the primary and secondary replicas of one tenant class is not the same, we use two SQL Server instances to place the replicas of the same tenant class. The first DBMS instance holds all the primary replicas and the second DBMS instance holds all the secondary replicas of the tenants that belong to the same tenant class. The performance of each SQL Server instance is throttled by limiting the amount of memory allocated to it. Exploring other placement options is an interesting direction for future work.

Since the primary and secondary replicas on the same tenant cannot be placed on the same machine (*test* machine), due to replication constraints, to be able to fully characterize the workload performance, we used additional machines of the same configuration to place the extra replicas. Our methodology is the following:

1. We first find the maximum number of primary replicas that can be placed on the *test* machine (this step is repeated for each tenant class if multiple classes are considered). For this purpose, one additional machine that holds all the corresponding secondary replicas is used. (Since the load on a primary replica is higher than the load of its corresponding secondary replica, we expect to be able to host more secondary replicas on one machine than primary replicas of the same tenant class). As shown in figure 4.1, we can host up to 60 primary replicas of the 10 tps tenant class on the *test* machine. If more primary replicas are placed, then the *test* machine becomes the bottleneck and some tenants will not be able to meet the 10tps SLO.

2. We now investigate how many secondary replicas of the 10 tps tenant class can be placed on the *test* machine this step is repeated for each tenant class if multiple classes are considered). The corresponding primary replicas are placed on as many machines as needed so that each one does not hold more than 60 primary replicas. Figure 4.1

shows that we can place up to 80 secondary replicas on the *test* machine. If we attempt to place more than 80 secondary replicas, then the machine becomes the bottleneck and the corresponding primary replicas on the remaining machines will not be able to handle a load of 10 tps.

3. When the *test* machine contains a mix of primary and secondary replicas, we use as many additional machines as needed to place the corresponding secondary and primary replicas, respectively. Each additional machine holds replicas of one type only, and the number of replicas placed on the machine cannot be greater than 60 if these are primary replicas, or 80 in case these are secondary replicas. After having estimated the maximum number of primary replicas that can be placed on the *test* machine, we systematically substitute a fixed number of the primary replicas with secondary replicas. For each sample, we run a benchmark with the current mix of replicas and record the observed per-tenant performance at all the machines which hold primary replicas. If all the performance SLOs are satisfied, we also try adding more secondary replicas on the *test* machine (and the corresponding primary replicas on the additional machines) and repeat the experiment. We keep increasing the number of secondary replicas till the performance SLO of a tenant (in any of the machines used) falls below 10 tps. In this case, we know that we reached the boundary of the machine characterizing function. (If multiple tenant classes are considered, we can iterate through fixed size combinations of primary and secondary replicas of the other tenant classes).

4.3.1 Estimating the Number of Machines

In this section, we present the *lower bound* on the number of machines used to accommodate the performance SLOs of all the tenants. Assume that a replica placement algorithm that guarantees all the tenants' SLOs and does not violate the replication constraints uses M machines, $M \geqslant n$, where *n* is the replication factor. Then, given T tenant classes, *n* tenants, the corresponding tenant ratios $\{r_1, \ldots, r_T\}$ and a multi-linear machine characterizing function the lower bound M^* for the number of machines can be estimated using the following procedure:

$$\begin{aligned} &\forall j \; 1 \leqslant j \leqslant M \\ a_1 * p_{1j} + b_1 * s_{1j} + \ldots + a_T * p_{Tj} + b_T * s_{Tj} + d &\leqslant 0 \\ \text{where } a_i, b_i > 0,\; 1 \leqslant i \leqslant T \text{ and } d &< 0 \end{aligned}$$

This expression ensures that the combination of primary and secondary replicas, on *every* machine, is a point placed on or below the frontier defined by the multi-linear machine characterizing function. By adding the M inequalities,we get:

$$\begin{aligned} a_1 * \sum_{j=1}^{M} p_{1j} + b_1 * \sum_{j=1}^{M} s_{1j} + \ldots + a_T * \sum_{j=1}^{M} p_{Tj} + b_T * \sum_{j=1}^{M} s_{Tj} \\ +M * d \leqslant 0 \Rightarrow \\ a_1 * r_1 * N + b_1 * r_1 * (n-1) * N + \ldots + \\ a_T * r_T * N + b_T * r_T * (n-1) * N + M * d \leqslant 0 \end{aligned}$$

From the last inequality we can find a lower bound M^* for the number of machines used by any replica placement algorithm that guarantees the performance SLOs for all tenants. Since this number must be an integer at least equal to n, we have:

$$M \geqslant max(\lceil N * [\sum_{i=1}^{T} a_i * r_i + (n-1) * \sum_{i=1}^{T} b_i * r_i]/(-d)\rceil, n) \Rightarrow$$

$$M^* = max(\lceil N * [\sum_{i=1}^{T} a_i * r_i + (n-1) * \sum_{i=1}^{T} b_i * r_i]/(-d)\rceil, n) \qquad (4.6)$$

The lower bound of expression 4.6 is a positive number, since $d < 0$. Note, that this bound denotes the minimum number of machines needed to guarantee the performance SLOs of the tenants. However, there is no guarantee that there exists a replica placement algorithm that meets all the replication constraints and at the same time can use the number of machines specified by this lower bound. In Section 4.5, we show how the number of machines needed by the algorithms studied in this work, compares to this lower bound. Algorithm 1 finds the number of machines M that a replica placement algorithm a needs (Output 1).

Algorithm 1: Find the Minimum Number of Machines for algorithm a

```
Data: Algorithm: $a$, Number of tenants: $n$
Result: $M$
found ← 0;
while (found==0) do
    if (algorithm a can produce replica mappings that meet the replication constraints using $M$ machines)
    then
        for $j \leftarrow 1$ to $M$ do
            if ($a_1 * p_{1j} + b_1 * s_{1j} + ... + a_T * p_{Tj} + b_T * s_{Tj} + d \leqslant 0$) then
                $M \leftarrow M++$;
                continue;
            end
        end
        found ← 1;
    end
end
return M;
```

4.4 Replica Placement Algorithms

In this section we present a set of replica placement algorithms that we study in this thesis and discuss some of their properties. In Section 4.5, we evaluate these algorithms using the evaluation criteria presented in Section 4.2.2.

Our algorithms are *online* algorithms and do not assume *a priori* knowledge of the actual workload. When a tenant first arrives, all the replica placement algorithms find a machine for the tenant's primary replica and then place the $n-1$ secondary replicas one by one. In this chapter, we study five algorithms, namely: the *First Fit (FF)* algorithm, the *Static Round Robin (SRR)* algorithm, the *Static Round Robin-Scattered (SRR-S)* algorithm, the *k-Random Choices (Rk)* algorithm and *k-Random Choices with Constraints (RkC)* algorithm. The *SRR* and *SRR-S* algorithms are static algorithms which are designed to be used only as **reference** algorithms with respect to criteria C2 and C3 respectively. The *FF* algorithm is a generalization of the First Fit heuristic used to solve the online bin packing problem and is used to evaluate how an online bin packing algorithm performs for the replica placement problem as defined in this chapter. Finally, the *Rk* and *RkC* algorithms are more general algorithms that can also address systems where central tracking of the state of the replica assignments may not be feasible.

The algorithms, aim to balance the load across the available machines, by uniformly spreading the primary replicas and the secondary replicas of each tenant class across all the machines. We acknowledge that there could be other ways that can balance the load uniformly across all the machines (criterion C2), but our results show that there exist simple heuristic-based algorithms that produce almost uniform load distributions and at the same time the number of machines they use is no more than 5% over the theoretical lower bound defined in Section 4.2. Similarly, regarding criterion C3, our heuristics aim to spread the corresponding secondary replicas of a given primary replica across all the valid machines (based on the replication constraints). In this way, the algorithms that handle failures can exploit this placement of the secondary replicas to produce uniform load distributions after a node failure.

4.4.1 First Fit

The first algorithm that we study, is the ***First Fit (FF)*** algorithm. This algorithm is based on the well-known First Fit [65] heuristic strategy that has been used to solve the online bin packing problem, and has also been studied in the context of virtual machine [78] and database placement [89]. Note, we chose to describe the *FF* algorithm and not other bin packing algorithms like Best Fit, because this algorithm uses only the ordering of the machines as a criterion to place a given replica and does not require global knowledge of the load across all the machines. *F*F aims to minimize the number of machines that are used to accommodate all the tenants (Criterion C1). This algorithm differs from the traditional First Fit heuristic by taking into consideration the existence of replication constraints and the different performance requirements of the tenants. The *FF* algorithm assumes a fixed ordering of the machines. For each incoming tenant, the algorithm finds the first machine that can host its primary replica without creating any performance bottlenecks (based on a characterizing function as presented in Definition 1). Then, for each of the tenant's secondary replicas, the algorithm finds a set of candidate machines that could accommodate the replica without violating the replication constraints, and then places the replica on the first machine that can host it without creating any performance bottlenecks.

4.4.2 Static Round Robin and Static Round Robin – Scattered

The next two algorithms (SRR and SSR-S) that we study are specifically designed to take into account the load balancing criteria presented in Section 4.2.2, and use heuristics to produce good results with respect to these criteria. As a result, they both constitute a good **reference** point when evaluating the load distributions produced by the other replica placement algorithms that are examined in this section. Both the SRR and the SRR-S algorithms aim to assign the same number of primary and secondary replicas per tenant class on every machine. These algorithms are static methods since they pre-determine the load distribution across all the machines (based on the tenant ratios of the expected workload W_e). Consequently, the load generated by these algorithms is almost uniform.

The Static Round Robin (SRR) algorithm aims to balance the load of all the primary replicas and all the secondary replicas (Criterion C2). The Static Round Robin–Scattered (SRR-S) algorithm is based on *SRR* but also attempts to balance the load in case of failures (Criterion C3).

Both the *SRR* and the *SRR-S* algorithms assume that all the machines will host the same number of primary and secondary replicas for each tenant class. For example, assume that a given DaaS environment supports two tenant classes with the corresponding expected tenant ratios: $r_1 = 10\%$ and $r_2 = 90\%$. The *SRR* and *SRR-S* algorithms aim to spread the load in such a way that each machine hosts z primary replicas and $(n-1)*z$ secondary replicas of tenant class c_1, and $9*z$ primary replicas and $9*(n-1)*z$ of tenant class c_2.

The number z denotes the number of primary replicas of the tenant class with the lowest tenant ratio, that can be placed on one machine. Let $r_{min} = min\{r_1, ..., r_T\}$ where r_i is the tenant ratio for class c_i, $1 \leqslant i \leqslant T$. The *SRR* and *SRR-S* algorithms use the point $Q(r_1/r_{min}*z, r_1/r_{min}*(n-1)*z, ..., r_T/r_{min}*z, r_T/r_{min}*(n-1)*z)$ that is placed on the multi-linear machine characterizing function:

$$a_1 * r_1/r_{min} * z + b_1 * r_1/r_{min} * (n-1) * z + ... +$$
$$a_T * r_T/r_{min} * z + b_T * r_T/r_{min} * (n-1) * z + d = 0 \Rightarrow$$
$$z = (-d) * r_{min}/[\sum_{i=1}^{T} a_i * r_i + (n-1) * \sum_{i=1}^{T} b_i * r_i]$$

The number z could also be a real number. Since, the number of replicas on a given machine is an integer, the *SRR* and *SRR-S* algorithms use the point $Q_f = (r_1/r_{min} * \lfloor z \rfloor, r_1/r_{min} * (n-1) * \lfloor z \rfloor, ..., r_T/r_{min} * \lfloor z \rfloor, r_T/r_{min} * (n-1) * \lfloor z \rfloor)$ which is guaranteed to be on or below the frontier defined by the machine characterizing function (given that point Q satisfies this property). Note, that the *SRR* and the *SRR-S* algorithms cannot be used when $z < 1$. This is the case when the machine capacity is not enough to accommodate the number of replicas that the *SRR* and the *SRR-S* algorithms assume that a machine should host.

The number of machines used by *SRR* and *SRR-S* is then:

$$M \geqslant max(\lceil r_{min} * N/\lfloor z \rfloor \rceil, n) \tag{4.7}$$

where N is the total number of tenants.

Then, using Equation 4.6, M^* can be rewritten using the number z as follows:

$$M^* = max(\lceil r_{min} * N/z \rceil, n) \tag{4.8}$$

From Equations 4.7 and 4.8 we observe that the ratio $z/\lfloor z \rfloor$ largely determines how close the number of machines that are used by the *SRR* and the *SRR-S* algorithms are to the lower bound M^*. Let $z = X + \delta$ where X is a positive integer and $0 \leqslant \delta < 1$. Then:

$$z/\lfloor z \rfloor = (X + \delta)/X = 1 + \delta/X$$

In the worst case, $X = 1$ and $\delta = 1 - \epsilon$, which means that $\lfloor z \rfloor \approx 2 * z$. In this case, the *SRR* and the *SRR-S* algorithms will use approximately twice the lower bound on the number of machines given by Equation 4.6.

