Performance Evaluation of Distributed SQL Query Engines and Query Time Predictors

Stefan van Wouw

Delft University of Technology

“Work expands so as to fill the time available for its completion.”

– Cyril Northcote Parkinson

Performance Evaluation of Distributed SQL Query Engines and Query Time Predictors

Master's Thesis in Computer Science

Parallel and Distributed Systems Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Stefan van Wouw

10th October 2014

Author
Stefan van Wouw

Title
Performance Evaluation of Distributed SQL Query Engines and Query Time Predictors

MSc presentation
29th October 2014

Graduation Committee

Prof.dr.ir. D.H.J. Epema (chair)	Delft University of Technology
Dr.ir. A. Iosup	Delft University of Technology
Dr.ir. A.J.H. Hidders	Delft University of Technology
Dr. J.M. Viña Rebolledo	Azavista, Amsterdam

Abstract

With the decrease in cost of storage and computation of public clouds, even small and medium enterprises (SMEs) are able to process large amounts of data. This causes businesses to increase the amounts of data they collect, to sizes that are difficult for traditional database management systems to handle. Distributed SQL Query Engines (DSQEs), which can easily handle these kind of data sizes, are therefore increasingly used in a variety of domains. Especially users in small companies with little expertise may face the challenge of selecting an appropriate engine for their specific applications. A second problem lies with the variable performance of DSQEs. While all of the state-of-the-art DSQEs claim to have very fast response times, none of them has performance guarantees. This is a serious problem, because companies that use these systems as part of their business *do* need to provide these guarantees to their customers as stated in their Service Level Agreement (SLA).

Although both industry and academia are attempting to come up with high level benchmarks, the performance of DSQEs has never been explored or compared in-depth. We propose an empirical method for evaluating the performance of DSQEs with representative metrics, datasets, and system configurations. We implement a micro-benchmarking suite of three classes of SQL queries for both a synthetic and a real world dataset and we report response time, resource utilization, and scalability. We use our micro-benchmarking suite to analyze and compare three state-of-the-art engines, viz. Shark, Impala, and Hive. We gain valuable insights for each engine and we present a comprehensive comparison of these DSQEs. We find that different query engines have widely varying performance: Hive is always being outperformed by the other engines, but whether Impala or Shark is the best performer highly depends on the query type.

In addition to the performance evaluation of DSQEs, we evaluate three query time predictors of which two are using machine learning, viz. multiple linear regression and support vector regression. These query time predictors can be used as input for scheduling policies in DSQEs. The scheduling policies can then change query execution order based on the predictions (e.g., give precedence to queries that take less time to complete). We find that both machine learning based predictors have acceptable performance, while a baseline naive predictor is more than two times less accurate on average.

Preface

Ever since I started studying Computer Science I have been fascinated about the ways tasks can be distributed over multiple computers and be executed in parallel. Cloud Computing and Big Data Analytics appealed to me for this very reason. This made me decide to conduct my thesis project at Azavista, a small start-up company based in Amsterdam specialised in providing itinerary planning tools for the meeting and event industry. At Azavista there is a particular interest in providing answers to analytical questions to customers in near real-time. This thesis is the result of the efforts to realise this goal.

During the past year I have learned a lot in the field of Cloud Computing, Big Data Analytics, and (Computer) Science in general. I would like to thank my supervisors Prof.dr.ir. D.H.J Epema and Dr.ir. A. Iosup for their guidance and encouragement throughout the project. Me being a perfectionist, it was very helpful to know when I was on the right track. I also want to thank my colleague and mentor Dr. José M. Viña Rebolledo for his many insights and feedback during the thesis project. I am very grateful both him and my friend Jan Zahálka helped me understand machine learning, which was of great importance for the second part of my thesis.

I want to thank my company supervisors Robert de Geus and JP van der Kuijl for giving me the freedom to experiment and providing me the financial support for running experiments on Amazon EC2. Furthermore I want to also thank my other colleagues at Azavista for the great time and company, and especially Mervin Graves for his technical support.

I want to thank Sietse Au, Marcin Biczak, Mihai Capotă, Bogdan Ghiţ, Yong Guo, and other members of the Parallel and Distributed Systems Group for sharing ideas. Last but not least, I want to also thank my family and friends for providing great moral support, especially during the times progress was slow.

Stefan van Wouw

Delft, The Netherlands
10th October 2014

Contents

Chapter 1

Introduction

With the decrease in cost of storage and computation of public clouds, even small and medium enterprises (SMEs) are able to process large amounts of data. This causes businesses to increase the amounts of data they collect, to sizes that are difficult for traditional database management systems to handle. Exactly this challenge was also encountered at Azavista, the company this thesis was conducted at. In order to assist customers in planning itineraries using its software for event and group travel planning, Azavista processes multi-terabyte datasets every day. Traditional database management systems that were previously used by this SME simply did not scale along with the size of the data to be processed.

The general phenomenon of exponential data growth has led to Hadoop-oriented Big Data Processing Platforms that can handle multiple terabytes to even petabytes with ease. Among these platforms are stream processing systems such as S4 [44], Storm [22], and Spark Streaming [64]; general purpose batch processing systems like Hadoop MapReduce [6] and Haloop [25]; and distributed SQL query engines (DSQEs) such as Hive [53], Impala [15], Shark [59], and more recently, Presto [19], Drill [35], and Hive-on-Tez [7].

Batch processing platforms are able to process enormous amounts of data (terabytes and up) but have relatively long run times (hours, days, or more). Stream processing systems, on the other hand, have immediate results when processing a data stream, but can only perform a subset of algorithms due to not all data being available at any point in time. Distributed SQL Query Engines are generally built on top of (a combination of) stream and batch processing systems, but they appear to the user as if they were traditional relational databases. This allows the user to query structured data using an SQL dialect, while at the same time having much higher scalability than traditional databases. Besides these different systems, hybrids also do exist in form of so called lambda architectures [17], where data is both processed by a batch processing system and by a stream processor. This allows the stream processing to get fast but approximate results, while in the back the batch processing system slowly computes the results accurately.

In this work we focus on the DSQEs and their internals, since although authors

claim them to be fast and scalable, none of them provides deadline guarantees for queries with deadlines. In addition, no in-depth comparisons between these systems are available.

1.1 Problem Statement

Selecting the most suitable of all available DSQEs for a particular SME is a big challenge, because SMEs are not likely to have the expertise and the resources available to perform an in-depth study. Although performance studies do exist for Distributed SQL Query Engines [4, 16, 33, 34, 47, 59], many of them only use synthetic workloads or very high-level comparisons that are only based on query response time.

A second problem lies with the variable performance of DSQEs. While all of the state-of-the-art DSQEs claim to have very fast response times (seconds instead of minutes), none of them has performance guarantees. This is a serious problem, because companies that use these systems as part of their business *do* need to provide these guarantees to their customers as stated in their Service Level Agreement (SLA). There are many scenarios where multiple tenants[1] are using the same data cluster and resources need to be shared (e.g., Google's BigQuery). In this case, queries might take much longer to complete than in a single-tenant environment, possibly violating SLAs signed with the end-customer. Related work provides a solution to this problem in form of BlinkDB [24]. This DSQE component for Shark does provide bounded query response times, but at the cost of less accurate results. However, one downside of this component is that it is very query engine dependent, as it uses a cost-based analytical heuristic to predict the execution time of different parts of a query.

In this thesis we try to address the lack of in-depth performance evaluation of the current state-of-the-art DSQEs by answering the following research question:

RQ1 What is the performance of state-of-the-art Distributed SQL Query Engines in a single-tenant environment?

After answering this question we evaluate an alternative to BlinkDB's way of predicting query time by answering the following research question:

RQ2 What is the performance of query time predictors that utilize machine learning techniques?

Query time predictors are able to predict both the query execution time and query response time. The query execution time is the time a query is actively being processed, whereas the query response time also takes the queue wait time

[1] A tenant is an actor of a distributed system that represents a group of end-users. For example, a third party system that issues to analyze some data periodically, in order to display it to all its users on a website.

into account. The predicted execution time can be used to change the query execution order in the system as to minimize response time. We are particularly interested in applying machine learning techniques, because it has been shown to yield promising results in this field [60]. In addition, machine learning algorithms do not require in-depth knowledge of inner DSQE mechanics. Thus, machine learning based query time predictors can easily be applied to any query engine, while BlinkDB is tied to many internals of Shark.

1.2 Approach

To answer the first research question (**RQ1**) we define a comprehensive performance evaluation method to assess different aspects of query engines. We compare Hive, a somewhat older but still widely used query engine, with Impala and Shark, both state-of-the-art distributed query engines. This method can be used to compare current and future query engines, despite not covering all the methodological and practical aspects of a true benchmark. The method focuses on three performance aspects: processing power, resource utilization and scalability. With the results from this study, system developers and data analysts can make informed choices related to both cluster infrastructure and query tuning.

In order to answer research question two (**RQ2**), we evaluate three query time predictor methods, namely *Multiple Linear Regression (MLR)*, *Support Vector Regression (SVR)*, and a base-line method *Last2*. We do this by designing a workload and training the predictors on the output traces of the three different ways we executed this workload. Predictor accuracy is reported by using three complementary metrics. The results from this study allow engineers to select a predictor for use in DSQEs that is both fast to train and accurate.

1.3 Thesis Outline and Contributions

The thesis is structured as follows: In Chapter 2 we provide background information and related work, including the definition of the Cloud Computing paradigm, an overview of state-of-the-art Distributed SQL Query Engines and background information regarding machine learning. In Chapter 3 we evaluate the state-of-the-art Distributed SQL Query Engines' performance on both synthetic real-world data. In Chapter 4 we evaluate query time predictors that use machine learning techniques. In Chapter 5 we present conclusions to our work and describe directions for future work.

Our main contributions are the following:

- We propose a method for performance evaluation of DSQEs (Chapter 3), which includes defining a workload representative for SMEs as well as defining the performance aspects of the query engines: processing power, resource utilization and scalability.

- We define a micro-benchmark setup for three major query engines, namely Shark, Impala and Hive (Chapter 3).
- We provide an in-depth performance comparison between Shark, Impala and Hive using our micro-benchmark suite (Chapter 3).
- We design a performance evaluation method for evaluating 3 different query time predictors, namely MLR, SVR and Last2. This method includes workload design and performance metric selection (Chapter 4).
- We provide an in-depth performance comparison between MLR, SVR and Last2 on output traces of the workload we designed (Chapter 4).

The material in Chapter 3 is the basis of the article that was submitted to ICPE'15:

[57] Stefan van Wouw, José Viña, Dick Epema, and Alexandru Iosup. *An Empirical Performance Evaluation of Distributed SQL Query Engines*. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE), 2015.

Chapter 2

Background and Related Work

In this chapter we provide background information to our work. In Section 2.1 the field of Cloud Computing is introduced. Section 2.2 provides an overview of state-of-the-art Distributed SQL Query Engines, followed by related performance studies of these engines in Section 2.3. Section 2.4 discusses the basics of machine learning, followed by Section 2.5 which describes the basic idea of principal component analysis. Both are needed to understand the machine learning approach we used for the query time predictors in Chapter 4.

2.1 Cloud Computing

As defined by the National Institute of Standards and Technology (NIST), *Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction* [42].

In this section we describe the Cloud Service Models that exist in Cloud Computing (Section 2.1.1) and the costs involved with these services (Section 2.1.2). Big Data Processing Platforms related to our work are discussed in Section 2.1.3.

2.1.1 Cloud Service Models

Three major service models exist in Cloud Computing (see Figure 2.1). The Infrastructure as a Service (IaaS) model is the lowest level model among the three. In this model one can lease virtual computing and storage resources, allowing the customer to deploy arbitrary operating systems and applications. These virtual resources are typically offered to the customer as Virtual Machines (VMs) accessible through SSH. Amazon EC2 and S3 [2, 3], Windows Azure [23] and Digital Ocean Simple Cloud Hosting [10], are all examples of IaaS services.

The Platform as a Service (PaaS) model offers platforms upon which applications can be developed using a certain set of programming models and program-

ming languages. The customer does not have control over the underlying infrastructure. Examples of PaaS services are Google BigQuery [14] and Amazon Elastic MapReduce (EMR) [1]. Google BigQuery offers a platform on which terabytes to petabytes of data can be queried by means of BigQuery's SQL dialect. Amazon EMR offers a platform to execute MapReduce jobs on-demand.

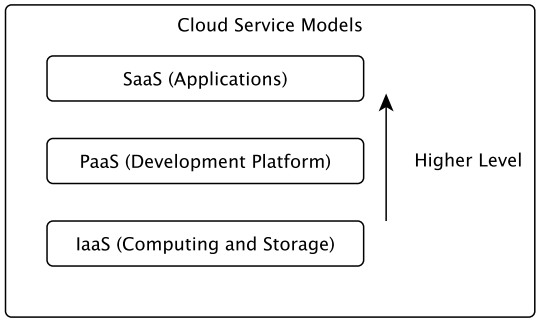


Figure 2.1: An overview of the three major Cloud Service Models.

The last cloud service model is Software as a Service (SaaS). With SaaS the customer uses applications running in the Cloud. Examples include Dropbox [11] for storing documents in the Cloud, as well as Gmail [12] for e-mail and SAP [21] for Enterprise Resource Planning (ERP).

2.1.2 Cloud Vendors and Cloud Costs

Many different cloud vendors exist, all offering different services, ranging from IaaS to PaaS to SaaS. SaaS services are usually either free of charge (Gmail) and paid by other means such as ads, or require a monthly subscription (Dropbox, SAP). We do not describe the details of SaaS costs, since SaaS applications are not related to our work. Instead, we discuss the IaaS cost models employed by different cloud vendors, as well as the cost models of PaaS platforms.

IaaS computing services are typically charged per full hour per VM instance, and IaaS storage services are typically charged per GB/month used and per I/O operation performed. In addition, data-out (data transfer crossing data center's borders) is also charged for. Table 2.1 gives a condensed overview of the IaaS compute cloud costs of Amazon EC2 for Linux in the US East region[1]. There are many different instance types, each optimized for either low cost, high computing power, large memory pools, high disk I/O, high storage, or a combination of these. The storage accompanied with these instances is included in the hourly price, and will be freed when an instance is destroyed.

In some cases no instance storage is provided, then additional costs apply for the Elastic Block Store (EBS) allocated to that instance. A benefit of EBS storage,

[1]For an up-to-date pricing overview see `http://aws.amazon.com/ec2/pricing/`

Table 2.1: IaaS costs for Linux Amazon EC2 in the US East region (Condensed, only the ones with lowest and highest specifications per category). One EC2 Compute Unit (ECU) is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

Type	**Category**	**Memory (GiB)**	**vCPU/ECU**	**Storage (GB)**	**Cost ($/h)**
t1.micro	Low-cost	0.615	1/Variable	EBS	0.020
m1.small	General	1.7	1/1	160	0.060
m3.2xlarge	General	30	8/26	EBS	1.000
c1.medium	High-CPU	1.7	2/5	350	0.145
c1.xlarge	High-CPU	7	8/20	1,680	0.580
m2.xlarge	High-Mem	17.1	2/6.5	850	0.410
m2.4xlarge	High-Mem	68.4	8/26	1,680	1.640
hi1.4xlarge	High-I/O	60.5	16/35	2,048 (SSD)	3.100
hs1.8xlarge	High-Storage	117	16/35	49,152	4.600
cc2.8xlarge	High-CPU+Network	60.5	32/88	3,360	2.400
cr1.8xlarge	High-Mem+Net+I/O	244	32/88	240 (SSD)	3.500

Table 2.2: Additional costs for Amazon EC2.

Service Component	**Cost ($)**
Data Transfer IN	FREE
Data Transfer OUT (Another Region)	0.02/GB
Data Transfer OUT (Internet)	0.05-0.12/GB
EBS Space Provisioned	0.10/GB/Month
I/O operations	0.10/One Million

however, is that it can persist even after an instance is destroyed. An overview of EBS and data-out costs is depicted in Table 2.2.

Cloud vendors such as GoGrid [13], RackSpace [20] and CloudCentral [8], have a similar pricing scheme as Amazon. However, they do not charge for I/O operations. Digital Ocean [10] (among others) does not charge per GB of data transfered separately from the hourly instance cost, unless you exceed the fair-use limit of some TB per month.

In addition to the public cloud services offered by all of these cloud vendors, some vendors also offer private or hybrid cloud services. In this case the consumer does not have to share resources with other tenants (or in case of hybrid cloud, only partly). This can improve performance a lot, but naturally comes at a higher price per instance.

PaaS services are charged, like IaaS services, on a pay-as-you-go basis. For Amazon Elastic MapReduce this comes down to roughly 15% to 25% of the IaaS instance price[2] (see Table 2.1). Another example is Google BigQuery where the consumer is charged for the amount of data processed instead of the time this takes.

[2]Up to date prices at `http://aws.amazon.com/elasticmapreduce/pricing/`

Table 2.3: Overview of differences between platforms.

	Batch Processing	**Stream Processing**	**Interactive Analytics**
Run Time	hours/days	continuous	seconds/minutes
Response Time	hours/days	(milli)seconds	seconds/minutes
Processing Capability	TBs/PBs	TBs/PBs	GBs/TBs
Excels at	Aggregation	Online algorithms	Iterative Algorithms
Less suitable for	Online/Iterative algorithms	Iterative Algorithms	Online Algorithms

2.1.3 Big Data Processing Platforms

Since our work concerns the processing of gigabytes to terabytes or even petabytes of data, we consider the relevant state-of-the-art Big Data processing platforms in this section. Two main Big Data processing categories exist: batch processing, and stream processing. Batch processing platforms are able to process enormous amounts of data (Terabytes and up) but have relatively long run times (hours, days, or more). These systems excel at running algorithms that require an entire multi-terabyte dataset as input.

Stream processing platforms can also handle enormous amounts of data, but instead of requiring an entire dataset as input, these platforms can immediately process an ongoing datastream (e.g. lines in a log file of a web server), and return a continuous stream of results. The benefit of these platforms is that results will start coming in immediately. However, stream processing is only suitable for algorithms that can work on incomplete datasets (e.g. online algorithms), without requiring the whole dataset as input.

Besides the two main categories, a new category starts to take form which we call: Interactive analytics. The platforms in this category attempt to get close to the fast response time of stream processing platforms by heavy use of intermediate in-memory storage, while not limiting the kind of algorithms that can be run. This allows data analysts to explore properties of datasets without having to wait multiple hours for each query to complete. Because of the intermediate in-memory storage, these systems are very suitable for iterative algorithms (such as many machine learning and graph processing algorithms). An overview of the major differences and similarities between these three types of platforms can be found in Table 2.3.

All these data processing platforms's implementations are based on certain programming models. Programming models are generalized methodologies with which certain types of problems can be solved. For instance, a very popular programming model in batch processing is MapReduce, introduced by Google in 2004 [29]. In this programming model a *map* function can be defined over a dataset, which emits a key-value tuple for of the values in its input. The *reduce* function then reduces all values belonging to a unique key to a single value. Multiple map functions can be run in a distributed fashion, each processing part of the input. A popular example to illustrate how MapReduce works is WordCount. In this example the *map* func-

tion gets a list of words and emits each word in this list as key, together with the integer 1 as value. The *reduce* function then receives a list of 1s for each unique word in the original input of *map*. The *reduce* function can can sum over these 1s to get the total number of occurences per word in the original input.

MapReduce has been implemented in many execution engines (frameworks that implement programming models), such as Hadoop [6], Haloop [25] and the more recent YARN (which is basically a general cluster resource manager capable of running more than just Hadoop MapReduce) [52].

When the Hadoop execution engine was introduced, it came with Hadoop Distributed File System (HDFS), a fault tolerant storage engine which allows for storing unstructured data in a distributed fashion. Later, high level languages were introduced on top of the Hadoop stack to ease development. PigLatin is one of such lanuages, which is converted to native MapReduce jobs by the Pig interpreter [46].

Figure 2.2 gives an overview of state-of-the-art Big Data processing platforms and places them in the right layer of the Big Data processing stack. The frameworks on the border between programming models and execution engines (as seen in the figure) all are execution engines that have implemented their own programming model.

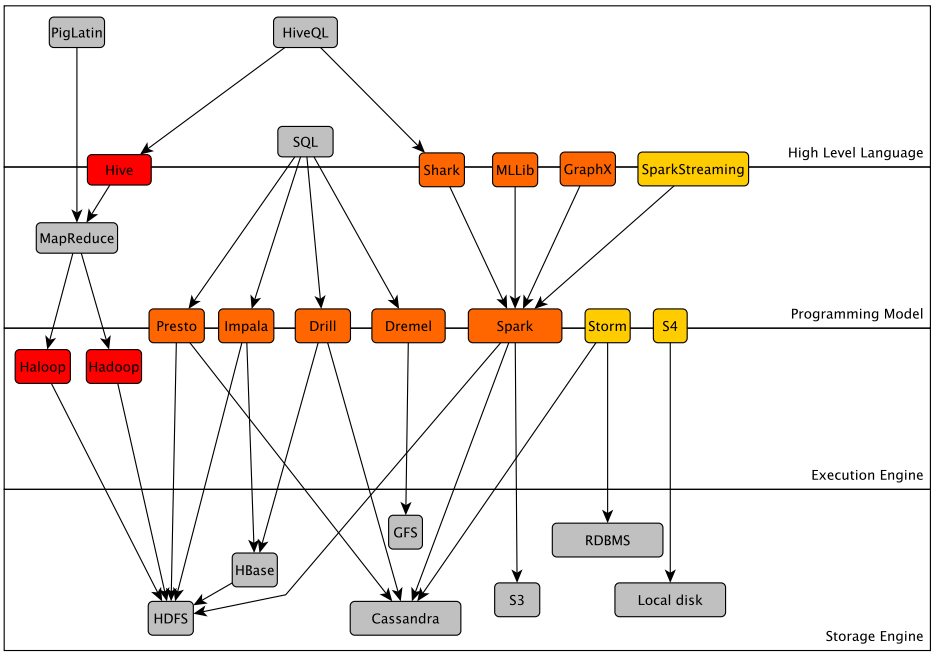


Figure 2.2: An overview of the Big Data processing Stack(red: Batch Processing; orange: Interactive Analytics; yellow: Stream Processing).

In the category of stream processing platforms, S4 [44], Storm [22] and Spark Streaming [64] are the most noticeable. Impala [15], Drill [35], Dremel [43], Presto [19], and Shark [32] are all Distributed SQL Query Engines which fall under the interactive analytics category. While Hive is also a Distributed SQL Query Engine, it is considered to be batch processing, because it directly translates its queries to Hadoop MapReduce jobs.

Spark [63] is the main execution engine powering both Spark Streaming and Shark. This execution engine also powers GraphX [58] for Graph Processing and MLLib [18] for Machine Learning. As our focus lies with the Distributed SQL Query Engines, we will explain these in more detail in Section 2.2.

2.2 State-of-the-Art Distributed SQL Query Engines

Distributed SQL Query Engines appear to the user as if they were relational databases, allowing the user to query structured data using an SQL dialect, while at the same time having much higher scalability. For instance, Google Dremel [43], one of the first Interactive Distributed SQL Query Engines, is able to scale to thousands of CPUs and Petabytes of data.