For every primary replica of an incoming tenant of class c_i, the *SRR* and the *SRR-S* algorithms find the set of machines with the same lowest load (S_p) of primary replicas (p_i) of class c_i. Then, both algorithms pick a random machine m_p from S_p and place the replica there. This approach is similar to the *worst fit* heuristic, used for the bin packing problem. The intuition is that using this heuristic, the load will be balanced across all the machines at any given point in time which is not possible when using a best fit approach. For every secondary replica, the *SRR* and the *SRR-S* algorithms find a set of candidate machines S_{rc} that meet the replication constraints. Then, they both find the set of machines S_s from S_{rc} that have the same lowest load of secondary replicas (s_i) for class c_i. The *SRR* algorithm picks a random machine from S_s to place the replica. In contrast, the *SRR-S* algorithm, which is optimized to balance the load during failures (criterion C3), keeps from S_s the machines with the lowest load of ss_i^p, where ss_i^p is the number of secondaries of class c_i that have their corresponding primary replica on machine m_p. This is the set S_{ss}. Finally, it randomly picks one machine from this set S_{ss}, and assigns the secondary replica to that machine. This heuristic guarantees that the load generated after failure of the machine m_p, can be redistributed to as any machines as possible.

4.4.3 k-Random Choices

The next algorithm is called the ***k-Random Choices (Rk)*** algorithm. *Rk* is essentially a family of algorithms, each with a different value of *k*, $1 \leqslant k \leqslant M$. For each primary replica of a tenant class c_i, $1 \leqslant i \leqslant T$, the *Rk* algorithm randomly picks *k* machines, and then finds the set of machines S_p that have the lowest number of primary replicas for the tenant class c_i. Then, it picks a random machine m_p from S_p to place the primary replica. For each secondary replica of a given tenant class, the *Rk* algorithm finds a candidate set of machines that can host the replica based on the replication constraints (S_{rc}) and randomly picks *k* machines out of S_{rc}. Then, it finds the set of machines S_s that have the lowest number of secondary replicas. As with the *SRR-S* algorithm, the *Rk* algorithm uses the same heuristic to optimize for load balancing during failures. More specifically, the *Rk* algorithm assigns the secondary replica to one of the machines in S_s that has the lowest number of secondary replicas for tenant class c_i with the corresponding primary replica on machine m_p.

The major difference between the *Rk* algorithm and both the *SRR* and the *SRR-S* algorithms is that the *Rk* algorithm considers only *k* machines when placing a replica and not the whole cluster. Moreover, the *Rk* algorithm is a dynamic and not a static algorithm, since it does not predetermine the replica distributions across the cluster. The fact that the *Rk* algorithm uses information about only *k* machines, and not about the whole cluster, makes the algorithm desirable in cases where a machine has to be contacted in order to get information about its current load. In this type of environments, contacting all the machines in the cluster may significantly slow down the replica placement process. In our experiments, we examine how similar the load distributions produced by the *Rk* algorithm are to those produced by the reference algorithms, *SRR* and *SRR-S*.

When $k = 1$ the algorithm makes only one random choice for each replica, so the load on the machines is not taken into consideration. The authors of [31, 72] showed that having just two random choices, instead of one, results in a large reduction in the maximum load of a machine, thus making this technique very useful in situations where load balancing is important. This technique has been applied in a variety of different environments (e.g., cuckoo hashing [77], peer-to-peer systems [37] etc). To the best of our knowledge, this is the first time this technique is applied to the replica assignment problem in multi-tenant cloud environments.

4.4.4 k-Random Choices with Constraints

Note, that the *Rk* algorithm does not make use of the machine characterizing function. Instead, it directly places the replica on the machine with the lowest load. The next family of algorithms studied, called k-Random Choices with Constraints (RkC) is similar to the *Rk* family of algorithms, but it also considers the machine characterizing function to exclude the machines that cannot accommodate the replica due to violations of the performance SLO requirements. For each primary replica, the *RkC* algorithm randomly picks *k* machines, checks which ones cannot host the primary replica (based on the characterizing function), and then disregards those. The *RkC* algorithm keeps picking machines until it finds *k* machines that can host the replica or until the entire set of machines has been examined. It then continues in the same way as the *Rk* algorithm. The *RkC* algorithm follows a similar process for the secondary replicas. The pseudocode for the *RkC* algorithm is shown in Algorithm 2.

The *RkC* algorithm finds the minimum load on the *k* machines, before it decides which machine will host the replica. This operation is repeated two times for each tenant's secondary replica (given that the second heuristic is applied), and one time for each tenant's primary replica. When $k = M$, the algorithm's complexity is $O(n * M)$.

In Section 4.5, we show how the *Rk* and the *RkC* algorithms compare with respect to our evaluation criteria. More specifically, we present results

Algorithm 2: k-Random Choices with Constraints (RkC)

```
Data: Incoming tenant: $i + 1$, Replication factor: $n$, Set of machines: $MS$, Value of k: $k$, $Q_{pi}$, $Q_{si}$
Result: $Q_{p(i+1)}$, $Q_{s(i+1)}$
c ← findTenantClass(i+1);
$Q_{p(i+1)} \leftarrow Q_{pi}$;
$Q_{s(i+1)} \leftarrow Q_{si}$;
$S_k$ ← Pick randomly $k$ machines out of $MS$;
$S_k$ ← Remove from $S_k$ the machines which cannot hold the replica based on the multi-linear characterizing function;
while ($|S_k| < k$ and exist machines in $MS$ that have not been examined) do
    $S_{tmp}$ ← Pick randomly k-$|S_k|$ machines out of the machines in $MS$ that have not been examined;
    $S_k \leftarrow S_k + S_{tmp}$;
    $S_k$ ← Remove the machines from $S_k$ which cannot hold the replica based on the multi-linear characterizing function;
end
if ($|S_k|$ == 0) then
    exit;
end
$S_p$ ← Pick the set of machines from $S_k$ with the minimum number of primaries of class $c$.;
$m_p$ ← Pick randomly 1 machine from $S_p$;
$Q_{p(i+1)}.add(p_{(i+1)}, m_p)$;
for j ← 1 to $n - 1$ do
    $S_{rc}$ ← Pick the set of machines from $MS$ that meet the replication constraints;
    $S_k$ ← Pick randomly $k$ machines out of $S_{rc}$;
    $S_k$ ← Remove from $S_k$ the machines which cannot hold the replica based on the multi-linear characterizing function;
    while ($|S_k| < k$ and exist machines in $S_{rc}$ that have not been examined) do
        $S_{tmp}$ ← Pick randomly k-$|S_k|$ machines out of the machines in $S_{rc}$ that have not been examined;
        $S_k \leftarrow S_k + S_{tmp}$;
        $S_k$ ← Remove from $S_k$ the machines which cannot hold the replica based on the multi-linear characterizing function;
    end
    if ($|S_k|$ == 0) then
        exit;
    end
    $S_s$ ← Pick the set of machines from $S_k$ with the minimum number of secondaries of class $c$.;
    $S_{ss}$ ← Pick the set of machines from $S_s$ with the minimum number of secondaries of class c whose primary is placed on $m_p$;
    $m_q$ ← Pick randomly 1 machine from $S_{ss}$;
    $w \leftarrow ((i + 1) - 1) * (n - 1) + j$;
    $Q_{s(i+1)}.add(s_w, m_q)$;
end
```

that highlight the effect of using the machine characterizing function on the number of machines that need to be provisioned. We also evaluate the impact of static (*SRR, SRR-S*) and dynamic (*RkC*) policies on each algorithm's ability to gracefully handle workload changes (Criterion C4).

4.5 Experimental Evaluation

In this section we compare the algorithms presented in Section 4.4 and evaluate them using the three criteria discussed in Section 4.2.2. More specifically, we simulate different types of cloud environments by varying several parameters such as: the number of tenant classes, the replication factor, the machine capacity and the replication constraints and observe how each algorithm handles these changes with respect to our evaluation criteria.

Table 4.1 contains a brief description of our setup. We considered two kinds of tenant classes, and following previous approaches [46, 68], use TPC-C as the workload for each class. The tenants that belong to the first tenant class have a performance requirement of 10tps, and the tenants that belong to the second class have a performance requirement of 100tps. For the variables in the multi-linear characterization function, we use the suffix 1 to denote that the variable refers to the 10tps tenant class and the suffix 2 to refer to variables of the 100tps tenant class.

As shown in Table 4.1, we experimented we three types of machines, namely *high capacity machine (HC)*, *medium capacity machine (MC)* and *low capacity machine (LC)*. At each level, the capacity of the machine differs by a factor of 10 from the capacity of the machine at the lower level. For example, the *high capacity machine (HC)* can hold up to 6000 primary replicas and up to 8000 secondary replicas of the 10tps tenant class, whereas the *medium capacity machine (MC)* can hold up to 600 primary replicas and up to 800 secondary replicas of the same tenant class. As shown in Table 4.1, when experimenting with tenants of two tenant classes, we assume that a tenant that belongs to the 100tps tenant class consumes 10X more resources than a tenant of the 10tps tenant class. As a result, a *medium capacity* machine can hold up to 600 primary replicas of a 10tps tenant, but only up to 60 primary replicas of a 100tps tenant.

To evaluate the algorithms based on criteria C2, C3 we use the following approach:

1. We summarize the distribution of the primary replicas for each tenant class across all the machines, using a box plot (Criterion C2).
2. We summarize the distribution of the secondary replicas for each tenant class across all the machines, using a box plot (Criterion C2).
3. For each machine that hosts some primary replicas of a tenant class, we find the distribution of the corresponding secondary replicas across the machines that can host them based on the replic ation constraints. We take the union of these distributions and summarize the resulting distribution it using a box plot (Criterion C3).

The bottom and the top of the box in a box plot, represents the 25^{th} and the 75^{th} percentile, respectively. The ends of the whiskers represent the lowest datum still within 1.5 IQR (interquartile range) of the first quartile, and the highest datum still within 1.5 IQR of the third quartile. Any data not included between the whiskers is plotted as an outlier with a circle.

In all the experiments where two tenant classes are considered, the order in which tenants arrive is random, and in proportion to the tenant class ratios.

In the first part of the experiments we assume that the actual workload (W_a) is the same as the expected workload (W_e) and evaluate all our algorithms using criteria C1, C2 and C3. In the last part, we evaluate how the proposed algorithm performs when $W_a \neq W_e$ using all the evaluation criteria (including criterion C4).

4.5.1 Varying the number of tenant classes

In this experiment, we compare the algorithms presented in Section 4.4 using the *MC* machine type. Experiments with the other two machine types (HC, LC) had similar results. We assume that the replication factor

# Tenant Classes	Machine Type	Machine Characterizing Function
1	LC	$4 * p_1 + 3 * s_1 - 240 = 0$
	MC	$4 * p_1 + 3 * s_1 - 2400 = 0$
	HC	$4 * p_1 + 3 * s_1 - 24000 = 0$
2	LC	$40 * p_2 + 30 * s_2$ $+4 * p_1 + 3 * s_1 - 240 = 0$
	MC	$40 * p_2 + 30 * s_2$ $+4 * p_1 + 3 * s_1 - 2400 = 0$
	HC	$40 * p_2 + 30 * s_2$ $+4 * p_1 + 3 * s_1 - 24000 = 0$

Table 4.1: Experimental Setup.

n is 3 and that according to the replication constraints, the replicas of the same tenant must be placed on different machines. For the *Rk* and the *RkC* family of algorithms, we consider the cases where $k = 1$ and $k = 2$. In the following experiments, we will present results with higher values of the parameter k.

In the first part of the experiment, we compare all the algorithms in an environment where all the tenants belong to the 10tps tenant class. The second part of the experiment includes tenants from both the 10tps and the 100tps tenant class. In both experiments, we use $N = 100000$ tenants. When two tenant classes are considered, 90% of the incoming tenants belong to the 10tps tenant class and the remaining 10% of the tenants belong to the 100tps tenant class, that is $r_1 = 90\%$ and $r_2 = 10\%$. We also performed experiments with different tenant ratios and the results are similar to the ones presented here.

Table 4.2 presents the number of machines that are used by each algorithm to place all the replicas in a way that meets the replication constraints and the performance requirements of the tenants. The last row, M^*, denotes the lower bound on the number of machines used, given by Equation 4.6. Table 4.2, shows that the R1 algorithm needs more machines than the remaining algorithms. As presented in [31, 72], having two ran-

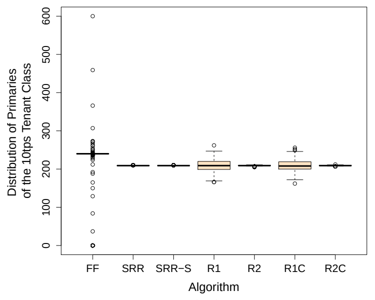
Algorithm	**1 Tenant Class**	**2 Tenant Classes**
FF	418	793
$SRR - S$	417	834
SRR	417	834
R1	478	1200
R2	420	834
R1C	417	794
R2C	417	795
M*	**417**	**792**

Table 4.2: Number of Machines Used.

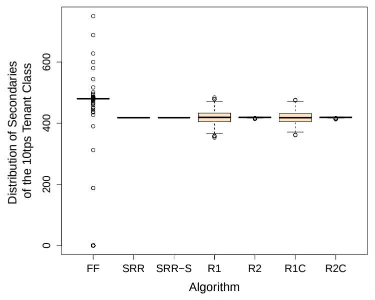
dom choices (R2) can lead to a large reduction in the maximum load of a machine over having only one random choice (i.e., R1). This increased maximum load results in some machines not being able to meet the performance SLOs. By adding more machines, we can reach the point where the maximum load produced by random placement is a point on or below the multi-linear characterization function. Note, that this is not the case for the R1C algorithm, which uses fewer machines than R1. The reason for this behavior is that the R1C algorithm never places a replica on a machine that is overloaded. Instead, the additional check that it performs forces every replica to be placed on machine which has the capacity to host it without creating any performance bottlenecks.

In the experiment presented below, to allow us to compare the algorithms with each other, we used the maximum number of machines used by any algorithm, that is 478 machines for the experiments with one tenant class, and 1200 machines for the experiments with two tenant classes.

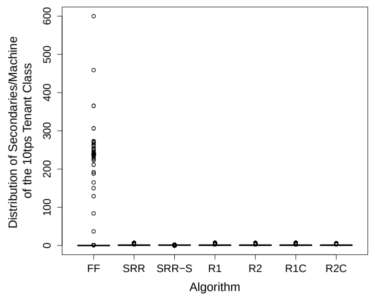
Figure 4.2 presents the replica distributions of the 10tps tenant class when 478 machines are used. The first part of the figure, i.e., Figure 4.2(a), presents the distribution of the primary replicas across the 478 machines. While, Figure 4.2(b) presents the distribution of the secondary replicas across the 478 machines. Since there are 100000 tenants, in a fully balanced state we would expect to have 209 or 210 primary replicas per machine, and 418 or 419 secondary replicas per machine. Both the *SRR* and the



(a)



(b)

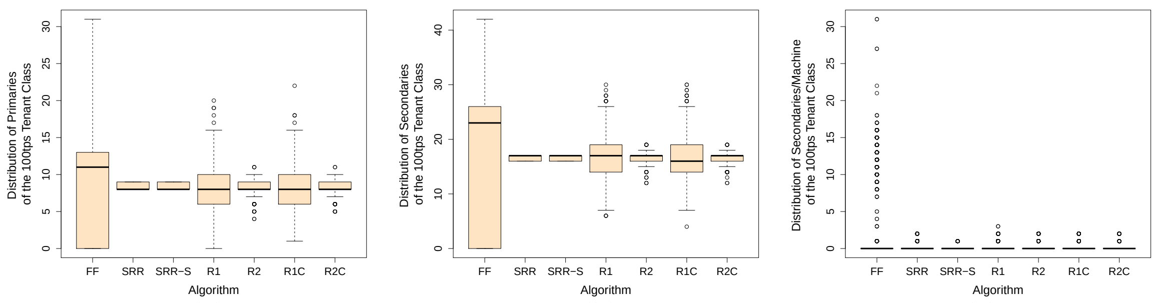


(c)

Figure 4.2: Comparison of all the algorithms on the 10tps tenant class on 478 machines.

SRR-S algorithms produce these distributions. The distributions produced by the R2 and the R2C algorithms are similar, and so are the distributions produced by the R1 and the R1C algorithms.

Figure 4.2(c), is related to Criterion C3. If the load is fully balanced, then each machine would contain 209 or 210 primary replicas, which would have 418 or 420 corresponding secondary replicas. Since, the number of machines is 478, these secondary replicas would be placed across 477 machines. As a result, if the load is fully balanced during failures, each machine would host 0 or 1 secondary replicas whose corresponding primary replica is placed on the same machine. The *SRR-S* algorithm has the best behavior for this criteria (its median is 1). All the other algorithms, except for FF, come close to *SRR-S* on this criteria. In the first few iterations of FF, a tenant's primary replica is placed on the first machine, its first secondary replica is then placed on the second machine and the next secondary replica is placed on the third machine, following the replication constraints. This pattern continues till the first machine is fully filled with primary replicas. All the corresponding secondary replicas are placed on the two neighboring machines, even if there are more machines available. A similar pattern also occurs in the later stages of the algorithm. This is because the FF algorithm is an algorithm that aims to use as few machines as possible, and it does not consider load balancing at all. However, our results suggest that there exist other simple algorithms that can use as many machines as FF uses (see Table 4.2) and at the same time are able to balance the load across all machines. For this reason, we omit FF from the following experiments.



(a) (b) (c)

Figure 4.3: Comparison of all the algorithms on the 100tps tenant class on 1200 machines.

Figure 4.3 presents the load distributions for the 100tps tenant class when 1200 machines are used. The results for the 10tps tenant class are similar and are omitted. As shown in the figure, the R2C and the R2 algorithms produce more balanced distributions compared to the R1C and R1 algorithms and at the same time use fewer machines (see Table 4.2). Our experiments showed that these results hold with different machine capacities, different replication factors, and different tenant ratios. For this reason, for the rest of this section, we will not consider the R1 and the R1C algorithms. Another interesting point, is that the R2 algorithm typically uses more machines than the R2C algorithm. Experiments with one tenant class have shown that the R2 algorithm uses up to 7% more machines than the R2C algorithm. When two tenant classes are involved, we encountered a case where the R2 algorithm used 80% more machines than the R2C algorithm (when the replication factor $n = 3$, and LC machines are used). In general, the *RkC* family of algorithms, typically needs fewer machines than the *Rk* family of algorithms, due to the additional check that it performs. For this reason, we omit *Rk* in the following experiments. In this experiment, *SRR-S* is the algorithm that produces almost uniform replica distributions (Criteria C2 and C3). Compared to RkC which considers only 1 or 2 machines when placing the replicas, the *SRR-S* algorithm takes into account the load on all the machines when it applies its first and second heuristic. In the following experiment, we show how the *RkC* algorithm improves as the value of k increases beyond 2.

4.5.2 Varying the machine capacity

In this section, we present experiments on machines with different capacities. More specifically, we use two tenant classes (10tps, 100tps) with 90% and 10% as the corresponding tenant ratios, a replication factor $n = 3$, and three types of machines, namely HC, MC and LC. The algorithms we consider in this experiment are *SRR*, *SRR-S* and *RkC* where $k = 2, 4, 8, 16, 32$. We run each experiment using the maximum number of machines needed by any algorithm, which ends up being 80 HC machines, 834 MC machines and 10000 LC machines.

Regarding the number of machines that are used by each algorithm, when HC machines are considered all the algorithms need 80 machines to place the replicas ($M^* = 80$). When the MC machines are used, the *SRR* and the *SRR-S* algorithms need 834 machines, whereas the remaining algorithms only need 795 machines ($M^* = 792$). Finally, the *SRR* and the *SRR-S* algorithms use 10000 LC machines, whereas the *RkC* algorithms use about 8200 machines ($M^* = 7918$). This difference between *RkC*, *SRR* and *SRR-S* is due to the round-off errors discussed in Section 4.4. More specifically, when the LC machines are used, the value of z is 1.2631. As a result $\delta = 26.31\%$. This means that the number of machines used by *SRR* and *SRR-S* is about 26.31% more than M^*, which is actually the case that we observe empirically. However, note that the *RkC* algorithm which is less rigid can actually use 18% fewer machines than the *SRR* and the

SRR-S algorithms.

Figure 4.4 presents the distributions produced by each algorithm for the tenants of the 100tps tenant class on the HC machine. The distribution of the 10tps class tenants on HC machines as well as the distributions for the 10tps and 100tps tenant classes on MC machines are similar and are ommitted here. The results for LC machines, are similar to those on the MC machines, and are omitted here.

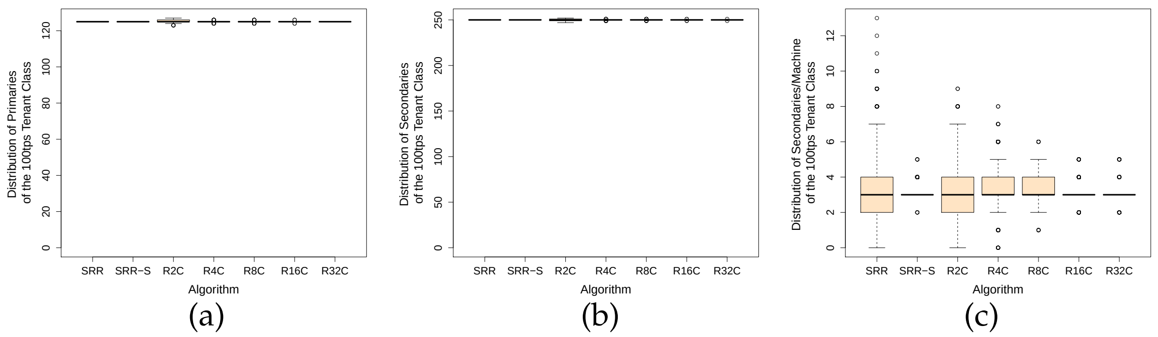


Figure 4.4: Comparison of replica placement algorithms on the 100tps tenant class on 80 HC machines.

As shown in Figure 4.4 (a) and (b), the primary replica and the secondary replica distributions are very similar across all the algorithms, especially for values of k greater than 2 (Criterion C2). Figure 4.4 (c) shows that the heuristic that *SRR-S* uses has a clear advantage over the *SRR* algorithm with respect to criterion C3. The *RkC* algorithms produce a distribution similar to the one produced by *SRR-S* for values of k greater than 8. This is expected, since as the value of k increases, the probability that more machines will be compared when the second heuristic is applied is higher. For example, when $k = 2$, R2C will pick 2 machines that are valid candidates and it will compare the number of secondary replicas that they host. R2C will evaluate the second heuristic, which is related to criterion C3, only if these two machines have the same load. If they have different loads, then the heuristic will not be applied. As k increases, the chance that some machines will have the same load is higher and as a result the second heuristic will be used. This is the reason why larger values of k produce distributions close to the one produced by *SRR-S*. Note, however, that not all the machines need to be examined in order to get good results. In the following experiments, we will not examine *SRR*, but will use *SRR-S* as a reference.

4.5.3 Varying the replication factor

In this experiment we vary the replication factor and keep the number of tenants, tenant classes, tenant ratios and machine type constant. More specifically, we use the HC machine (other machine types have similar results), 100000 tenants and the two tenant classes used before (10tps, 100tps) with a $90\% - 10\%$ tenant ratio. Since we have already examined the HC machine case when $n = 3$, we experimented with two additional replication factor values, namely $n = 5, 7$.

As shown in the previous experiment, all the algorithms have very good results with respect to criterion C2. This is also the case in this experiment, at all the replication factors examined.

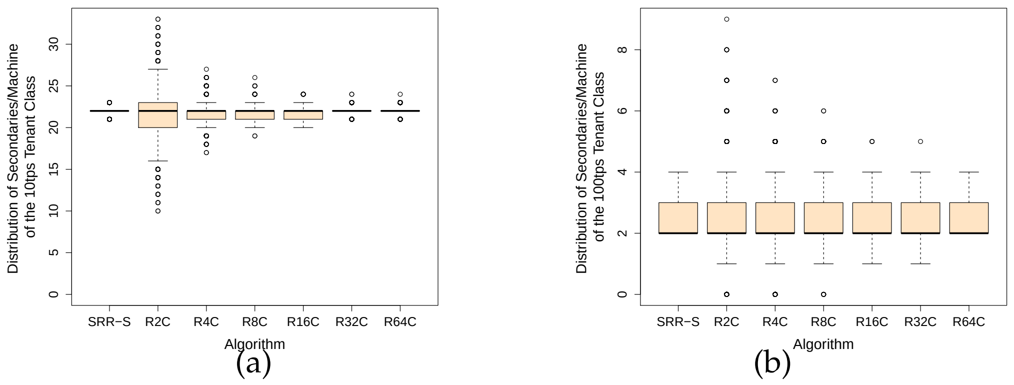
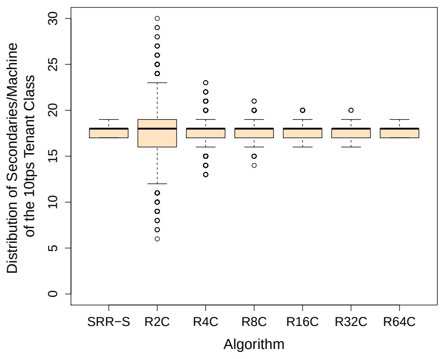
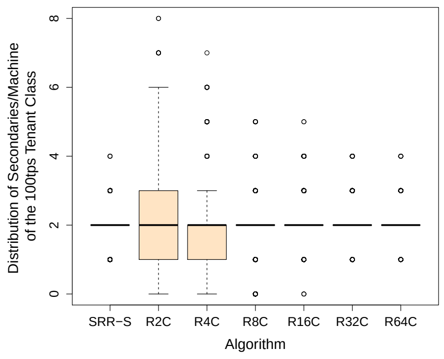


Figure 4.5: Comparison of replica placement algorithms when the replication factor is 5.