In this Section we will give an overview of the architectures of the state-of-the-art Distributed SQL Query Engines. Starting with one of the oldest and most mature, but relatively slow systems: Hive (Section 2.2.1), followed by Google's Dremel (Section 2.2.2). Most of the other systems are heavily inspired by Dremel's internals, while building on Hive's Metadata Store. These systems are Impala (Section 2.2.3), Shark (Section 2.2.4), Presto (2.2.5) and Drill (Section 2.2.6).

2.2.1 Hive

Facebook's Hive [53] was one of the first Distributed SQL Query Engines built on top of the Hadoop platform [6]. It provides a Hive Meta Store service to put a relational database like structure on top of the raw data stored in HDFS. Whenever a HiveQL (SQL dialect) query is submitted to Hive, Hive will convert it to a Hadoop MapReduce job to be run on Hadoop MapReduce. Although Hive provides mid-query fault-tolerance, it relies on Hadoop MapReduce. Whenever queries get converted to multiple MapReduce tasks, they get slowed down by Hadoop MapReduce storing intermediate results on disk. The overall architecture is displayed in Figure 2.3.

2.2.2 Dremel

The rise of large MapReduce clusters allowed companies to process large amounts of data, but at a run time of several minutes to hours. Although Hive provides an SQL interface to the Hadoop cluster, it is still considered to be batch processing. Interactive analytics focuses on reducing the run time to seconds or minutes in order to allow analysts to explore subsets of data quickly and to get quick feedback

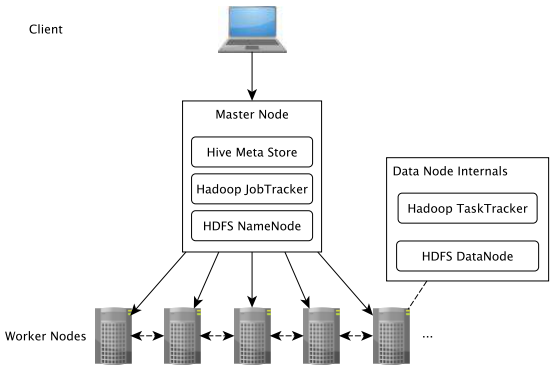


Figure 2.3: Hive Architecture

in applications where this is required (e.g. analysing crash reports, spam analysis, time series prediction, etc.). To achieve this speed up, Google's Dremel, one of the first interactive solutions, has incorporated three novel ideas which we will discuss briefly:

1. Instead of row-based storage relational databases typically use, Dremel proposes a columnar storage format which greatly improves performance for queries that only select a subset of columns from a table, e.g.:
   `SELECT a,COUNT(*) FROM table GROUP BY a WHERE b = 2;`
   Here, only columns `a` and `b` are read from disk, while other columns in this table are not read into memory.

2. An SQL dialect is implemented into Dremel, together with algorithms that can re-assemble records from the separately stored columns.

3. Execution trees typically used in web search systems, are employed to divide the query execution over a tree of servers (see Figure 2.4). In each layer the query is modified such that the union of all results in the next layer equals the result of the original query. The root node receives the query from the user, after which the query propagates down the tree, results are aggregated bottom-up.

Dremel was tested by its creators on read-only data nodes without column indices applied, using aggregation queries. Most of these queries were executed within 10 seconds, and the creators claim some queries had a scan throughput of roughly 100 billion records per second. For more results we refer to the official Dremel paper [43].

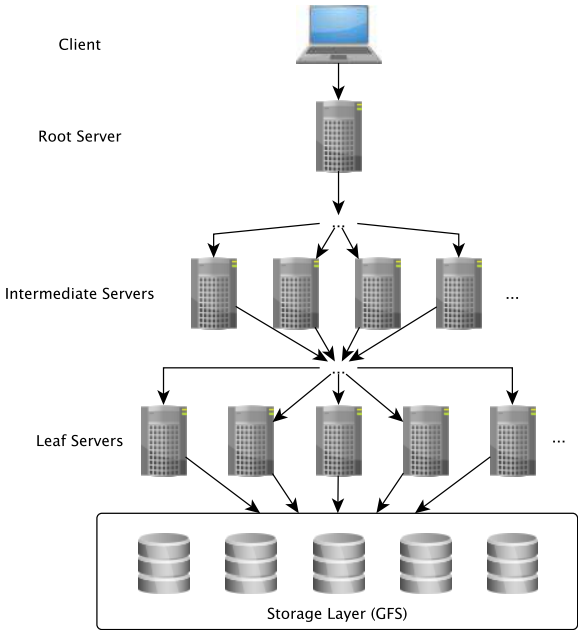


Figure 2.4: Execution tree of Dremel queries across servers (Adapted version of Figure 7 in [43]).

2.2.3 Impala

Impala [15] is a Distributed SQL Query Engine being developed by Cloudera and is heavily inspired by Google's Dremel. It employs its own massively parallel processing (MPP) architecture on top of HDFS instead of using Hadoop MapReduce as execution engine. One big downside of this engine is that it does not provide fault-tolerance. Whenever a node dies in the middle of query execution, the whole query is aborted. The high level architecture of Impala is depicted in Figure 2.5.

2.2.4 Shark

Shark [59] is a Distributed SQL Query Engine built on top of the Spark [63] execution engine, which in turn heavily relies on the concept of Resilient Distributed Datasets (RDDs) [62]. In short this means that whenever Shark receives an SQL query, it will convert it to a Spark job, executing it in Spark, and then returning the results. Spark keeps all intermediate results in memory using RDDs, and only spills them to disk if no sufficient memory is available. Mid-query fault-tolerance is provided by Spark. It is also possible to have the input and output dataset cached entirely in memory. Below is a more extensive explanation to RDDs, Shark and Spark.

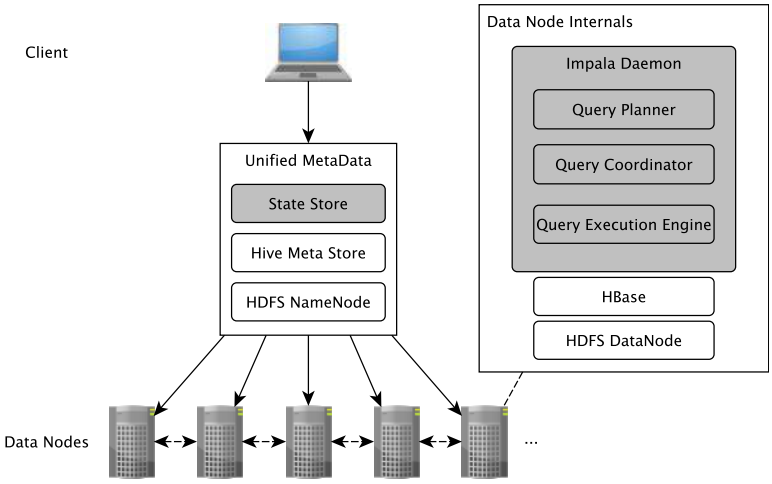


Figure 2.5: Impala High Level Architecture.

Resilient Distributed Datasets

An RDD is a fault-tolerant, read-only data structure on which in-memory transformations can be performed using the Spark execution engine. Examples of transformations are *map*, *filter*, *sort* and *join*, which all produce a transformed RDD as result. RDDs represent datasets loaded from external storage such as HDFS or Cassandra, and are distributed over multiple nodes. An example of an RDD would be an in-memory representation of an HDFS file, with each node containing a partition of the RDD of size equal to the block size of HDFS.

Shark

Shark builds on Spark, inheriting all features Spark has. It provides HiveQL compatibility, columnar in-memory storage (similar to Dremel's), Partial DAG execution, data co-partitioning and much more (see [59]). Shark allows the user to compose an RDD using HiveQL and perform transformations using the interface to Spark. The architecture of Shark (including Spark) is depicted in Figure 2.6. The master node has similar responsibilities as the root server in Dremel. The slave nodes all have HDFS storage, a shared memory pool for storing RDDs, and the Spark execution engine on top.

Spark Execution Engine

Spark is an execution engine which can make heavy use of keeping input and output data in memory. Therefore it is very suitable for iterative algorithms as, for example, the algorithm in the graph processing library GraphX [58], and the machine learning library MLLib [18], which are both also built on top of Spark, just like Shark.

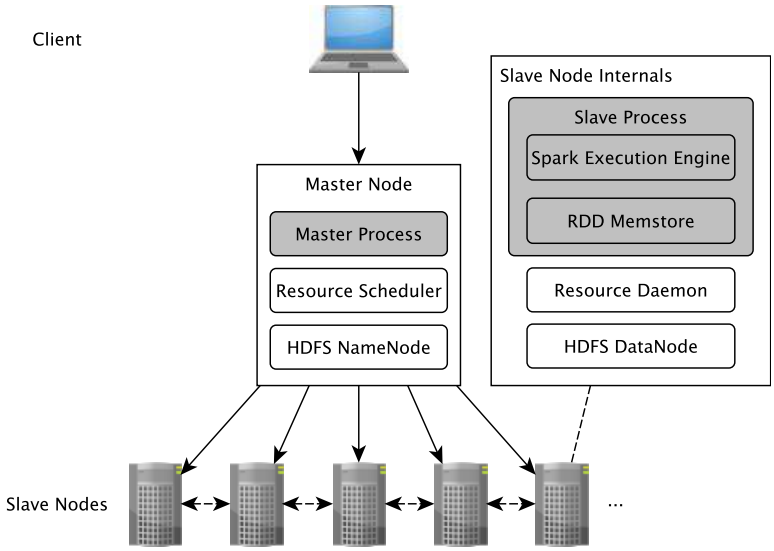


Figure 2.6: Shark Architecture (Adapted version of Figure 2 in [59]).

2.2.5 Presto

Like Hive, Presto was engineered at Facebook. Presto is very similar to the other systems, and although Presto is used by Facebook and other companies in production systems, it only recently started supporting writing query results into an output table. It is still missing a lot of features compared to the other systems on the market. Presto both supports HDFS and Cassandra [5] as storage backend. The global architecture is depicted in Figure 2.7.

2.2.6 Drill

Drill is in its very early stages of development and tries to provide an open source implementation of Dremel with additional features [35]. The main difference between Drill and the other state-of-the-art platforms is that Drill supports multiple (schemaless) data sources and provides multiple interfaces to them including an SQL 2003 compliant interface to them, instead of an SQL dialect. The high level architecture is depicted in figure 2.8. As you can see there is no root server or master node, but the client can send its query to any of the Drill Bits, which in turn converts the query to a data source compatible version.

No performance tests have been performed on Drill yet, since no stable implementation is available at the moment of writing (Only an alpha version which demonstrates limited query plan building and execution on a single node cluster).