(a)



(b)

Figure 4.6: Comparison of replica placement algorithms when the replication factor is 7.

Regarding the number of machines used, all the algorithms needed 129 machines when $n = 5$ ($M^* = 127$) and 176 machines when $n = 7$ ($M^* = 175$). Figure 4.5 presents the distribution of secondary replicas per machine (Criterion C3) for the 10tps and 100tps tenant class when the replication factor is 5. Figure 4.6 presents the distributions of secondary replicas per machine (Criterion C3) for the 10tps and 100tps tenant classes when the replication factor is 7 and HC machines are used. As shown in the figure, RkC still produces good results with respect to criterion C3, for values of k greater than 8. Another interesting point is that, *SRR* and *SRR-S* cannot be used when the replication factor is 5 and LC machines are used. The reason is that the value of z in this case is 0.78, which is less than 1. On the other hand, the *RkC* algorithms were able to place all the replicas using 1.4% more machines than the lower bound M^*.

4.5.4 Introducing rack constraints

In all the previous experiments, we assumed that the replication constraints require that all the replicas of the same tenant must be placed on different machines. In this experiment, we introduce rack constraints and study how the *SRR-S* and the *RkC* algorithms perform when this kind of constraints are involved. Our environment consists of two tenant classes and 100000 tenants. The replication factor n is equal to 3 and each rack consists of 40 MC machines (experiments with different types of machines produced similar results). We used 21 racks for our experiment, that is 840 machines. Our experiment shows that the results seen in the previous experiments still hold when rack constraints are involved.

We also performed experiments with more coarse-grained constraints (e.g. switch constraints). As an example, we assumed that the replication constraints require that each tenant's replicas must be placed in different failure zones, where each failure zone consists of 7 racks connected to one switch. Using again 21 racks (3 failure zones), we were able to produce similar results.

An interesting point is that the *SRR-S* algorithm is not always able to use the same number of machines that RkC uses when rack constraints are involved. For example, in one of our experiments, we used 200 machines, divided into 5 racks and a replication factor equal to 3. In this case, the number of racks is not a multiple of the replication factor. $SRR - S$ actually needed 6 racks to place all the replicas, whereas RkC used 5 racks only. As part of future work, we plan on deeper investigating the relationship between replica placement algorithms, arbitrary replication constraints and number of available failure zones.

4.5.5 Handling Workload Changes

The goal of this experiment is to quantify the advantage that the RkC algorithm(s) has over the static *SRR-S* algorithm when the actual workload (W_e) changes from the expected workload (W_a) using criterion C4. (Recall, that as mentioned in Section 4.4, the *SRR-S* algorithm is a reference point, and it is expected to do worse than the RkC algorithm when the workload changes, so the goal here is to quantify the difference).

We performed experiments where $N_a = N_e$, but the actual tenant ratios differ from the expected tenant ratios, including extreme cases where the expected workload contains tenants of one class but all the arriving tenants belong to a different class. Our results show that RkC can gracefully adapt to these changes. We omit these experiments and we present an experiment on a more complicated setting where the actual tenant ratios are initially the same as the expected tenant ratios but after a while they change.

More specifically, we investigate how the replica placement algorithms perform when the **tenant ratios** of new incoming tenants change from the initial provisioning phase. We assume that the DaaS provider expects to support approximately $N_e = 100000$ tenants that belong to two classes with 10tps and 100tps SLOs, with 90% and 10% corresponding expected tenant ratios, and that the replication factor is 3, i.e., $n = 3$. Based on these expectations the service provider provisions a certain number of machines. We used both HC and MC machines for this experiment. Both the *SRR-S* and the *RkC* algorithms need 80 HC machines. When the MC machines are used, the *SRR-S* algorithm needs 834 machines and the RkC algorithm needs 795 machines to place 100000 tenants. Thus, as in the previous experiments, we used 80 HC machines and 834 MC machines to cover both algorithms. We assume that the DaaS provider adds 20% of extra machines ($f = 1.2$) to account for machine failures and additional tenants, thus using 96 HC machines or 1001 MC machines.

In our setup, 100000 tenants arrive with a $90\% - 10\%$ corresponding tenant ratios (for the 10tps and 100tps SLOs respectively), and then the tenant ratio changes to $10\% - 90\%$. Our goal is to examine how many tenants each algorithm is able to serve (Criterion C4), given this unexpected change in the tenant ratios. If two different clusters of HC machines were used (for a total of 96 machines), each one handling tenants with different tenant ratios, then, both the *SRR-S* and the *RkC* algorithms would use a cluster of 80 HC machines to place the 100000 tenants with a $90\% - 10\%$

tenant ratio. In the second cluster that consists of 16 HC machines, the *SRR-S* algorithm can place 4160 tenants with a 10% − 90% tenant ratio, and the *RkC* algorithm can place 4200 tenants with a 10% − 90% ratio. When using one cluster of 96 machines, the *SRR-S* algorithm can actually place 2270 tenants of the 10% − 90% ratio, which is about 55% of the maximum. On the other hand *RkC* is able to place all the tenants (4200 tenants).

We repeated the experiment, using 1001 MC machines. In the case of two separate clusters, the *SRR-S* algorithm needs 834 machines to place 100000 tenants with the 90% − 10% tenant ratio. In the second cluster that consists of 167 machines, the *SRR-S* algorithm can place 3340 tenants with the 10% − 90% tenant ratio. The *RkC* algorithm would need a cluster of 795 machines to place 100000 tenants. The second cluster that consists of 206 machines, can accommodate about 5400 tenants with the 10% − 90% tenant ratio. When using one cluster of 1001 machines, the *SRR-S* algorithm is able to serve 2200 tenants out of the 3340 (about 65%) whereas *RkC* placed 5150 out of 5400 tenants (about 95%). The main reason for this behavior is that *SRR-S*'s static policy works perfectly if all the expectations are met (number of tenants, tenant ratios). On the other hand, the additional check that *RkC* performs, makes it more flexible and amenable to workload changes.

Figure 4.7 presents the distribution produced by all the algorithms for the 100tps tenant class, when the HC machines are used and the total number of tenants is 102270. This is the maximum number of tenants *SRR-S* can handle. The results for the 10tps tenant class and the results on the MC machines are similar. As shown in the figure, all the algorithms balance the load of both the primary and the secondary replicas. Regarding load balancing during failures, as in the previous experiments, *RkC* has similar results to *SRR-S* for values of k greater than 8.

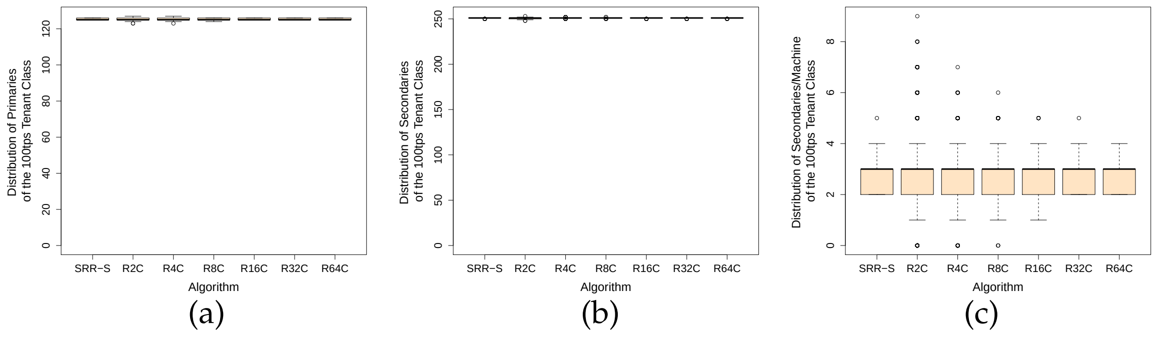


Figure 4.7: Comparison of replica placement algorithms on the 100tps tenant class with varying tenant ratios.

4.5.6 Discussion

The experiments presented in this section show that *RkC* is a replica placement algorithm with many appealing aspects. More specifically, RkC has the following properties:

- It has low cost since the number of machines it uses is very close to the lower bound M^* (Criterion C1).
- It is able to spread evenly the primary and the secondary replicas across all the machines (Criterion C2).
- It is able to place the secondary replicas in such a way that the load could be balanced in case of machine failures (Criterion C3).
- It can accommodate rack constraints.
- It is flexible enough to accommodate changes in the tenant ratios due to its dynamic behavior that results from making use of the machine characterizing function during the replica placement process.
- The above properties hold for a variety of replication factors, tenant ratios and machine capacities.

The value of k chosen depends on how the system is designed to perform the replica placement operation. If all the metadata about the load (number of replicas) on each machine are stored on one central machine that is responsible for assigning replicas to machines, then the value of k can be as high as the cluster size. In this case, the complexity of the RkC algorithm is $O(n * M)$. Practically, this means that for very large clusters (thousands of nodes), one may want to use smaller values of k. However, we generally do not expect this to be a problem when the replica placement process is done by only one machine. On the other hand, if the system is designed in such a way that in order to get information about the current load of a machine, the machine has to be contacted, a large value of k could significantly slow down the placement process. In this case, according to our experiments making k equal to 16 or 32, will produce very good results with respect to our evaluation criteria. The algorithm could be modified so that the system polls a larger (than k) number of machines at run-time, and only waits for the first k valid responses.

4.6 Related Work

As shown in [59], provisioning and maintaining a cloud infrastructure is a costly investment. For this reason, DaaS environments aim to maximize server utilization by consolidating multiple tenants on each machine [39, 81]. The first step in providing performance SLOs in these environments is to model the performance of the system under a real workload [25] using benchmarking [68, 43, 86], empirical models [45] or machine learning [55, 83] in order to predict performance [53, 34].

A typical approach used in cloud environments to balance the load is to either swap the roles of primary and secondary replicas [73] or to migrate replicas to different machines [54, 47]. These techniques deal with the problem of balancing the load in case of workload spikes. Our work, focuses on the *initial replica placement problem* and is complimentary to these studies.

The work in [84] studies the problem of replica placement in multi-tenant main-memory clusters and proposes algorithms for the static and incremental replica placement problem. Our work, does not address the incremental placement problem which accounts for variations in the load of existing tenants. We focus on the *online* replica placement problem in which the actual workloads are not known in advance. The work in [84] studies the offline version of the problem since the heuristic algorithms that proposes for the static placement problem, sort the set of incoming tenants based on their load before placing them on the machines. In the environment studied, the load of each tenant can be equally shared across multiple replicas. Thus, in case of single-node failures the additional load on the remaining replicas can be quantified in advance. In our setting, the role of each replica is different and the load of a tenant cannot be shared equally across all the replicas. As a result, we assume that the failure-handling algorithms that will assign new roles to the remaining replicas following a node failure, will be able to explore our load balancing techniques in order to produce uniform load distributions. Finally the heuristics proposed in [84] for the static placement problem, sort all the machines based on their current load before making an assignment. Our proposed algorithm does not require global knowledge of the load.

The work in [68] proposes a framework for using performance characterizing functions to determine cost-effective hardware provisioning policies, but that work does consider multiple database replicas, replication constraints or load balancing. Similarly, the work in [45] provides non-linear programs for tenant placement, but the techniques proposed assume knowledge of the entire workload (offline techniques).

Finally, our problem is related to the bin packing problem [65, 57, 40]. Our proposed techniques are related to the worst fit heuristic used for bin packing when optimizing for load balancing during the replica placement process (criterion C2). In our work, we also optimize for load balancing in case of single-node failures (criterion C3), which the bin packing heuristics cannot address directly.

4.7 Summary

To the best of our knowledge, this thesis presents the first study that examines the interactions between *online* database placement, replication constraints, and load balancing in multi-tenant environments, where the tenants have different performance SLOs. We present a framework to design and evaluate replica placement algorithms using four criteria. These criteria consider the cost associated with each replica placement algorithm, calculated as the number of machines needed to place all the replicas, the way each algorithm balances the load during normal operation and in case of a node failure and the adaptivity of the placement algorithm when the workload characteristics change. Our framework requires as input a characterizing function that determines which replica placement assignments do not violate the performance SLOs. Based on this framework, we designed a number of algorithms and found that the RkC family of algorithms are particularly appealing as they can balance the load evenly across machines, are simple to implement, are adaptive, and at the same time require almost as few machines as theoretically possible.

Chapter 5

Towards Benchmark-as-a-Service

5.1 Introduction

One of the greatest hurdles associated with deploying traditional on-site relational database management systems (RDBMSs) is the overall complexity of choosing, configuring, and maintaining the RDBMS as well as the server it operates on. In choosing and configuring a particular RDBMS and server to deploy, the users must have a firm understanding of the characteristics of their particular workload. Some of the important characteristics include the size of the database, the nature of the queries (transactional or ad-hoc/analytic), and the desired metric of performance (latency or throughput). Along with the upfront decisions of a particular RDBMS and corresponding server, the user must consider the long-term licensing, maintenance, and administration costs of running the system. This complexity that is associated with managing onsite DBMSs is a key reason why cloud-based Database-as-a-Service (DaaS) is starting to gain in popularity as an alternative to on-site RDBMS systems, especially for small and mid-sized database users.

The widely perceived advantage of the DaaS paradigm is that the user has now transferred the complex and nuanced decisions, and the heavy costs of operating an on-site RDBMS to the DaaS provider. Specifically, by turning to a DaaS, the user stores the data in the DaaS, and uses the DaaS APIs to query their data, for a monthly subscriber fee. This monthly fee incorporates all the responsibilities (such as data availability) that the provider has taken on. This fee also includes an “on-demand” payment model for computing resources that are consumed (this later component includes the costs that are associated with the CPU cycles and the storage that is consumed). However, the DaaS providers recognize that the needs of the database users varies significantly, and that one fixed pricing model will alienate one or more segments of the customer market. Consequently, in order to appeal to the entire spectrum of potential users, the DaaS providers have begun to diversify their offerings with multiple pricing options, each promising different levels of computing power, storage capability, and measures of performance. However, from the users’ perspective, there is now a bewildering set of choices. As with the process of choosing an on-site RDBMS, they must now fully understand the characteristics, such the raw DBMS performance and query workload characteristics, when choosing an appropriate DaaS product. In fact, with the addition of the pay-as-you-go model for the computing resources, they now have an additional factor to consider – namely, the impact of the computing resources usage on their bottom line.

Initially, it may seem that the DaaS products alleviate many of the pains that are associated with running an on-site RDBMS. However, as we show in this study, the truth is that the users are actually in a tough position – they must now make an upfront decision of choosing a DaaS offering, while the long-term performance and cost consequences of their decisions are harder to figure out.

A crucial point that we make in this work is that currently the DaaS users do not have an effective method to compare the suitability of one DaaS option over another, and fully understand the actual “cost" of their service. In a traditional RDBMS setting, the database users know that they can always turn to well-established benchmarks (such as the TPC benchmarks), to estimate whether one solution is more suitable than another. However, while such benchmarks identify price and performance as key metrics, these metrics have not been defined for the complex variable pricing models of DaaS products. For instance, they do not consider storage costs of the database or the utilization hours as factors of the price/performance. Moreover, TPC benchmarks usually take into consideration the total cost of ownership as a primary metric. This is incompatible with the "pay-as-you-go" model of cloud computing since the cloud customers are not directly exposed to the hardware, software maintenance, and administration costs of the deployment.

To highlight the practical need for an easy to use and accurate pricing model, consider the popular Amazon Relational Database Service (RDS) [28]. While Amazon initially provided users with a database service backed by MySQL [74], recently they have unveiled an option to swap the back-end to an Oracle RDBMS instance [76]. Of course, these two options are not price equivalent, and currently the "Quadruple Extra Large DB" instance cost of the MySQL option is $2.60 per compute-hour, while the Oracle option is 31% more expensive at $3.40 per compute-hour. This price difference is largely due to the licensing cost ($0.80 per compute-hour) of the commercial Oracle system over the open-source MySQL system. While a cursory glance at these numbers would suggest that the cost-conscious user should buy the MySQL option, this choice ignores the fact that the often superior performance of a commercial DBMS may actually result in less computation time than the "free" MySQL option, and thus may actually be the cheaper option in some cases!

To better illustrate this point, consider running the following Wisconsin benchmark [50] query (Query 21):

```
INSERT INTO TMP
SELECT MIN (unique3) FROM TABLE1
GROUP BY onePercent
```

When we run this query on MySQL and a commercial DBMS (SQL Server) on the same physical machine (configuration details are described in Section 5.3), SQL Server runs this query in 185 seconds while MySQL takes 621 seconds to execute this query. How is the user's cost affected by this 3.3X performance gap when the user decides to run this workload on a DaaS?

Assuming a simple pricing model where the user pays a fixed cost of $1.30 per compute-hour for the specific DB Instance Class used, and a monthly storage fee of $25 for a database of 250GB, Figure 5.1 shows the cumulative monthly cost for the full deployment when these two RDBMSs are used, and when the workload consists of repetitions of the above query. For the SQL Server-based service, the user has to pay an extra hourly license fee/cost. Figure 5.1 examines four possible pricing models for the hourly license costs (`lc`) ranging from $0.65 to $3.90.

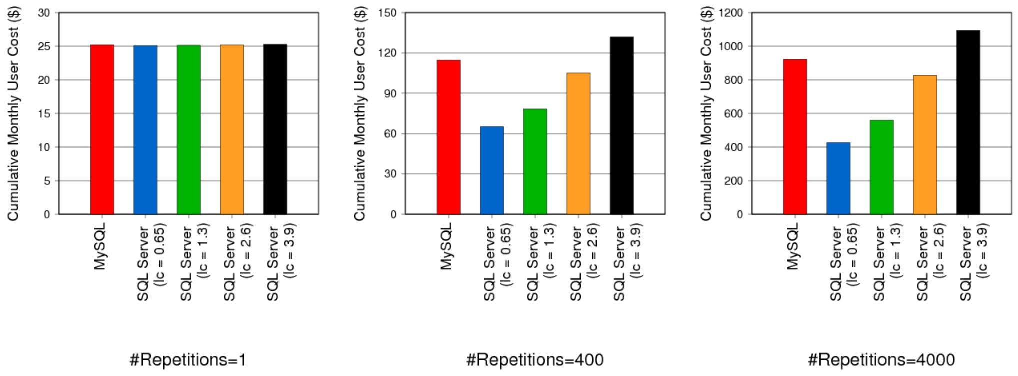


Figure 5.1: Cumulative monthly user cost as a function of workload repetitions, DBMS type, and pricing model.

Interestingly, Figure 5.1 shows that although the user does not need to pay any license fee for MySQL, using this DBMS results in a higher total cost when the license fee for SQL Server is less than $3.90, and the user frequently issues such query requests. If the user load is high (e.g., around 4000 queries/month), then choosing the commercial-based RDBMS can save the user up to 54%. The reason for this behavior is that the higher efficiency of SQL Server results in decreased resource usage, and overall reduces the end-to-end cost for the user.

5.1.1 Towards BaaS

As the example above illustrates, the actual cost that a DaaS user will incur is hard to guess upfront. A simple approach to solve this pain point is to take the existing pricing model a step further. So in this scenario, the user provides the database and workload characteristics to the DaaS provider and in return the DaaS provider gives a price quote for running that workload. Along with the price quote the DaaS provider can attach a Service Level Agreement (SLA) that make guarantees on aspects such as performance variability and availability, and also lays out the penalty associated with SLA violations. (Alternatively, some parameters in the SLA could be provided upfront by the user, or the DaaS provider may come back with multiple price quotes at different SLA levels.)

Thus, what we need is for the DaaS providers to run another service – namely, "*Benchmark as a Service*" (BaaS) that makes it transparent to the user what it would cost to run their workload. Such a BaaS service could be free, and then would be crudely analogous to the utility model that is present in other parts of our lives – for example, internet service providers give an accurate price and specify the upper limit of the bandwidth. We acknowledge that a DaaS provider may have a more complicated problem at hand since the SLAs in a DaaS setting could be complex (hence, this is a promising direction for future work). But, from the perspective of the end-consumer of DaaS, a transparent pricing model could be very appealing, and perhaps a competitive advantage for the DaaS providers that choose to simplify their DaaS offering by coupling it with a BaaS service. As a side note, we would like to point out that the term “Benchmark” in BaaS represents the user process of benchmarking a given workload under all possible hardware/software configurations, with the goal of figuring out the most cost-effective combination. BaaS removes the hassle of benchmarking from the user and at the same time prices the user’s workload.

The BaaS approach also has a number of potential advantages for the DaaS provider as it provides a strong motivation to find the most optimal way of running the backend DBMS engine (rather than punting this decision to the end user), thereby reducing their operational cost (and perhaps improving their bottom line). Furthermore, the BaaS approach may provide more flexibility in managing the DaaS infrastructure – for example, a DaaS provider may not need to offer a range of DBMSs or data processing backends, and could simplify their infrastructure management by using only a single data processing engine. Finally, with a BaaS approach, the overall DaaS system potentially operates at a much higher operating efficiency (generally the queries across the system are likely to run far more efficient ly then when the end user has to make nuanced decisions about configuring their DBMS and making bad choices), which in most cases is also likely to produce a more energy-efficient way of operating the DaaS, since in many cases the goal of energy efficiency lines ups with the goal of optimizing for traditional performance goals.

The remainder of this chapter is organized as follows: Section 5.2 presents our cost model. Experimental results are presented in Section 5.3, while Section 5.4 discusses related work. Section 5.5 contains our concluding remarks and points to some directions for future work.

5.2 Cost Model

This section presents a simple cost model for using relational DBMSs in the cloud. The model considers the cost that is incurred by the end user in using a DaaS offering. We then use this model in our experiments (see Section 5.3) to explore the costs associated with using a DaaS product.

We patterned a simple pricing model crudely using Amazon's DaaS product as a reference. According to the Amazon's DaaS pricing model [28], the users pay only for the resources that they consume. Several parameters determine this cost. The first one, is an hourly fee that corresponds to the specific DB Instance Class chosen by the customer. The DB Instance is a database environment in the cloud with the compute and storage resources that the customer specifies. For example, currently in Amazon's RDS, 6 DB Instance Classes are provided. The "Small" DB Instance Class has 1.7GB of main memory and one 1.0-1.2 GHz CPU core (1 ECU), whereas the "Extra Large DB Instance" has 15GB of main memory and four 2.0-2.4 GHz CPU cores (8 ECUs). Generally, the hourly rates vary with the DB Instance Classes, since each class has different hardware characteristics. An extra hourly license cost/fee, is added for DB Instances backed by a commercial DBMS, which also varies according to the DB Instance Class chosen. The last parameter is a monthly storage fee per GB of the provisioned storage needed by the workload.

Consider a fixed database instance type chosen by the user with corresponding hourly cost `dbc`, an hourly license fee for the DBMS equal to `lc`, a monthly fee for the provisioned storage per GB equal to `stc`, and `H` hours of utilization per month of the DB instance. Given that the DB instance has associated capacity of `DS` GB, the monthly user cost (MUC) can be determined as:

$$MUC = H * (dbc + lc) + DS * stc \tag{5.1}$$

To keep our model simple, we do not consider the monthly network related costs.We also do not consider the costs for the extra backup storage that may be needed. These rates affect the total cost in a way similar to the storage fee `stc` and can easily be added to the above equation. Moreover, our model assumes that only one database instance is used by the customer. Creating and validating a more complex model that considers a combination of different database instance classes and multiple database instances per class is part of future work.

5.3 Experimental Evaluation

In this section, we discuss our experimental results which include performance measurements of a database server running different workloads and using different storage organizations, using MySQL and SQL Server. Based on these performance results and the pricing model presented in Section 5.2, we compare the total cost that the user has to pay when using these two DBMSs in a DaaS.

The work by Schad et al. [82] presents experimental results showing that performance unpredictability is a major issue when running workloads in the cloud. The variance observed can be attributed to several factors, including different types of virtual systems provided by the service, different availability zones (distinct locations that are insulated from failures in other availability zones), and time of the day/week when the workload was run. Similar observations are discussed in the work of Armbrust et al. [30]. All these parameters make it difficult to estimate the impact on the cost and the performance of different database systems serving applications in a cloud-based environment. In this study, in an effort to eliminate these variances, we decided to measure the performance of the different DBMSs on a stand-alone local server machine. We show that even in this isolated environment, where variance due to the factors mentioned above is eliminated, the impact of the workload type and the efficiency of the DBMS on the monthly user's bill is not straightforward to estimate.

5.3.1 Server Configuration

Our test platform is a HP Proliant server with a dual quad-core hyper-threaded Intel Xeon L5630 processors (@ 2.13GHz), 32 GB of memory, and 12 HP 146GB 10K RPM SAS drives.

The server is dual booted with 64-bit Ubuntu Server 9.10 and 64-bit Windows Server 2008 R2 Enterprise Edition. The Linux version is used to run MySQL (MySQL Community Server 5.5.9) and the Windows version to run SQL Server 2008 R2 (Data Center Edition). Each disk is partitioned roughly evenly between the two operating systems. The first hard disk is used for the installation of the operating systems and all the database binaries.

5.3.2 DBMS Configuration

In our experiments, the database buffer pool is set to 24GB for both DBMSs. One disk is used to store the log files and the remaining 10 disks are reserved for the data files and the temporary space that is needed during query execution.

For SQL Server, we created a "file group" of 20 data files across the 20 data disk partitions (the 10 Windows partitions are further subdivided into two partitions). In this way, each of the 16 (hyperthreaded) cores can be assigned to one disk partition to allow parallel query processing. MySQL currently does not support such intra-query parallelism. For this reason, we created one data file striped across the 10 data disks so that we can get a high aggregate disk bandwidth. For MySQL we used the

InnoDB storage engine, which is the default setting and the one used in Amazon's RDS.