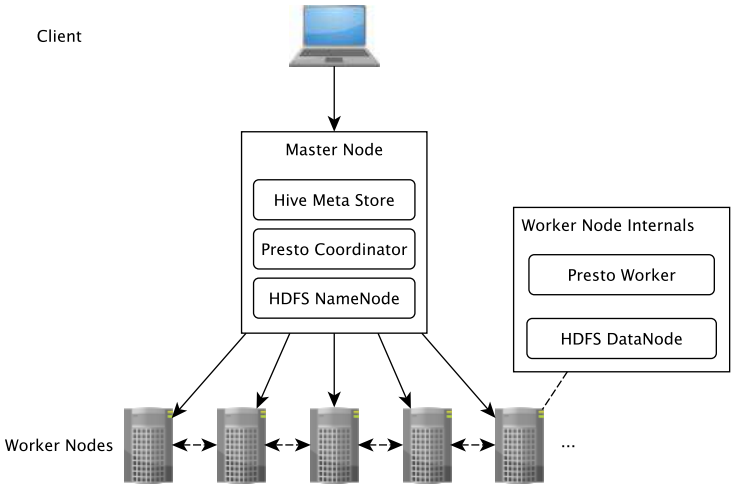


Figure 2.7: Presto Architecture

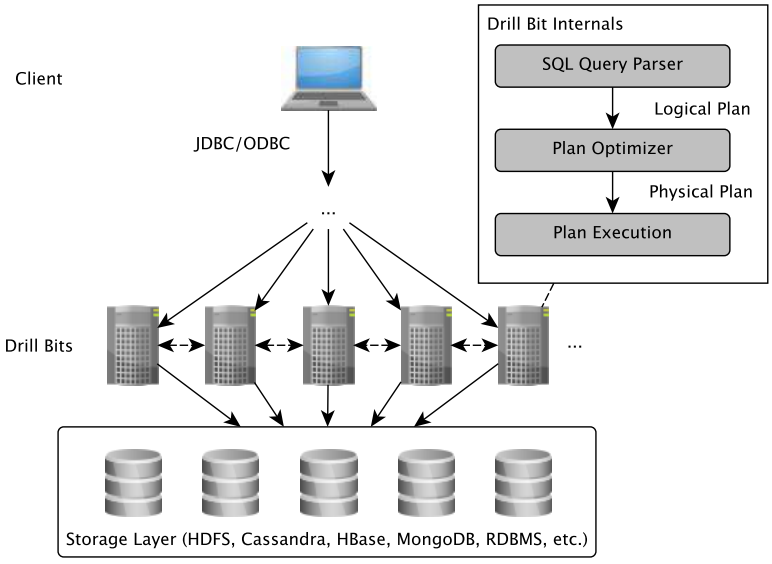


Figure 2.8: Drill High Level Architecture.

2.3 Related Distributed SQL Query Engine Performance Studies

We wanted to evaluate the major Distributed SQL Query Engines currently on the market using a cluster size and dataset size that is representative for SMEs, but still comparable to similar studies. Table 2.4 summarizes the related previous works. Some of them run a subset or enhanced version of the TPC-DS benchmark

Table 2.4: Overview of Related Work. Legend: Real World (R), Synthetic (S), Modified Workload (+)

Query Engines	Workload	Dataset Type	Largest Dataset	Cluster Size
Hive, Shark [59]	Pavlo+, other	R, S	1.55 TiB	100
Redshift, Hive, Shark, Impala, Tez [4]	Pavlo+	S	127.5 GiB	5
Impala, Tez, Shark, Presto [16]	TPC-DS+	S	13.64 TiB	20
Teradata DBMS [34]	TPC-DS+	S	186.24 GiB	8
Hive, Impala, Tez [33]	TPC-DS/H+	S	220.72 GiB	20
DBMS-X, Vertica [47]	Pavlo	S	931.32 GiB	100
Our Work	**Pavlo+, other**	**R, S**	**523.66 GiB**	**5**

[48] which has only recently been adopted for Big Data analytics in the form of BigBench [34]. Other studies run a variant of the Pavlo et al. micro-benchmark [47] which is widely accepted in the field.

Overall, most studies use synthetic workloads, of which some are very large. Synthetic workloads do not necessarily characterise real world datasets very well. For our work we have also taken a real world dataset in use by an SME. Besides our work, only one other study uses real world datasets [59]. However, like most of the other studies, it only reports on query response times. Our work evaluates performance much more in-depth by reporting more metrics and evaluating more performance aspects including scalability and detailed resource utilization. We argue that scalability and resource utilization are also very important when deciding which query engine will be used by an SME.

2.4 Machine Learning Algorithms

Machine Learning is a field in Computer Science, where (in short) one tries to find patterns in data in order to detect anomalies or perform predictions [55]. Within Machine Learning there exists a plethora of approaches, which are each tailored to a specific type of problem. In this section we will explain the idea of supervised learning (Section 2.4.1), after which we introduce the supervised learning algorithms we evaluated: Linear Regression (Section 2.4.2) and Support Vector Regression (Section 2.4.3). Other machine learning techniques are outside the scope of this thesis.

2.4.1 Supervised Learning Overview

One of the major classes of machine learning algorithms is supervised learning. The algorithms in this class all try to infer a hypothesis function h from a dataset of observations of independent input variables x_1 to x_n (features) and their respective

values of (usually one) dependent output variable y. The dataset of observations of features with their corresponding output values is called a *labeled* dataset. After the function h has been trained on such a dataset, it can be used to predict the output values that correspond to a set of features observed but where we do not know the output value already - this is called an *unlabeled* dataset. The supervised learning class has two sub-classes, namely *classification* and *regression*, which we clarify below.

Classification

With classification the function h is called a classifier. It maps the different input variables to *discrete valued* output. Consider a classifier h that was trained on a dataset of cars registered in the Netherlands. For each car, many features were reported (number of seats, color, age, number of doors, engine capacity, etc.) together with the brand. Then the classifier contains a mapping from the observations of the features for all these different cars to the corresponding brand. Whenever the classifier h gets a description of a *never seen* car's features as input, it can predict what the brand of the car is with a certain accuracy.

Regression

In regression analysis the function h is called the regression function. It maps the different input variables to *real valued* output. An example of using this could be the prediction of the price of an apartment based on features like: garden size, orientation, amount of floors, number of neighbours, etc.

Training Procedure

An important part of supervised learning is inferring the function h from the labeled training data and testing its accuracy on some labeled test data. A common approach for supervised learning is as follows, as recommended by [27]:

1. Start out with a matrix X which contains all m observations of n features x_1 to x_n, and a vector y which contains the m observations of the to be predicted value (discrete in case of classification, numeric in case of regression). We want to infer a function h by training a machine learning algorithm A.

2. Clean the dataset by removing each feature vector x_i of which all observations are identical (constant). Remove all redundant features (keep only one feature of each feature combination that has a correlation higher than a certain threshold). This prevents the amplification effect of some features being highly correlated.

3. Normalize the feature matrix X to values between -1 and 1 or 0 and 1. This prevents features with large absolute values from automatically dominating the features with small absolute values.

4. (Optional) Use Principal Component Analysis (PCA; see Section 2.5) to cut down the number of features further at the cost of losing a bit in accuracy. PCA can also be used to visualize the cleaned dataset.

5. Randomize the order of the rows (observations) of X and split X in two distinct sets of which the corresponding output values in y are stratified (the output variable's values should have about the same distribution in both sets):

   Training Set $T_{training}$ Approximately 70-80% of the data. This set will be used to infer the function h.

   Test Set T_{test} Approximately 20-30% of the data. This set will be kept aside during the whole training procedure and is only used in determining the *generalization error*. This determines how well the algorithm A performs on unlabeled data that it has never seen before.

6. Use (stratified) $k-$fold cross validation on the training set $T_{training}$ (explained in the next paragraph) while trying all (reasonable) combinations of parameters to the machine learning algorithm A. This allows for selecting the parameters to the algorithm A which has the smallest possible *validation error*.

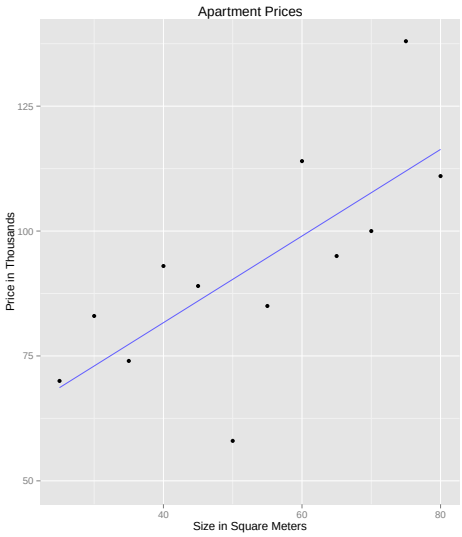
7. Train the machine learning algorithm A on the whole training set $T_{training}$ using the parameters found in the previous step. This gives infered function h.

8. Test the accuracy of the inferred function h (define the *generalization error*). For regression the metric used here is usually the Mean Squared Error (MSE), which is simply the mean of all squared differences in predicted values y' and actual values y.

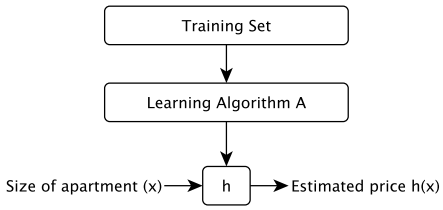
9. (Optional) Repeat step 4 to 7 i times to get insight how the accuracy changes when using a different data partitioning (step 4).

Cross Validation

When tuning the parameters for the machine learning algorithm A (see step 5 in the previous section), an exhaustive (or approximate) search is performed through all combinations of parameter values that yields the smallest error on a certain validation set. This test set cannot be equal to T_{test}, because then the inferred function h includes characteristics of T_{test} (the parameters are tuned to minimize error on T_{test}). In that case we can no longer use T_{test} for computing the generalization error. To circumvent this problem, an additional validation set is required. We can get this set of data by splitting $T_{training}$ or T_{test}, but the downside of this is that we have either less data to use for training in step 5 or less data to use for testing in step 6.



(a) Visualization of the model (the diagonal line).



(b) Diagram view of the training procedure.

Figure 2.9: Predicting apartment prices using simple linear regression.

This is where (k-fold) cross validation comes to the table. Instead of having a fixed validation set, we simply split $T_{training}$ into k stratified random folds. Of these folds, one fold will be used as validation set and the other $k-1$ folds will be used for training during parameter tuning. We make sure each fold will be used as validation set exactly once, and report the mean error among all cross folds as the validation error.

2.4.2 Linear Regression

Linear regression is one of the simplest concepts within supervised learning. Although many variants of this technique exist, we will discuss both the simplest (for illustrative purposes) and the one we evaluated in Chapter 4. The notation and knowledge presented in this section is partly adapted from [45].

Simple Linear Regression

In simple linear regression we try to fit a linear model that maps values of the predictor variable x to values of the outcome variable y. In this case we only deal with one predictor variable (feature) x to predict one outcome variable y's value. A good example of this would be predicting apartment prices based on their total size in square meters (see Figure 2.9a). Here, the size of the apartment would be the predictor x, and the price of the apartment the outcome y.

We train the model using the training procedure described in Section 2.4.1 and using m training examples as training set. This gives us a model h as the example shown in Figure 2.9b. With this model we can predict the price of an apartment given its size in square meters. For simple linear regression the model function is defined as:

$$h_\theta(x) = \theta_0 + \theta_1 x, \tag{2.1}$$

where the values of θ_0 and θ_1 are chosen such that $h_\theta(x)$ is as close to the real observed values of y in the training set. This can be defined as follows:

$$\theta = \arg\min_{\theta_0\theta_1} J(\theta_0, \theta_1), \tag{2.2}$$

where the cost function $J(\theta_0, \theta_1)$ is defined as the mean squared error (MSE) on the training set:

$$J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2, \tag{2.3}$$

which iterates over all m training example pairs $(x^{(i)}, y^{(i)})$ in the training set. In practice, finding the optimal value for θ is done using optimization methods like gradient descent [26, 41].

Multiple Linear Regression

In Multiple Linear Regression we try to fit a linear model to values of multiple predictor variables x_1 to x_n (the features) to predict one outcome variable y's value. The principle of this regression method is the same as simple linear regression, except that the outcome variable y is now determined by multiple features rather than a single feature. An example of this would be to include other features of apartments like the number of bedrooms in order to predict the price.