5.3.3 The Wisconsin Benchmark

For our experiments we decided to use workloads based on the Wisconsin benchmark [50]. Our decision was driven by the fact that it is a simple "micro" benchmark that is fairly easy to set up and does not have complicated rules about how to run and measure a benchmark. Furthermore, this benchmark contains a variety of queries including selections, joins, projections, aggregations and updates. These simple queries are building blocks for more complex workloads and provide good insights about the potential impact on more complex workload characteristics.

The benchmark uses three basic relations, two that have the same number of tuples $(\mathtt{T})$ and one that contains $\mathtt{T}/10$ tuples. Each relation consists of sixteen attributes, thirteen 4-byte signed integers and three 52-byte varchars. The most widely used attributes in the benchmark are `unique1`, `unique2` and `onePercent`. The values of the `unique1` attribute are uniformly distributed unique random numbers in the range 0 to $\mathtt{T}-1$. The values of `unique2` are in sequential order from 0 to $\mathtt{T}-1$. The original benchmark paper [50] contains more information about each attribute and its values.

The benchmark explores two different kinds of storage organizations. The first one is called `StorageOrg-H` and contains one heap file for each relation. This storage layout doesn't contain any primary key indices. In the second storage organization, called `StorageOrg-I`, each relation has a clustered index on the `unique2` attribute, a unique non-clustered index on the `unique1` attribute and a non-unique non-clustered index on the `onePercent` attribute.

5.3.4 Experimental Setting

For our experiments, we created six different types of workloads based on the Wisconsin benchmark. The first two workloads contain all the queries in the benchmark, and are called `mixedworkload1` and `mixedworkload2`. The first workload, `mixedworkload1`, uses heapfiles as the storage layout (`StorageOrg-H`). The second workload, `mixedworkload2`, uses the clustered and non-clustered indices defined by the benchmark (`StorageOrg-I`). We generated a DSS-like workload using a subset of the Wisconsin benchmark queries. From this set of queries, we created two DSS workloads, `dssworkload1` and `dssworkload2`, corresponding to the two storage layouts (`StorageOrg-H` and `StorageOrg-I` respectively). Similarly, we generated two OLTP workloads consisting of OLTP-like queries. These two workloads are `oltpworkload1` and `oltpworkload2`, and correspond to the storage layouts `StorageOrg-H` and `StorageOrg-I` respectively.

Note that some of the queries of the mixed workloads are not presented in the OLTP or in the DSS workloads. More specifically, the 10% selection queries (Q2, Q4, Q6) as well as the 1% selection to screen query (Q8) are only included in the mixed workloads. We did not include the 10% selections in the other workloads because we wanted to experiment with high-selective queries in the OLTP workloads, and we wanted the DSS workloads to mainly consist of join and aggregation queries. Query Q8 was omitted since most of its execution time with MySQL was spent in printing the output to the screen, and not actually evaluating the query result. In the original Wisconsin benchmark paper [50], some of the queries are executed only on either `Storage-H` or `Storage-I`. In this work, we decided to execute all the queries using both storage layouts. This decision was driven by the fact that for some queries the DBMSs don't pick the execution plan described in the benchmark. For example, Query 6 is supposed to use a non-clustered index, that's why it is tested only in `Storage-I`. However, in our experiments the actual plan picked by the optimizer of both DBMSs is a scan on the table. That's the reason why some of the queries are presented twice in some workloads (e.g., Q6 in `mixedworkload1` and `mixedworkload2`).

We created three data files using a Wisconsin benchmark generator. Each file corresponds to one relation of the benchmark. The two tables of the database contain 400M tuples, and the third one has 40M tuples. The size of the flat files for these tables is 80GB, 80GB, and 8GB respectively. Thus, the total raw database size is approximately 168GB. Between the executions of queries we purge the buffer pool (i.e., all reported numbers are "cold"). We also update the statistics for all the tables that are used in a query before its execution starts. The time to clean the buffer pool and update the statistics is not included in the experiment's total execution time. The temporary (TMP) tables that are used to store the results of each query are dropped after the query is executed and recreated when needed. Each query was run 3 times and the average value is reported. We did not see a lot of variance across runs of the same query. All the time values are reported in seconds. The data loading times were fairly similar across both DBMSs, and are not included in computing the total cost below.

We used the model presented in Section 5.2 to estimate the total cost incurred by the end DaaS subscriber/user. To compute the monthly user cost ($\mathcal{MUC}$), we set the DB instance fee (`dbc`) to \$1.30 per hour. This `dbc` is equal to the rate of a high-memory double extra large DB Instance offered in Amazon RDS, which is the closest Amazon Instance configuration to our server. To get a better sense of how the total cost is affected by the license cost/fee (`lc`), we experimented with the following hourly license rates for the commercial DBMS: $\{\$0.65, \$1.30, \$2.60, \$3.90\}$. Since MySQL is open-source, its licensing fee is \$0. The monthly storage fee `stc` is set to \$0.10 per GB (similar to Amazon's RDS rate). We set the provisioned storage `DS` (data, log files and temporary space) for both DBMSs to 250GB.

To evaluate how the storage fee combined with the hourly fees affects the monthly user cost, we varied the number of repetitions of the workload, so that we can experiment with short and long-running workloads of the same type. We first report the cumulative user cost when the workload is executed only once (#repetitions=1). The next number of repetitions reported (#repetitions=N), corresponds to a total execution time close to a period of one month (computed based on the execution time of the workload on the slowest DBMS). This case represents the scenario were the end user application is driving the provisioned DBMS instance nearly to its peak capacity (for the slowest DBMS). Finally, we also present the comparative monthly costs when $N/10$ repetitions are performed. For example in Figure 5.1, $N = 4,000$, since the slowest DBMS (MySQL) can execute Query 21, approximately $4,000$ times in a period of a month.

5.3.5 Mixed Workloads

The mixed workloads contain all the queries in the Wisconsin benchmark that finished within 3 hours with both DBMSs. Some queries (i.e., MySQL running joins in `mixedworkload1`) were stopped after 14 hours of execution. Although the same queries were completed using SQL Server, we do not take into account these numbers. It is clear that having such queries in the workload will lead to poor performance and higher cost, and hence will favor the usage of the commercial DBMS. However, we believe it's interesting to see what happens with respect to performance and cost when all the queries of the workload are completed in both systems in a reasonable amount of time. Note that all the queries that MySQL could finish within 14 hours were also completed by SQL Server within 14 hours.

Tables 5.1 and 5.2 contain the execution times for both DBMSs using `mixedworkload1` and `mixedworkload2` respectively. The last rows of the tables contain the total execution time for each database system used. Figures 5.2 and 5.3 show the estimated monthly cost for the customer

Table 5.1: Mixed Workload 1

Query	**Query Description**	**SQL Server Time (secs)**	**MySQL Time (secs)**
Q1	1% selection on `unique2`	224	665
Q2	10% selection on `unique2`	482	1185
Q5	1% selection on `unique1`	195	739
Q6	10% selection on `unique1`	332	1191
Q7	Single tuple selection to screen	191	555
Q8	1% selection to screen	236	1721
Q18	1% projection	129	1523
Q20	Min. aggregate	190	482
Q21	Min. aggregate with group by	185	621
Q22	Sum aggregate with group by	187	747
Q26	Insert 1 tuple	0.20	0.23
Q27	Delete 1 tuple	192	637
Q28	Update on `unique2`	192	595
Q32	Update on `unique1`	197	609
Total		**2932**	**11270**

when MySQL or SQL Server is used.

As shown in Table 5.1, when the database consists only of heapfiles (`mixedworkload1`), MySQL is approximately 3.84X (2.3 hours) slower than SQL Server. Notice that Table 5.1 does not show the original Wisconsin benchmark Queries 9-17 – these are join queries that did not complete

Table 5.2: Mixed Workload 2

Query	**Query Description**	**SQL Server Time (secs)**	**MySQL Time (secs)**
Q3	1% selection on `unique2`	22	51
Q4	10% selection on `unique2`	203	551
Q6	10% selection on `unique1`	883	1146
Q7	Single tuple selection to screen	0.75	0.60
Q8	1% selection to screen	62	1245
Q12	JoinAselB	412	1071
Q13	JoinABPrime	408	1004
Q14	JoinCselAselB	583	1512
Q18	1% projection	864	1495
Q23	Minimum aggregate	0.21	0.83
Q29	Insert 1 tuple	0.99	0.57
Q30	Delete 1 tuple	0.65	0.66
Q31	Update on `unique2`	1.47	0.73
Q32	Update on `unique1`	0.75	0.71
Total		**3441**	**8079**

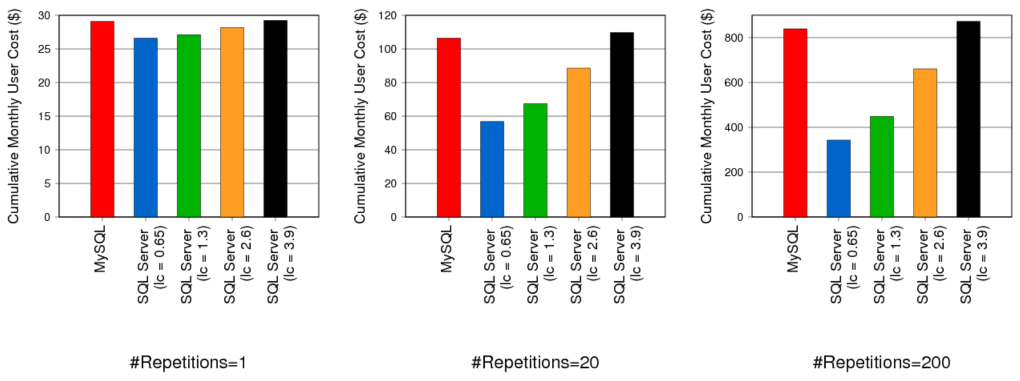


Figure 5.2: Cumulative monthly user cost as a function of workload repetitions, DBMS type and pricing model (`mixedworkload1`).

with MySQL but completed using SQL Server in a reasonable amount of time (between 400-1000 seconds for each query).

Figure 5.2 shows how the total user cost is affected by the performance gap that exists between the two systems, when the repetitions of the workload as well as the hourly license fee for the commercial DBMS is varied. As shown in this figure, when the workload is executed only once, the difference in cost between SQL Server and MySQL is very small. In this case, the execution time is not long enough to make a significant impact, and thus the total cost is dominated by the monthly storage fee. The difference in the total cost between the two systems increases with the number of queries issued. As shown in the figure, the "free" open-source DBMS results in higher total cost when the license fee for the commercial DBMS is below \$3.90. When the workload is executed 20 times the cost savings with SQL Server is 17%($\mathtt{lc} = \$2.60$), 37% ($\mathtt{lc} = \$1.30$) and 47% ($\mathtt{lc} = \$0.65$). In the case of 200 repetitions (almost a month running time with MySQL), when the license fee is \$2.60, using MySQL results in a 21% increase in the user's monthly bill. In the case of a license fee of \$0.65, the increase is more significant(59%).

Regarding, the performance of the `mixedworkload2`, as shown in Table 5.2, when the clustered and non-clustered indices are used, MySQL is approximately 2.34X (1.28 hours) slower than SQL Server. In this case, the existence of the clustered index on the `unique2` attribute significantly improved the execution of some joins (Q12, Q13, Q14) as well selections (Q3, Q4) and updates (Q29, Q31). The existence of the non-clustered index on the `unique1` attribute improved the performance of the queries 30 and 32. However, it had an adverse impact on other queries (e.g Q5 in MySQL). This behavior can be attributed to the fact that the non-clustered index contains only two attributes: `unique1` and the primary key `unique2`. However, the query result contains all the 16 attributes of the relation. Evaluating this query using the non-clustered index as an access method possibly results in high random I/O behavior. A clustered index scan would probably result in a more efficient query execution (as was the case for the similar Q6 in both MySQL and SQL Server).

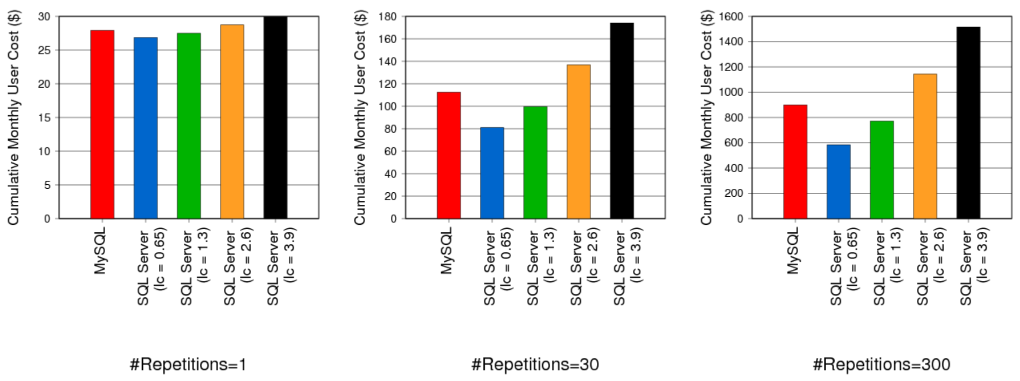


Figure 5.3: Cumulative monthly user cost as a function of workload repetitions, DBMS type and pricing model (`mixedworkload2`).

Figure 5.3 presents the total user cost, similarly to Figure 5.2, for `mixedworkload2`. As before, the free MySQL systems often results in higher costs, though now the license fee for the commercial DBMS has to be lower (around or below $1.30) than it was in Figure 5.2 to win over MySQL.

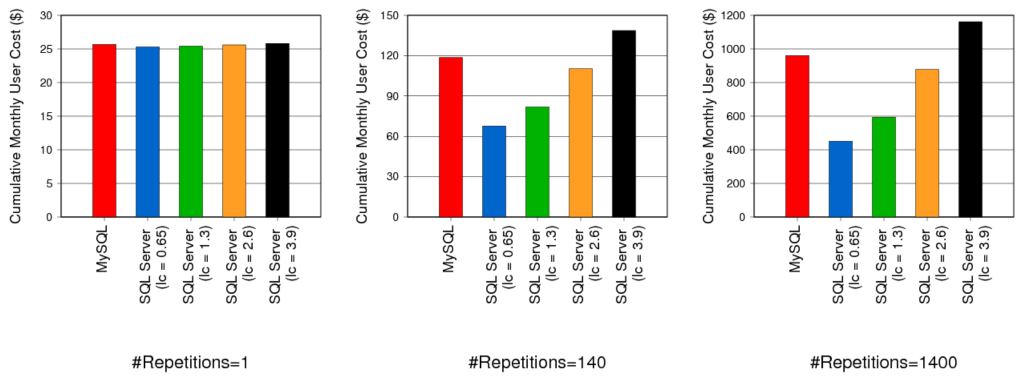


Figure 5.4: Cumulative monthly user cost as a function of workload repetitions, DBMS type and pricing model (`DSS Workload 1`).

5.3.6 DSS Workloads

In this section, we evaluate the performance of the two DBMSs when the workload contains only decision-support queries. Similar to Section 5.3.5, based on these results and the cost model developed in Section 5.2 we estimate the total user cost for both cases. The DSS workload includes all the join and aggregation queries of the Wisconsin benchmark. Again, we report execution times only for the queries that were completed in both systems.

Table 5.3: DSS Workload 1

Query	**Query Description**	**SQL Server Time (secs)**	**MySQL Time (secs)**
Q20	Minimum aggregate	190	482
Q21	Minimum aggregate with 100 partitions	185	621
Q22	Sum aggregate with 100 partitions	187	747
All		**562**	**1850**

Table 5.4: DSS Workload 2

Query	Query Description	SQL Server Time (secs)	MySQL Time (secs)
Q12	JoinAselB	412	1071
Q13	JoinABprime	408	1004
Q14	JoinCselAselB	583	1512
Q23	Minimum aggregate	0.21	0.83
All		**1403**	**3588**

Regarding `dssworkload1`, as it is shown in Table 5.3, when using heap-files as a storage layout only the aggregation queries were completed in both systems. In this case, MySQL was approximately 3.29X slower than SQL Server.

Figure 5.4 presents the total cost for the user varying the same parameters as in Figures 5.2 and 5.3. As it is shown in this figure, a similar pattern to that of `mixedworkload1` is observed. The per hour cheap option (MySQL) does not always result in the lowest total cost. In fact, when the hourly license fee for SQL Server is less or equal to \$2.60, choosing that over the free DBMS can result in cost savings of up to 53% (lc = \$0.65, 1400 repetitions). On the other hand, using MySQL can result in cost savings of up to 17% when the license fee is equal to \$3.90 and the workload is executed 1400 times.

Table 5.4 presents performance results for `dssworkload2`. The existence of the indices allows many joins to complete with MySQL, but negatively affected some aggregation queries. The reasons for this behavior are discussed in section 5.3.5. In this case, MySQL is 2.55X slower than SQL Server.

The corresponding user cost is presented in Figure 5.5. Similarly to the previous results, the open-source DBMS is a more cost-effective choice when the license fee is greater or equal to \$2.60. As before, the cost savings

Table 5.5: OLTP Workload 1

Query	Query Description	SQL Server Time (secs)	MySQL Time (secs)
Q1	1% selection on `unique2`	224	665
Q5	1% selection on `unique1`	195	739
Q7	Single tuple selection to screen	191	555
Q26	Insert 1 tuple	0.2	0.23
Q27	Delete 1 tuple	192	637
Q28	Update on `unique2`	192	595
Q32	Update on `unique1`	197	609
All		**1191**	**3800**

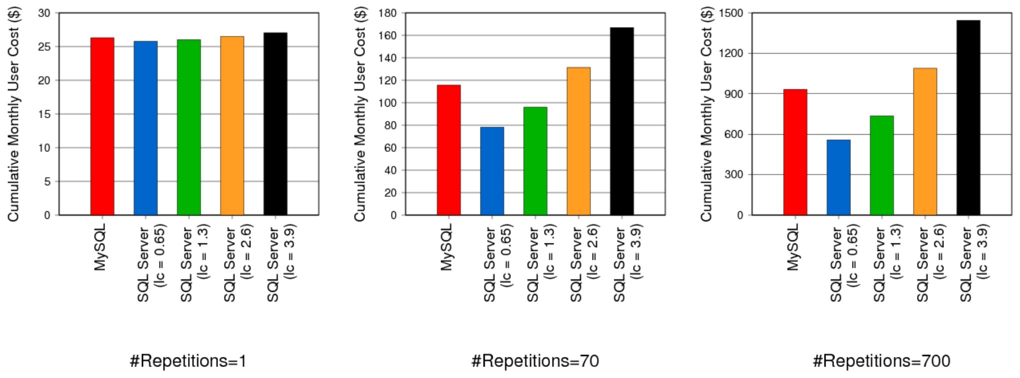


Figure 5.5: Cumulative monthly user cost as a function of workload repetitions, DBMS type and pricing model (`DSS Workload 2`).

increases as the execution time increases, since in this case the monthly storage fee does not have a significant impact on the total cost.

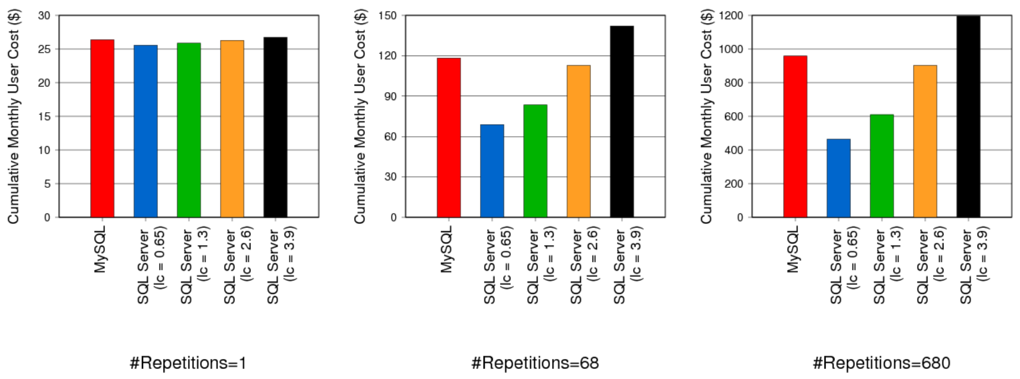


Figure 5.6: Cumulative monthly user cost as a function of workload repetitions, DBMS type and pricing model (`OLTP Workload 1`).

5.3.7 OLTP Workloads

The OLTP workload consists of the queries of the Wisconsin benchmark that contain high-selective selections, insertions, deletions and updates. Similarly to the previous experiments, only the queries that were completed in both DBMSs are reported for each workload.

As shown in Table 5.5, when the database consists only of heapfiles, MySQL is 3.19X slower than SQL Server. The corresponding user's cost is presented in Figure 5.6. Similarly to the previous experiments, MySQL is the most cost-effective option when the hourly license fee is equal to $3.90. In all the other cases, the cost savings when using SQL Server can be as high as 51%.

When indices are used, MySQL is approximately 2X slower than SQL Server. Table 5.6 and Figure 5.7 present the performance results and the associated user cost.

5.3.8 Discussion

We have shown that the process of estimating the cost of a DaaS is not straightfoward, even in the simple case where the database system in not deployed in a virtualized environment and factors such as different

Table 5.6: OLTP Workload 2

Query	Query Description	SQL Server Time (secs)	MySQL Time (secs)
Q3	1% selection on `unique2`	22	51
Q7	Single tuple selection to screen	0.75	0.60
Q29	Insert 1 tuple	0.99	0.57
Q30	Delete 1 tuple	0.65	0.66
Q31	Update on `unique2`	1.47	0.73
Q32	Update on `unique1`	0.75	0.71
All		**26.61**	**54.27**

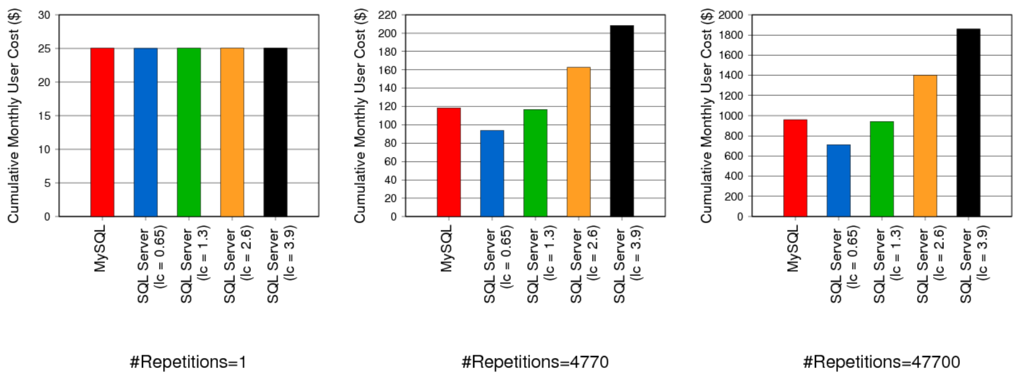


Figure 5.7: Cumulative monthly user cost as a function of workload repetitions, DBMS type and pricing model (`OLTP Workload 2`).

availability zones, locations or points of time are not taken into consideration. Parameters such as database efficiency, type of workload, and pricing model can all affect the resulting user cost. Consequently, often the option that initially seems cheap per hour, e.g., an open-source DBMS, can actually result in a higher monthly bill than that of a non-free, licensed

DBMS.

5.4 Related Work

DBMS benchmarking is an age-old sport in the database community. The Wisconsin benchmark [50] was one of the first benchmarks developed for evaluating RDBMSs. Today, the series of the TPC benchmarks [88] are widely used for measuring the performance and the cost or relational database systems.

Following the advent of cloud computing, recent work has evaluated different cloud services on different types of workloads. More specifically, a recent paper [33] presents some initial ideas on what a general cloud benchmark should consider, focusing on the different kinds of cloud services and architectures and their corresponding pricing plans. One of the (many) considerations in this paper is the end-user cost. A follow-up work [67] presents an evaluation of different cloud services when running enterprise web applications with OLTP workloads. Along the same lines, Berkeley's Cloudstone project [87] proposes a workload and metrics to study cloud infrastructures that deploy Web 2.0 applications. Comparing different cloud services has also been the focus of the recent work by Garfinkel [58], which evaluates three popular Amazon Computing Services (EC2, S3 and SQS). Another work [44] compares a traditional open-source RDMS and existing cloud computing technology (HBase). Cooper et al. [43] propose a benchmark to compare different popular datastores like Cassandra and PNUTS.

Virtualization techniques have been widely adopted in cloud-based environments. In a recent paper [71], the performance of relational database systems running on top of virtual machines has been studied. Bose et al. [36] present performance results from experiments running TPC database workloads on top of virtual machines, and make the case for a database benchmark on top of virtual machines. The follow-up work [85] presents a high-level overview of TPC-V, a benchmark designed for database workloads running in virtualized environments.

Recently, Amazon announced its AWS Trusted Advisor Service [29]. This service checks existing user deployments for underutilization or idleness and suggests ways to reduce the cost by eliminating the underutilized components. The service also makes suggestions on how to potentially improve performance or increase the level of security and fault-tolerance. BaaS is a higher level abstraction that goes one step further. It is used before the user actually makes any deployment decision.

5.5 Summary

This chapter has explored how two important dimensions in cloud environments, namely performance and cost, are influenced when different types of DBMSs are chosen by a DaaS user. More specifically, we have used a variety of simple workloads and storage organizations to evaluate two different relational DBMSs (one open-source and one commercial RDBMS). Our results show that given the range of the pricing models and the flexibility of the "on-demand" allocation of resources in cloud-based environments, it is hard for a user to figure out their actual monthly cost upfront. Interestingly, DaaS settings that at first sight seem cheaper per hour (since the backend is an open-source DBMS) and thus more-cost effective, can result in higher total costs in the long-run, since the backend DBMS may have poor performance characteristics on the users' workload. On the other hand, a DaaS setting backed by a high performance commercial DBMSs, while more expensive on a per hour basis, may be cheaper overall since its higher performance more than makes up for the hourly price differential. We note that these results should not be construed to mean that free open source DBMSs are always more expensive in the DaaS environment (or vice versa) – we have only tried two DBMSs in this work, picking the most popular free open-source DBMS and a commercial DBMS. Rather, our work highlights that the real cost of running a workload in the DaaS is complicated, and may in some cases produce surprising results.