The model function of Multiple Linear Regression is defined as:

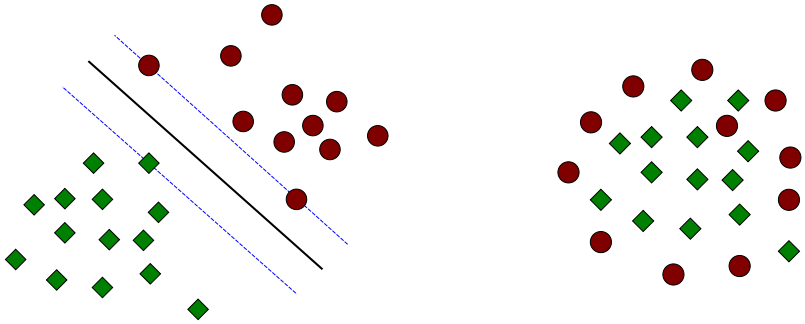
$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \theta^T x \quad \text{where } x_0 = 1, \tag{2.4}$$

where θ and x are 0-indexed vectors.

2.4.3 Support Vector Regression

Support Vector Regression [31] is the regression variant of Support Vector Machines for classification. Vapnik et al. proposed the SVM technique in [28], and many implementations are available (such as libSVM (Java and C++) [27]). We describe the main ideas behind SVMs rather than going into too much technical details (See [28] for a more in depth explanation).

An SVM is a binary classifier that tries to separate the dataset in two distinct classes using a hyper plane H. If the data is linearly separable (see Figure 2.10a



(a) Linearly Separable. (b) Not Linearly Separable.

Figure 2.10: Example of SVM method applied for classification. The training examples on the dotted lines form the Support Vectors. The black line separates the two classes of data. The area between the two dotted lines is called the margin.

for an example in two dimensional feature space), the SVM algorithm fits a hyper plane H between the two distinct classes such that the euclidean distance between H and the data points of the training set is maximized. The points closest to H in each class form the so called *Support Vectors*.

When the training data is not linearly separable (see Figure 2.10b, new dimensions to each training example are added (new features are created), using the so called *kernel trick*. A kernel function is applied to each of the points in the training set in order to map them to a higher dimensional feature space that might be linearly separable. Different functions can be used as kernel.

When more than two classes need to be distinguished in the data one can simply train k SVM classifiers. Each classifier then distinguishes one single class from all the other classes.

2.5 Principal Component Analysis

Principal Component Analysis (PCA) is a procedure to reduce the number of features in a dataset into principal components. These principal components are simply a linear combination of the original features in the dataset, such that the first principal component (PC1) accounts for the highest variance in the original dataset, the second principal component (PC2) accounts for the second highest variance and so on [38, 56]. PCA has two applications, as explained below:

1) Visualization: If you want to get insight into a dataset that has a very large number of features, PCA can help to reduce this large number to just a hand full of

derived principal components. As long as the first two principal components cover the majority of the variance of the original dataset, they can be used to get an initial feeling of the dataset by plotting them. Looking at how much each original feature contributes to the first couple of principal components also gives an idea which of the original features are likely to be of importance for the machine learning model.

2) Speed-up the Training Process: The training speed of many machine learning algorithms depends on the number of features that are used for training. Reducing this number can significantly speed up the training process. Because PCA makes its principal components take account for the variance in the original dataset in monotonous decreasing order (PC1 covers the most), we can drop all principal components after we have covered at least some amount of variance (e.g., 99%). We could for example only use the first five principal components as features for machine learning training, instead of the possibly thousands of original features, at the cost of losing only a tiny bit in accuracy.

Chapter 3

Performance Evaluation of Distributed SQL Query Engines

In this chapter we evaluate the state-of-the-art Distributed SQL Query Engines that have been described in Section 2.2 in order to answer **RQ1**: *What is the performance of state-of-the-art Distributed SQL Query Engines in a single-tenant environment?* This chapter is the basis of our work in [57]. We describe our query engine selection in Section 3.1. Then we define a method of evaluating the query engine performance in Section 3.2, after which we define the exact experiment setup in Section 3.3. The results of our experiments are in Section 3.4, and are summarised in Section 3.5.

3.1 Query Engine Selection

In this study we initially attempted to evaluate 5 state-of-the-art Distributed SQL Query engines: Drill, Presto, Shark, Impala and Hive, which we describe in Section 2.2. However, we ended discarding Drill and Presto because these systems lacked required functionality at the time of testing. Drill only had a proof-of-concept one-node version, and Presto did not have the functionality needed to write output to disk (which is required for the kind of workloads we wanted to evaluate).

3.2 Experimental Method

In this section we present the method of evaluating the performance of Distributed SQL Query Engines. First we define the workload as well as the aspects of the engines used for assessing this performance. Then we describe the evaluation procedure.

Table 3.1: Summary of Datasets.

Table	# Columns	Description
uservisits	9	Structured server logs per page.
rankings	3	Page rank score per page.
hotel_prices	8	Daily hotel prices.

3.2.1 Workload

During the performance evaluation we use both synthetic and real world datasets with three SQL queries per dataset. We carefully selected the different types of queries and datasets to match the scale and diversity of the workloads SMEs deal with.

1) *Synthetic Dataset*: Based on the benchmark from Pavlo et al. [47], the UC Berkeley AMPLab introduced a general benchmark for DSQEs [4]. We have used an adapted version of AMPLab's Big Data benchmark where we leave out the query testing User Defined Functions (UDFs), since not all query engines support UDF in similar form. The synthethic dataset used by these 3 queries consists of 118.29 GiB of structured server logs per URL (the uservisits table), and 5.19 GiB of page ranks (the rankings table) per website, as seen in Table 3.1.

Is this dataset representative for SME data? The structure of the data closely resembles the structure of click data being collected in all kinds of SMEs. The dataset size might even be slightly large for SMEs, because as pointed out by Rowstron et al. [51] analytics production clusters at *large* companies such as Microsoft and Yahoo have median job input sizes under 13.03 GiB and 90% of jobs on Facebook clusters have input sizes under 93.13 GiB.

On this dataset, we run queries 1 to 3 to test raw data processing power, aggregation and *JOIN* performance respectively. We describe each of these queries below in addition to providing query statistics in Table 3.2.

Query 1 performs a data scan on a relatively small dataset. It simply scans the whole rankings table and filters out certain records.

Query 2 computes the sum of ad revenues generated per visitor from the uservisits table in order to test aggregation performance.

Query 3 joins the rankings table with the uservisits table in order to test JOIN performance.

2) *Real World Dataset*: We collected price data of hotel rooms on a daily basis during a period of twelve months between November 2012 and November 2013. More than 21 million hotel room prices for more than 4 million hotels were collected on average every day. This uncompressed dataset (the hotel_prices table) is 523.66 GiB on disk as seen in Table 3.2. Since the price data was collected every day, we decided to partition the dataset in daily chunks as to be able to only use

Table 3.2: Summary of SQL Queries.

Query	Input Size		Output Size		Tables
	GiB	*Records*	*GiB*	*Records*	
1	5.19	90M	5.19	90M	`rankings`
2	118.29	752M	40	254M	`uservisits`
3	123.48	842M	$< 10^{-7}$	1	`uservisits`, `rankings`
4	523.66	7900M	$< 10^{-2}$	113K	`hotel_prices`
5	20	228M	4.3	49M	`hotel_prices` subsets
6	8	94.7M	4	48M	`hotel_prices` subsets

data of certain collection days, rather than having to load the entire dataset all the time.

Is this dataset representative for SME data? The queries we selected for the dataset are in use by Azavista, an SME specialized in meeting and event planning software. The real world scenarios for these queries relate to reporting price statistics per city and country.

On this dataset, we run queries 4 to 6 to also (like queries 1 to 3) test raw data processing power, aggregation and *JOIN* performance respectively. However, these queries are not interchangeable with queries 1 to 3 because they are tailored to the exact structure of the hotel price dataset, and by using different input and output sizes we test different aspects of the query engines. We describe each of the queries 4 to 6 below in addition to providing query statistics in Table 3.2.

Query 4 computes average prices of hotel rooms grouped by certain months.

Query 5 computes linear regression pricing curves over a timespan of data collection dates.

Query 6 computes changes in hotel room prices between two collection dates.

3) Total Workload: Combining the results from the experiments with the two datasets gives us insights in performance of the query engines on both synthetic and real world data. In particular we look at how the engines deal with data scans (Query 1 and 4), heavy aggregation (Query 2 and 5), and the JOINs (Query 3 and 6).

3.2.2 Performance Aspects and Metrics

In order to be able to reason about the performance differences between different query engines, the different aspects contributing to this performance need to be defined. In this study we focus on three performance aspects:

1. *Processing Power*: the ability of a query engine to process a large number of SQL queries in a set amount of time. The more SQL queries a query engine can handle in a set amount of time, the better. We measure the processing power in terms of response time, that is, the time between submitting an SQL query to the system and getting a response. In addition, we also calculate the throughput per SQL query: the number of input records divided by response time.

2. *Resource Utilization*: the ability of a query engine to efficiently use the system resources available. This is important, because especially SMEs cannot afford to waste precious system resources. We measure the resource utilization in terms of mean, maximum and total CPU, memory, disk and network usage.

3. *Scalability*: the ability of a query engine to maintain predictable performance behaviour when system resources are added or removed from the system, or when input datasets grow or shrink. We perform both horizontal scalability and data input size scalability tests to measure the scalability of the query engines. Ideally, the performance should improve (at least) linearly with the amount of resources added, and should only degrade linearly with every unit of input data added. In practice this highly depends on the type of resources added as well as the complexity of the queries and the overhead of parallelism introduced.

3.2.3 Evaluation Procedure

Our procedure for evaluating the DSQEs is as follows: we run each query 10 times on its corresponding dataset while taking snapshots of the resource utilization using the monitoring tool `collectl` [9]. After the query completes, we also store its response time. When averaging over all the experiment iterations, we report the standard deviation as indicated with error bars in the experimental result figures. Like that, we take into account the varying performance of our cluster at different times of the day, intrinsic to the cloud [37].

The queries are submitted on the master node using the command line tools each query engine provides, and we write the output to a dedicated table which is cleared after every experiment iteration. We restart the query engine under test at the start of every experiment iteration in order to keep it comparable with other iterations.

3.3 Experimental Setup

We define a full micro-benchmarking setup by configuring the query engines as well as tuning their data caching policies for optimal performance. We evaluate the most recent stable versions of Shark (v0.9.0), Impala (v1.2.3) and Hive (v.0.12). Many different parameters can influence the query engine's performance. In the

following we define the hardware and software configuration parameters used in our experiments.

Hardware: To make a fair performance comparison between the query engines, we use the same cluster setup for each when running the experiments. The cluster consists of 5 `m2.4xlarge` worker Amazon EC2 VMs and 1 `m2.4xlarge` master VM, each having 68.4 GiB of memory, 8 virtual cores and 1.5 TiB instance storage. This cluster has sufficient storage for the real-world and synthetic data, and also has the memory required to allow query engines to benefit from in-memory caching of query inputs or outputs. Contrary to other Big Data processing systems, DSQEs (especially Impala and Shark) are tuned for nodes with large amounts of memory, which allows us to use fewer nodes than in comparable studies for batch processing systems to still get comparable (or better!) performance. An additional benefit of this specific cluster setup is the fact it is the same cluster setup used in the AMPLab benchmarks previously performed on older versions of Shark (v0.8.1), Impala (v1.2.3) and Hive (v0.12) [4]. By using the same setup, we can also compare current versions of these query engines with these older versions and see if significant performance improvements have been made.

Software: Hive uses YARN [54] as resource manager while we have used Impala's and Shark's standalone resource managers respectively, because at the time of testing the YARN compatible versions were not mature yet. All query engines under test run on top of a 64-bit Ubuntu 12.04 operating system. Since the queries we run compute results over large amounts of data, the configuration parameters of the distributed file system this data is stored on (HDFS) are crucial. It is therefore imperative that we keep these parameters fixed across all query engines under test. One of these parameters includes the HDFS block size, which we keep to the default of 64 MB. The number of HDFS files used per dataset, and how these files are structured and compressed is also kept fixed. While more sophisticated file formats are available (such as RCFile [36]) we selected the Sequence file key-value pair format because unlike the more sophisticated formats this is supported by all query engines, and this format uses less disk space than the plain text format. The datasets are compressed on disk using the Snappy compression type, which aims for reasonable compression size while being very fast at decompression.

Each worker has 68.4 GiB of memory available of which we allow a maximum of 60GiB for the query engines under test. This leaves a minimum of 8 GiB of free memory for other processes running on the same system. By doing this we ensure that all query engines under test have an equal amount of maximum memory reserved for them while still allowing the OS disk buffer cache to use more than 8 GiB when the query engine is not using a lot of memory.

Dataset Caching: Another important factor that influences query engine performance is whether the input data is cached or not. By default the operating system will cache files that were loaded from disk in an OS disk buffer cache. Because both Hive and Impala do not have any configurable caching policies available, we will simply run the queries on these two query engines both with and without the input dataset loaded into the OS disk buffer cache. To accomplish this, we perform

Table 3.3: Different ways to configure Shark with caching.

Abbreviation	OS Disk Cache	Input Cache	Output Cache
Cold	No	No	No
OC	No	No	Yes
OSC	Yes	No	No
IC	N/A	Yes	No
OSC+OC	Yes	No	Yes
IC+OC	N/A	Yes	Yes

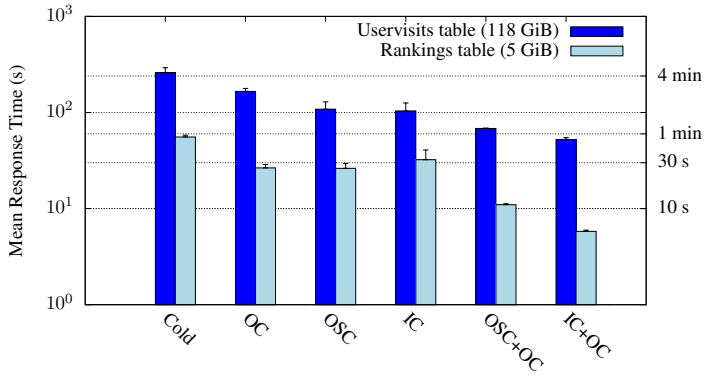


Figure 3.1: Response time for different Shark caching configurations (vertical axis is in log-scale).

a `SELECT` query over the relevant tables, so all the relevant data is loaded into the OS disk buffer cache. The query engines under test are restarted after every query as to prevent any other kind of caching to happen that might be unknown to us (e.g., Impala has a non-configurable internal caching system).

In contrast, Shark has more options available regarding caching. In addition to just using the OS disk buffer caching method, Shark also has the option to use an in-memory cached table as input and an in-memory cached table as output. This completely removes the (need for) disk I/O once the system has warmed up. To establish a representative configuration for Shark, we first evaluate the configurations as depicted in Table 3.3. OS Disk Cache means the entire input tables are first loaded through the OS disk cache by means of a `SELECT`. Input Cache means the input is first cached into in-memory Spark RDDs. Lastly, Output Cache means the result is kept in memory rather than written back to disk.

Figure 3.1 shows the resulting average response times for running a simple `SELECT * ` query using the different possible Shark configurations. Note that no distinction is made between OS disk buffer cache being cleared or not when a cached input table is used, since in this case Shark does not read from disk at all.

The configuration with both input and output cached tables enabled (IC+OC) is the fastest setup for both the small and large data set. But the IC+OC and the IC

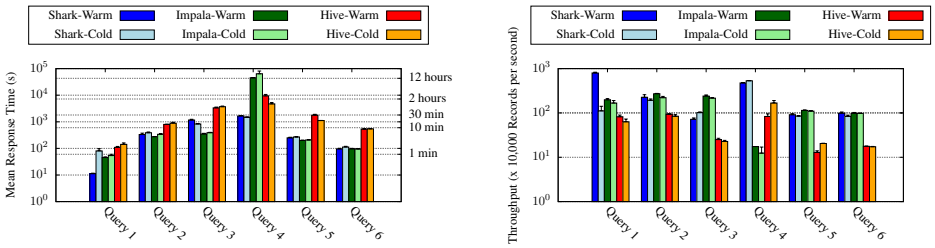


Figure 3.2: Query Response Time (left) and Throughput (right). Vertical axis is in log-scale.

configuration can only be used when the entire input data set fits in memory, which is often not the case with data sets of multiple TBs in size. The second fastest configuration (OSC+OC) only keeps the output (which is often much smaller) in memory and still reads the input from disk. The configuration which yields the worst results is using no caching at all (as expected).

In the experiments in Section 3.4 we use the optimistic IC+OC configuration when the input data set fits in memory and the OSC+OC configuration when it does not, representing the best-case scenario. In addition the Cold configuration will be used to represent worst-case scenarios.

3.4 Experimental Results

We evaluate the three query engines selected in Section 3.1 on the performance aspects described in Section 3.2.2 using the workloads described in Section 3.2.1. We evaluate processing power in Section 3.4.1, resource consumption in Section 3.4.2, and scalability in Section 3.4.4.

3.4.1 Processing Power

We have used the fixed cluster setup with a total of 5 worker nodes and 1 master node as described in Section 3.3 to evaluate the response time and throughput (defined as the number of input records divided by the response time) of Hive, Impala and Shark on the 6 queries in the workloads. The results of the experiments are depicted in Figure 3.2. All experiments have been performed 10 times except for Query 4 with Impala since it took simply too long to complete. Only 2 iterations have been performed for this particular query. We used the dataset caching configurations explained in Section 3.3. For Impala and Hive we used disk buffer caching and no disk buffer caching for the warm and cold situations, respectively. For Shark we used the *Cold* configuration for the cold situation. In addition we used input and output dataset caching (*IC+OC*) for the warm situation of queries 1 to 3, and disk buffer caching and output caching (*OSC+OC*) for the warm situation of queries 4 to 6, since the price input dataset does not entirely fit in memory.

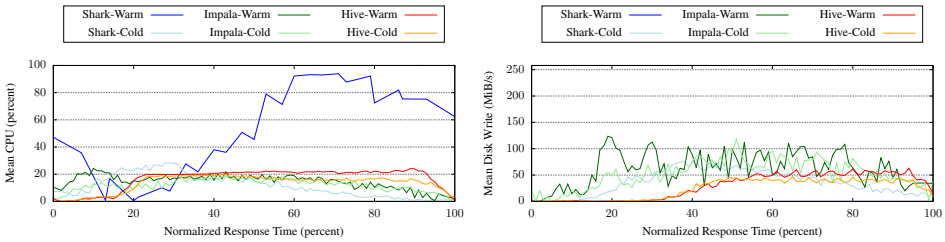


Figure 3.3: CPU utilization (left) and Disk Write (right) for query 1 over normalized response time.

Key Findings:

- Input data caching generally does not cause a significant difference in response times.
- Performance is relatively stable over different iterations.
- Impala and Shark have similar performance and Hive is the worst performer in most cases. There is no overall winner.
- Impala does not handle large input sizes very well (Query 4).

The main reason why Hive is much slower than Impala and Shark is because of the high intermediate disk I/O. Because most queries are not disk I/O bound, data input caching makes little difference in performance. We elaborate on these two findings in more detail in Section 3.4.3.

In the following we discuss the response times from the 6 queries in a pair-wise manner. We evaluate the data scan queries 1 and 4, the aggregation queries 2 and 5, and the JOIN performance queries 4 and 6 depicted in Figure 3.2.

1) Scan performance: Shark's response time for query 1 with data input and output caching enabled is significantly better than that of the other query engines. This is explained by the fact that query 1 is CPU-bound for the *Shark-Warm* configuration, but disk I/O bound for all other configurations as depicted in Figure 3.3. Since *Shark-Warm* caches both the input and output, and the intermediate data is so small that no spilling is required, no disk I/O is performed at all for *Shark-Warm*.

Results for query 4 for Impala are particularly interesting. Impala's response time is 6 times as high as Hive's, while resource utilization is much lower, as explained in Section 3.4.3. No bottleneck can be detected in the resource utilization logs and no errors are reported by Impala. After re-running the experiments for Impala on query 4 on a different set of Amazon EC2 instances, similar results are obtained, which makes it highly unlikely an error occured during experiment execution. A more in-depth inspection is needed to get to the cause of this problem, which is out of the scope of our work.

2) Aggregation performance: Both the aggregation query 2 and 5 are handled quite well by all the engines. The main reason why even though query 5 has a much smaller input dataset, the response times are close to the ones of query 2 is that this query is relatively much more compute intensive (see Figure 3.4).

3) JOIN performance: The query engines perform quite similar on the JOIN queries 3 and 6. A remarkable result is that the fully input and output cached configuration *Shark-Warm* starts to perform worse than its cold equivalent when dataset sizes grow. This is explained in more detail in Section 3.4.4.

3.4.2 Resource Consumption

Although the cluster consists of both a master and 5 worker nodes, we only evaluate the resource consumption on the workers, since the master is only used for coordination and remains idle the most of the time. For any of the queries the master never used more than 6 GiB of memory (<10% of total available), never exceeded more than 82 CPU Core Seconds (<0.0005% of the workers' maximum), has negligible disk I/O, and never exceeded total network I/O of 4 GiB (<0.08% of the workers' maximum).

Key Findings:

- Impala is a winner in total CPU consumption. Even when Shark outperforms Impala in terms of response time, Impala is still more CPU efficient (Figure 3.4).
- All query engines tend to use the full memory assigned to them.
- Disk I/O is as expected significantly higher for the queries without data caching vs. the queries with data caching. Impala has slightly less disk I/O than Shark. Hive ends last (Figure 3.5).
- Network I/O is comparable in all query engines, with the exception of Hive, which again ends last (Figure 3.5).

In the following we discuss the resource consumption per query averaged over 5 workers and 10 iterations (50 datapoints per average). We show the CPU Core Seconds per query in Figure 3.4. This shows how much total CPU a query uses during the query execution. The CPU Core Seconds metric is calculated by taking the area under the CPU utilization graphs of all workers, and then multiplying this number by the number of cores per worker (8 in our setup). For example, a hypothetical query running 2 seconds using 50% of CPU uses 1 CPU Second which is equal to 8 CPU Core Seconds in our case. A query running on Impala is the most efficient in terms of total CPU utilization, followed by Shark. This is as expected since although Shark and Impala are quite close in terms of response time, Impala is written in C/C++ instead of Scala, which is slightly more efficient.

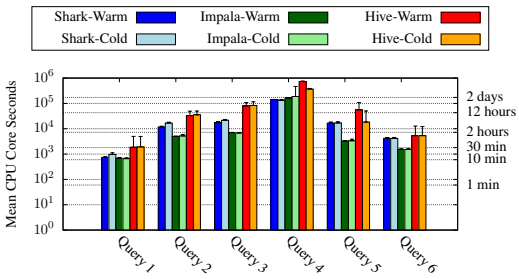


Figure 3.4: CPU Core Seconds. Vertical axis is in log-scale.