Thus, what we need is real transparency and clarity in pricing DaaS. An approach to this problem that we propose in this chapter is "Benchmark as a Service" (BaaS), where by the DaaS provider can take the user workload as input (with SLA parameters) and provide an accurate price for that workload, or perhaps different prices at different SLA levels. This BaaS approach would move the DaaS offering closer to a true utility model (like gas and electricity, or internet service).

Chapter 6

Conclusions and Future Work

This thesis has studied four aspects of efficient data processing under the broader umbrella of "big" data applications in cloud-based environments. First, we explored and evaluated the design of various data processing systems that can be deployed in cloud environments and its impact on the performance experienced by the users. Secondly, we focused on NoSQL systems tailored for analytical workloads, and proposed new storage layouts with the goal of improving their performance. Next, we focused on Database-as-a-Service environments and studied the database placement mechanisms that can be used in these environments, the guarantees that they can provide with respect to performance and availability and their impact of the total operating cost. Finally, we performed a study of the available DaaS pricing models and explored their effectiveness in making it easier for the end-user to determine the true cost that will be incurred when running a workload.

6.1 Contributions

In the first part of the thesis, we have evaluated two different classes of systems, SQL and NoSQL systems, on two major types of applications, namely decision support analysis and interactive data-serving using representative benchmarks. Our study has revealed, that RDBMSs can still provide better performance, in terms of both latency and throughput, for interactive data serving workloads. Regarding the analytical decision support applications, our analysis has revealed that the performance of the NoSQL system studied, still lags behind the traditional SQL system due to deficiencies in the storage layout and the optimization components of the NoSQL system.

In the second part of the thesis, we have focused on NoSQL systems tailored for analytical applications, like Hadoop [3]. The focus of this work was on a) considering how column-oriented techniques can be incorporated in Hadoop, b) implementing these techniques in a way that preserves the existing (and popular) programming paradigm offered by MapReduce, and c) evaluating the overall performance impact of these techniques. We have introduced a column-oriented storage format that is compatible with the replication and scheduling constraints of Hadoop, and have shown that with this technique we can speed up Hadoop jobs from a real workload by an order of magnitude. We have also shown that dealing with complex column types such as maps, arrays, and nested records (which are common in MapReduce jobs) incurs significant CPU overhead. We have introduced a novel skip list based column format and a lazy record construction strategy to avoid unnecessary deserialization and decompression providing an additional speedup of 1.5x for the map phase. In aggregate, our techniques can improve performance on Hadoop by one to two orders of magnitude.

In the next part of this thesis, we have investigated the replica placement mechanisms that can be used in Database-as-a-Service environments. More specifically, we have studied the *online* replica placement algorithms for multi-tenant DaaS environments and we have proposed an algorithm called RkC. Our study has shown that this algorithm has a unique combination of being simple to implement, has good load balancing properties both in the initial replica placement and in dealing with load-related changes following a node failure, has low initial hardware provisioning cost, and is able to guarantee tenant performance SLOs even without requiring global knowledge of the load across all the machines.

Finally, we explored the “pay-as-you-go” pricing models that are used in DaaS environments, by performing an experimental evaluation on different database systems and workloads. More specifically, we show that the current DaaS model can produce unpleasant surprises. Our study illustrates a scenario in which a DaaS service powered by a DBMS that has a lower hourly rate actually costs more to the end user than a DaaS service that is powered by another DBMS that charges a higher hourly rate. Our study points toward the need for a reform in the way the DaaS services are offered and priced to make it easier for the end-user to consider the true cost. One potential solution to this problem is for DaaS providers to offer a new service called Benchmark as a Service (BaaS) where in the user provides the parameters of their workload and SLA requirements, and get a price quote.

6.2 Future Directions

There are interesting directions for future work associated with each component of this thesis.

Regarding the performance comparison of SQL and NoSQL systems presented in Chapter2, an interesting direction for future work is to expand this work to other SQL and NoSQL systems and revisit the performance differences in a few years.

The work presented in Chapter 3, focuses on systems like Hadoop [3] and Hive [5] that are tailored for analytical workloads. However, there is an increasing trend of creating hybrid systems or integrating multiple existing systems into one platform, to facilitate the process of runnng both transactional and analytical workloads on the same data without having to migrate data from one system to the other. As an example, Hadoop [3] and HBase [4] can be installed on the same cluster and can operate on the same HDFS data. An interesting direction for future work is to extend our column oriented storage techniques to support transactional and analytical workloads at the same time. For example, supporting concurrent readers and writers and study the isolation guarantees provided is an interesting path for future research. Another interesting path for future work is to integrate our storage layouts with indexing and data partitioning.

For our work on the online replica placement problem presented in Chapter 4, to limit the scope of this study, we have made some simplifying assumptions on aspects such as performance models and tenant workload. Analyzing additional optimizations that can be performed in the case of non-linear characterizing functions is an interesting avenue for future work. Moreover, accommodating tenants that do not always run at peak performance is also an interesting direction for future research. Another promising direction is to explore if the techniques presented in this paper can be used in later stages of the replica placement process, for example when unexpected workload spikes are detected in practice. Exploring the modifications needed to accommodate heterogeneous clusters as well as studying the guarantees that our algorithms provide in case of leaving tenants are also important paths for future work.

Finally regarding our proposal on creating Benchmark-as-a-Service, we acknowledge that setting up a BaaS is challenging as there are important aspects that need to be considered. For example, how to specify the workload. A starting point for describing this workload could be for the user to provide the database schema, average tuple sizes for each table, and a query set. But, additional parameters may be required, such as estimated database growth rates, or acceptable ranges for SLA parameters (e.g., query/workload response time or throughput). For simplicity from the users' perspective it is desirable that the workload specifications should not be overly complicated, but from the DaaS provider's perspective more details are probably required. Finding a good and practical balance is one direction for future work. Other aspects of future work include designing methods for a DaaS provider to efficiently run a mix of workloads that started with a BaaS, and monitoring and reacting to changes in workloads that started with a price quote from the BaaS.

Bibliography

[1] Avro. `http://avro.apache.org`.

[2] Couchdb. `http://couchdb.apache.org/`.

[3] Hadoop. `http://hadoop.apache.org`.

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

[5] Hive. `http://hive.apache.org/`.

[6] Hive issue 2081. `https://issues.apache.org/jira/browse/HIVE-2081`.

[7] Hive issue 2130. `https://issues.apache.org/jira/browse/HIVE-2130`.

[8] Jaql. `http://code.google.com/p/jaql/`.

[9] LZO. `http://www.oberhumer.com/opensource/lzo/`.

[10] Microsoft sql server 2008 r2 parallel data warehouse. `http://www.microsoft.com/sqlserver/en/us/solutions-technologies/data-warehousing/pdw.aspx`.

[11] Mongodb. `http://www.mongodb.org/`.

[12] Mongodb - replica sets. `http://www.mongodb.org/display/DOCS/Replica+Sets`.

[13] Mongodb- mongostat. `http://www.mongodb.org/display/DOCS/mongostat`.

[14] Mongodb- splitting chunk shards. `http://www.mongodb.org/display/DOCS/Splitting+Shard+Chunks`.

[15] Nutch. `http://nutch.apache.org/`.

[16] Pig. `http://pig.apache.org/`.

[17] Protocol Buffers. `http://code.google.com/p/protobuf/`.

[18] Riak. `http://wiki.basho.com/`.

[19] Running tpc-h queries on hive. `https://issues.apache.org/jira/browse/HIVE-600`.

[20] Thrift. `http://incubator.apache.org/thrift/`.

[21] The tpc-h benchmark. `http://www.tpc.org/tpch/`.

[22] D. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, pages 967–980, 2008.

[23] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[24] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, pages 466–475, 2007.

[25] A. Aboulnaga, K. Salem, A. A. Soror, U. F. Minhas, P. Kokosielis, and S. Kamath. Deploying database appliances in the cloud. *IEEE Data Eng. Bull.*, 32(1):13–20, 2009.

[26] 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.

[27] A. Ailamaki, D. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[28] Amazon Relational Database Service. `http://aws.amazon.com/rds/`.

[29] Amazon Trusted Advisor. `https://aws.amazon.com/premiumsupport/trustedadvisor/`.

[30] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing, 2009.

[31] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[32] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *ICDE*, pages 1255–1263, 2011.

[33] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *DBTest*, 2009.

[34] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, pages 241–252, 2010.

[35] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[36] S. Bose, P. Mishra, P. Sethuraman, and H. R. Taheri. Benchmarking database performance in a virtual environment. In *TPCTC*, pages 167–182, 2009.

[37] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. pages 80–87, 2002.

[38] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.

[39] H. Cai, B. Reinwald, N. Wang, and C. Guo. Saas multi-tenancy: Framework, technology, and case study. *IJCAC*, 1(1):62–77, 2011.

[40] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *SODA*, pages 185–194, 1999.

[41] Q. Chen, A. Therber, M. Hsu, H. Zeller, B. Zhang, and R. Wu. Efficiently Support MapReduce-like Computation Models Inside Parallel DBMS. In *IDEAS*, pages 43–53, 2009.

[42] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(2):1459–1468, 2010.

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

[44] J.-D. Cryans, A. April, and A. Abran. *Criteria to Compare Cloud Computing with Current Database Technology*, volume 5338 LNCS, pages 114–126. Springer-Verlag, 2008.

[45] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD Conference*, pages 313–324, 2011.

[46] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.

[47] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[48] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1):107–113, 2008.

[49] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53:72–77, January 2010.

[50] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[51] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1):518–529, 2010.

[52] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[53] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD Conference*, pages 337–348, 2011.

[54] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*, pages 301–312, 2011.

[55] A. J. Elmore, S. Das, A. Puche, D. Agrawal, A. E. Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *SIGMOD Conference*, 2013.

[56] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[57] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. 1979.

[58] S. L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3 and sqs. Technical report, 2007.

[59] J. Hamilton. Cooperative expendable micro-slice servers (cems): Low cost, low power servers for internet-scale services.

[60] Y. He, R. Lee, S. Zheng, N. Jain, Z. Xu, and X. Zhang. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE*, 2011.

[61] M. Hsu, Q. Chen, R. Wu, B. Zhang, and H. Zeller. Generalized UDF for Analytics Inside Database Engine. *LNCS*, 6184:742–754, 2010.

[62] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-Stores. In *SIGMOD*, pages 297–308, 2009.

[63] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.

[64] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: right shoes for a running elephant. In *SoCC*, page 21, 2011.

[65] D. S. Johnson, A. J. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.

[66] T. Kaldewey, E. J. Shekita, and S. Tata. Clydesdale: structured data processing on mapreduce. In *EDBT*, pages 15–25, 2012.

[67] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD Conference*, pages 579–590, 2010.

[68] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance slos. In *ICDE*, pages 702–713, 2012.

[69] C. Lemka, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding Up Queries in Column Stores. 6263:117–129, 2010.

[70] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.

[71] U. F. Minhas, J. Yadav, A. Aboulnaga, and K. Salem. Database systems on virtual machines: How much do you lose? In *ICDE Workshops*, pages 35–41, 2008.

[72] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[73] H. J. Moon, H. Hacigümüs, Y. Chi, and W.-P. Hsiung. Swat: a lightweight load balancing method for multitenant databases. In *EDBT*, pages 65–76, 2013.

[74] MySQL. `http://www.mysql.com/`.

[75] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX*, pages 267–273, 2008.

[76] Oracle Database. `http://www.oracle.com/us/products/database/index.html`.

[77] R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA*, pages 121–133, 2001.

[78] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for vector bin packing. `http://research.microsoft.com/apps/pubs/default.aspx?id=147927`.

[79] 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*, pages 165–178, 2009.

[80] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *CACM*, 33(6):668–676, 1990.

[81] B. Reinwald. Multitenancy. `http://www.cs.washington.edu/mssi/2010/BertholdReinwald.pdf`, 2010.

[82] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1):460–471, 2010.

[83] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, pages 1264–1275, 2011.

[84] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. Franklin, and D. Jacobs. Rtp: Robust tenant placement for elastic in-memory database clusters. In *SIGMOD Conference*, 2013.

[85] P. Sethuraman and H. R. Taheri. Tpc-v: A benchmark for evaluating the performance of database applications in virtual environments. In *TPCTC*, pages 121–135, 2010.

[86] P. Shivam, V. Marupadi, J. S. Chase, T. Subramaniam, and S. Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX Annual Technical Conference*, pages 241–254, 2008.

[87] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[88] TPC Benchmarks. `http://www.tpc.org/information/benchmarks.asp`.

[89] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small apllications. In *CIDR*, 2009.