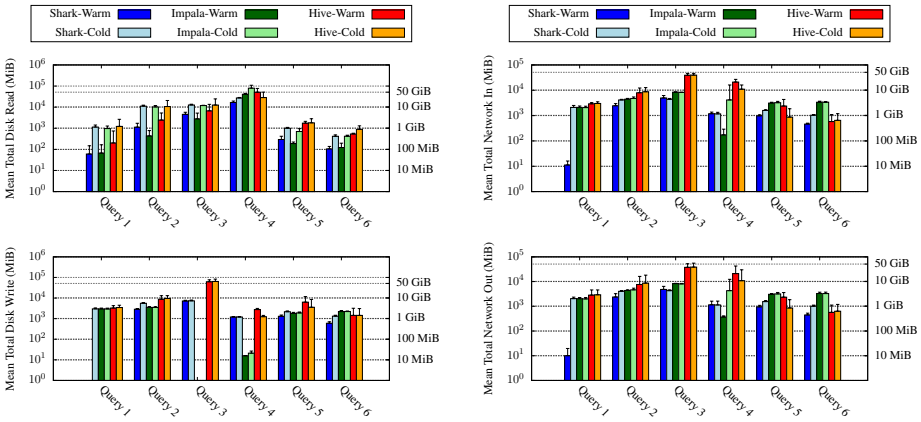


Figure 3.5: Total Disk Read (left-top), Disk Write (left-bottom), Network In (right-top) and Network Out (right-bottom). Vertical axis is in log-scale.

The total disk I/O per query is depicted in Figure 3.5. It shows that data caching does make a significant difference in the number of disk reads performed, but as shown in Section 3.4.3, this is hardly ever the bottleneck. Although the input datasets are entirely cached in the *Shark-Warm* configuration for queries 1 to 3, disk reading still occurs. This can be explained by the fact that Shark writes shuffle output of intermediate stages to the disk buffer cache (which eventually spills to disk). For queries 4 to 6 less significant differences occur since *Shark-Warm* only uses the OS disk buffer cache mechanism, like Impala and Hive. Note that because the input and output are compressed (compression ratio around 10), generally no more than 10% of the datasets is read or written to disk. Query 1 and 3 have very small output datasets, which makes *Shark-Warm*'s output not visible in the figure for query 1. Similarly for query 3 Impala does not show up at all because Impala does not write intermediate shuffle data to disk.

Figure 3.5 (right) shows the network I/O per query. Since most network I/O occurs between the workers, the network in and out look similar. Hive has a very variable network output total. Exact resource consumption numbers can be found

in the tables of Appendix A.

3.4.3 Resource Utilization over Time

In addition to reporting the average resources used per query, we show a more in depth analysis on the resource utilization over time for each query. Since queries have different response times each iteration, we normalize the response time to a range between 0 and 100% of query completion, and we show the mean values of each resource for all worker nodes averaged over all iterations (5 worker nodes, 10 iterations equals 50 data points per average total). We do not show network in performance in the figures since this is very similar to the network out, and we omit memory utilizations since for the majority of query engines the memory does not get deallocated until after the query execution has been completed. All the resulting figures are in Appendix B for brevity.

Key Findings:

- While the scan queries were expected to be disk I/O bound, query 1 is generally I/O bound, and query 4 is CPU bound because it also performs complex aggregation (Figure B.1 and B.4).
- The aggregation queries are both CPU bound (Figure B.2 and B.5).
- The JOIN queries are network I/O bound as expected (Figure B.3 and B.6).

In the following we go over each of the queries in detail in pair wise manner.

1) Scan performance: As explained in Section 3.4.1 query 1 is bound by disk and network I/O with the exception of the *Shark-Warm* configuration because the dataset is so small and entirely cached in memory (see Figure B.1). As seen in Figure B.4, query 4 is so computation heavy due to the additional aggregations performed, that it can suffice with a very low Disk read per second to keep up with the calculations. Because of the same reason the network out is low. And since the output size is so small, little to no disk writes are performed.

2) Aggregation performance: Both query 2 and 5 are CPU bound as shown in Figure B.2 and B.5 respectively. Both the queries show typical behaviour with higher CPU load at the start of query execution and higher disk I/O at the end.

3) JOIN performance: As expected both query 3 and 6 are network I/O heavy (see Figure B.3 and B.6 respectively). This is because one of the tables being joined is distributed over the network of workers.

3.4.4 Scalability

In this section we analyze both the horizontal and the data size scalability of the query engines. We decided to scale horizontally down instead of up because a cluster of 5 nodes of this caliber is already quite expensive for SMEs (more than

$7000 a month), and from our experimental results it shows that some queries already do no longer scale well from 4 to 5 worker nodes. We used queries 1 to 3 for data size scaling (since this dataset was already synthetic in the first place) and queries 4 to 6 for horizontal scaling.

Key Findings:

- Both Impala and Shark have good data size scalability on the scan and aggregation queries, whereas the response time has super-linear growth on the JOIN queries as the input size increases. This is as expected, since a JOIN is an operation that requires super-linear time.
- Hive has very good data scalability on all queries, but this is likely due to its large overhead, which dominates the response time at these data input sizes.
- If *Shark-Warm*'s input dataset is too large for its data storage memory, the response time will increase beyond *Shark-Cold* due to swapping.
- Shark and Impala horizontally scale reasonably well on all types of queries up to 3 nodes, whereas Hive only scales well on queries with large input sizes.
- The query engines do not benefit from having more than 3 or 4 nodes in the cluster. Impala even performs worse for query 6 at bigger cluster sizes.

1) Data Size Scalability: The data size scalability of the query engines is depicted in Figure 3.6 (left). We have sampled subsets from the original dataset of the following sizes: 5%, 10%, 25%, 50%, 75%, 90%, and 100%. We display these along the horizontal axis of the figure in terms of how many times the samples are bigger compared to the 5% sample (1 equals the 5% sample, 20 equals the 100% sample). The vertical axis displays how many times the response time is worse compared to the 5% sample. The colored lines correspond to each of the query engine configurations, whereas the dashed black line depicts the situation where the response time degrades just as fast as the data input size grows. Any query engine performing above this dashed line does not scale very well with the data input size. Query engines performing close to or below the dashed line do scale very well with the input size. Engines that have a much more gentle slope in their performance compared to the dashed line, have their system overhead dominate the response time.

For query 1, all query engines have good data size scalability, but both Shark and Hive have their system overhead dominating the response time. This is because query 1 has a very small input dataset of only 5 GiB. So scalability is not easily shown. For query 2, which has much larger data input size, it shows that both Impala and Shark scale very well. Hive's response time on the other hand, is still being dominated by its system overhead. The query engines have super-linear scalability on query 3, except for Hive, which shows close to linear scalability.

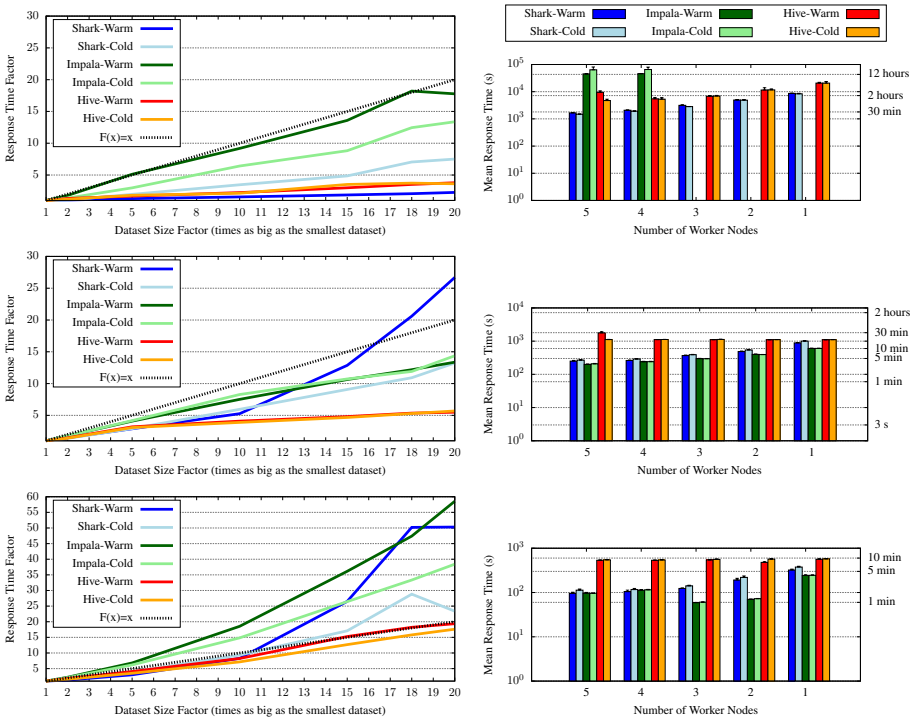


Figure 3.6: Data size scalability: Response time Query 1, 2 and 3 (left; from top to bottom). Horizontal scalability: Response time Query 4, 5 and 6 (right; from top to bottom) vertical axis in log-scale.

However, since a JOIN operation takes super-linear time, the response time is expected to grow faster than linearly.

An interesting phenomenom occurs with Shark for query 2 and 3. When the data input size grows and passes 50% of the size of the original dataset (10 on the horizontal axis in the figure), *Shark-Warm* actually starts scaling worse than *Shark-Cold*. This is caused by the fact that Shark allocates only around 34 GiB of the 60 GiB it was assigned for data storage and uses the remaining amount as JVM overhead. This means that the total cluster can only store about 170 GiB of data instead of the 300 GiB it was assigned. The input dataset for query 2 and 3 are close to 120 GiB, which fills some of the worker nodes' fully at the start of query execution. When data exchange occurs between the workers, even more memory is needed for the shuffled data received from the other worker nodes, causing the node to spill some of its input data back to disk.

2) Horizontal Scalability: The horizontal scalability of the query engines is depicted in Figure 3.6 (right) (note that we are scaling down instead of up). For Impala we only ran 4 and 5 nodes since it already took 12 hours to complete. Both Shark and Hive scale near linearly on the number of nodes. Hive only scales well on query 4 since Hive's Hadoop MapReduce overhead likely outweighs the computation time for query 5 and 6. This is because they have a relatively small input size. Impala actually starts to perform worse on query 6 if more than 3 nodes are added to the cluster. Similarly, both Shark and Impala no longer improve performance after more than 4 nodes are added to the cluster for query 4 and 5.

This remarkable result for horizontal scaling shows that whenever a query is not CPU-bound on a cluster with some number of nodes, performance will not improve any further when adding even more nodes. In the case of network I/O bound queries like query 6, it might even be more beneficial to bind these to a smaller number of nodes so less network overhead occurs.

3.5 Summary

In this chapter we have defined an empirical evaluation method to assess the performance of different query engines. Despite not covering all the methodological aspects of a scientific benchmark, this micro-benchmark gives practical insights for SMEs to take informed decisions when selecting a specific tool. Moreover, it can be used to compare current and future engines. The method focuses on three performance aspects: processing power, resource utilization, and scalability.

Using both a real world and a synthetic dataset with representative queries, we evaluate the query engines' performance. We find that different query engines have widely varying performance. Although Hive is almost always outperformed by the other engines, it highly depends on the query type and input size whether Impala or whether Shark is the best performer. Shark can also perform well on queries with over 500 GiB in input size in our cluster setup, while Impala starts to perform worse for these queries. Overall Impala is the most CPU efficient, and all query

engines have comparable resource consumption for memory, disk and network. A remarkable result found is that query response time does not always improve when adding more nodes to the cluster. Our detailed key findings can be found with every experiment in Section 3.4.

By performing this performance study we have answered the first research question (**RQ1**): *What is the performance of state-of-the-art Distributed SQL Query Engines in a single-tenant environment?*

Chapter 4

Performance Evaluation of Query Time Predictors

In this chapter we evaluate the performance of three query time predictors in order to answer **RQ2:** *What is the performance of query time predictors that utilize machine learning techniques?* The predictors can predict query response time as well as query execution time. The response time of a query also includes the time it had to wait in the query execution queue, whereas execution time does not. Response time predictions can be used to report progress to the user, and execution time predictions can be used to base scheduling decisions on. Before conducting the performance evaluation, we illustrate the usefulness of query time predictors by presenting the design of Perkin. This is a scheduling component for Shark, which uses predictors in its scheduling policies.

We first describe the predictors that we selected for our evaluation in Section 4.1. We then present the design of Perkin in Section 4.2. We describe our evaluation method in Section 4.3, followed by the results of our experiments in Section 4.4. Our findings are summarized in Section 4.5.

4.1 Predictor Selection

We evaluate two regression methods for predicting query execution and response time, *Multiple Linear Regression (MLR)* and *Support Vector Regression (SVR)*, which have been explained in more detail in Section 2.4.2 and 2.4.3, respectively. We selected these methods because they are both widely used and successful in different problem domains. In addition to comparing their performance with each other, we also compare it with one rule-of-thumb method *Last2*. This method takes the average of query times of the last two queries of the same query type. It then uses this as the query time prediction for the query which needs its time predicted. This method was derived from the findings in [30]. We discard any cost-based analytical method for predicting query time, because these kind of methods require a lot of system specific details to work. Our cost-based analytical approach has

been described in Appendix C.

4.2 Perkin: Scheduler Design

In order to give context to the query time predictors we evaluate, we present Perkin, a scheduling component that utilizes time and slowdown predictions in order to minimize query response time. This scheduling component builds on Spark, the execution engine of Shark. We chose Shark (and thus Spark), because from our experience in Chapter 3, it was the most stable system with very good performance.

In Section 4.2.1 we describe a major use case scenario for Perkin. We then present its high level architecture and scheduling policies in respectively Section 4.2.2 and Section 4.2.3.

4.2.1 Use Case Scenario

We show a typical use case scenario of Perkin below. One example how Perkin can be used, is to run complex queries on behalf of the non-analyst user:

1. The user visits an analytical view (e.g., a part of the Analytics module in Azavista's Web-based software), and issues a typical question like: "What are the best cities to plan my event in September 2014, given my event budget of $10,000?" (Note that these kind of queries are not cacheable as static page, and datasets are generally too large to fit in a relational database).

2. The front-end system submits a pre-defined query to Perkin using the user's query parameters (e.g., the budget, the event date etc.), and setting a soft-deadline in seconds.

3. Perkin estimates the query *response* time using a query *response* time predictor, and the query *execution* time using a query *execution* time predictor. Recall that response time also includes queue wait time, whereas execution time does not.

4. Perkin places the query in the query execution queue. The query will get precedence over the other queries currently in the queue according to the currently active scheduling policy (see Section 4.2.3). For example, The SJF-S policy gives precedence to queries that have a lower predicted execution time.

5. The front-end will display a progress bar to the user based on the response time estimate.

6. Once Perkin finishes the query, it returns the result data to the front-end system.

7. The front-end system visualizes the result data for the user.

In this typical scenario there are many interactive queries being fired to the cluster in short amounts of time. Thus, it is important that the query engine can handle this.

4.2.2 Architecture

Since Perkin heavily relies on Shark and Spark internals, we first explain the inner workings of these two systems in detail. The Shark Distributed SQL Query Engine heavily uses the underlying Spark execution engine to execute query computations, as depicted in Figure 4.1. Whenever a Shark query is submitted to Shark, it will be converted to a Spark job and submitted to the Spark execution engine. The *DAGScheduler* of Spark in turn splits the job into a Directed Acyclic Graph (DAG) of stages. A new stage is created whenever data exchange has to occur between the worker nodes (e.g., a data shuffle before a reduce operation can be performed). This means that simple scan queries often only need one stage, whereas complex aggregation queries need multiple stages. While stages belonging to the same job need to be executed strictly in order, each stage consists of a Bag of Tasks (BoT) which does not have any particular order. Whenever a stage is ready to be executed, it will submit its BoT to the *TaskScheduler*. Whenever worker resources become available, the *TaskScheduler* will then select the BoTs to schedule based on the scheduling policy specified (currently either FIFO or Fair Scheduling). The corresponding BoT's *TaskManager* will then decide to assign a task to a resource or not based on data locality, using delay scheduling [61].

The Perkin query scheduler builds on Spark, extending Spark's default *TaskScheduler* as the *PerkinTaskScheduler*. It will define its order of stages based on the scheduling policy selected (see Section 4.2.3).

More in detail information regarding the Shark and Spark architecture can be found in [59, 63].

4.2.3 Scheduling Policies

Spark comes with two built-in scheduling policies: First In first Out (FIFO), and the Fair Scheduler. The FIFO scheduling policy simply executes the queries in the order they arrive, while the Fair Scheduler allows system administrators to define different job pools which each get a fair share of the system's resources depending on their priority. In the remainder of this section we define scheduling policies which have not been implemented in Spark, but are possible to implement by using the predictors we evaluate or promising in general.

Earliest Deadline First (EDF)

The Earliest Deadline First (EDF) algorithm prioritises the queries based on the deadlines provided by the user. The query which has the earliest deadline will get precedence. This policy does not require any predictor to base its decisions on.

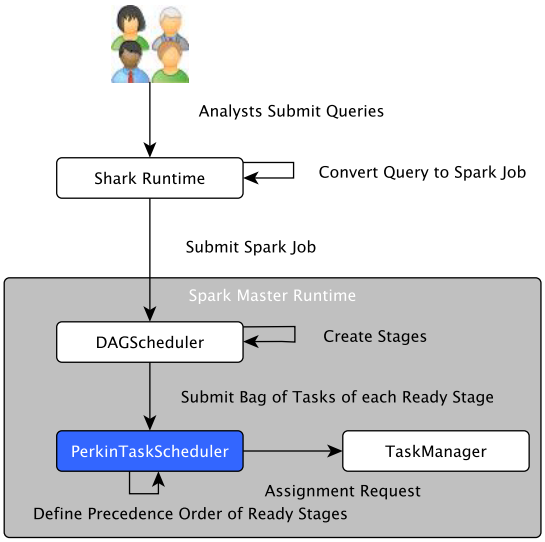


Figure 4.1: A simplified overview of the Spark Scheduler Architecture, including the Perkin module additions.

Shortest Job First with Stragglers (SJF-S)

For each query that arrives we predict the execution time using the query execution time predictor. With the Shortest Job First (SJF) policy we then give queries with shorter execution times precedence over queries with longer excution times. This policy comes with a caveat, namely *starvation*: When the cluster is at its maximum capacity, long queries will never complete because the short queries will always get precedence. To overcome this problem we configure the Fair Scheduler to have two pools. One of the pools will run SJF, and the other will run FIFO. Initially the queries will start in the SJF pool, but when the response time exceeds the average response time for their query type by two times the standard deviation, they will be moved over to the FIFO pool, guaranteeing completion. The FIFO pool is configured to get at least 50% of the resources when it is non-empty. The SJF pool will get assigned all the remaining resources. This results in SJF being run when the capacity of the cluster is not fully used, and SJF for the non-straggling as well as FIFO for the straggling queries when the cluster is hitting its maximum load. We call this the SJF-S policy.

Highest Slowdown First with Stragglers (HSF-S)

The Highest Slowdown First with Stragglers (HSF-S) scheduling policy works exactly the same as the SJF-S policy, except that it uses a slowdown estimation model rather than the query execution time predictor to define priority values. This policy also has the issue of starvation, and therefore we need to take similar precautions

as with the SJF-S policy.

Query slowdown is defined to be the ratio of a query's execution time in a multi-tenant environment vs. its execution time in an isolated single tenant environment.

This can be written as follows:

$$S = \frac{t_{multi}}{t_{single}}, \tag{4.1}$$

where t_{multi} and t_{single} are the execution times in the multi-tenant and single tenant environments respectively.

The query slowdown can be predicted using the same machine learning algorithm we will use for predicting the execution time. Instead of training based on observed execution times, we simply train the machine learning algorithm based on observed slowdowns. Note that although we explain how to create a slowdown predictor, we will only evaluate the query time predictors, starting in the next section.

4.3 Experimental Method

The machine learning based query time predictors need to be trained on an output trace. This is a dataset of queries together with their characteristics, that are labeled with their actual execution or response time. Because no such output traces are publicly available, we had to construct our own. In this section we describe our method for constructing the output traces, as well as the procedure of training and evaluating the predictors. We describe how we constructed realistic output traces in Section 4.3.1. We then elaborate on the performance metrics we use to measure the accuracy of the query time predictors (Section 4.3.2), followed by explaining the evaluation procedure used (Section 4.3.3).

4.3.1 Output Traces

In this section we present the output traces we are using to evaluate the query response time predictors on. We designed our own workload and executed this against different Spark scheduler configurations in order to obtain the output traces. The workload resembles situations where multiple tenants are executing queries on one cluster simultaneously, contrary to the single-tenant approach in Chapter 3.

For the workload we selected a pre-processed hotel price dataset similar to the data generated from the queries 4 to 6 of Section 3.2.1 (see Table 4.1 for an overview). We then use our workload generator to generate 3000 queries that operate on the selected datasets. These queries are representative for the query workload fired onto the cluster by the Azavista web front-end and are generated from 5 parameterised query types uniformly at random. The query inter arrival time is exponentially distributed with $\lambda = \frac{4}{7}$. This means a new query will be submitted to the cluster approximately every couple of seconds. We define the 5 query types we use below, in addition to summarizing their properties in Table 4.2.

Table 4.1: Summary of Datasets.

Table	# Columns	Description
hotel_prices_last	7	Hotel room prices of last collection day.
hotel_city_ranking	3	Average hotel room price per city.
hotel_trend	8	Recent hotel room price trends.
hotel_price_changes	8	Recent hotel room price changes.

Table 4.2: Summary of SQL Queries.

Query	Input Size		Tables
	MiB	*Records*	
1	8.7	54K	hotel_city_ranking
2	4488.5	45.9M	hotel_prices_last
3	14981.2	47M	hotel_trend
4	4488.5	45.9M	hotel_prices_last
5	14252.9	49M	hotel_price_changes

Query Type 1 computes the N most expensive cities within the provided budget Y on a check-in date range X to Z. The cost of a city is defined as the mean price of all its double rooms.

Query Type 2 computes the N cheapest hotels in a city Y on a check-in date X.

Query Type 3 computes the N cities with the highest average hotel room price increase over the last 5 days for check-in date X.

Query Type 4 computes the N cheapest dates to book an average priced hotel room in city Z between check-in date X and Y.

Query Type 5 computes the top N hotels with the largest average priced price changes between the last two collection dates, for check-in date X.

This general workload was executed on a 5-node Shark cluster. We used same experimental setup as in Section 3.3, with the exception that `c3.4xlarge` instances were used (only 30 GiB memory, but double the number of CPU cores). In addition we also artificially increased the number of task slots to 48, which is 3 times the number of CPU cores. This made for better resource utilization. Together with the response time, we collected data for the following 53 query features, which are categorised as follows:

- 15 features together determine the type of the query that was run. Each of these features characterises a certain query clause (e.g., `q_select` determines the number of columns in the select clause, and `q_join_t` determines the number of `JOIN`s in the query).

- 25 features represent different metrics of memory usage, CPU load, disk I/O and network I/O *of the cluster*, at query submission. They include the maximum, median and total sum of values of all worker nodes.

- 13 features represent metrics retrieved from the Spark run-time at query submission, including min/max/total active tasks, stages running/waiting, rdd blocks and block memory in use *by the cluster*, as well as the total number of input partitions and input size for each query.

The workload is executed in different experiments in order to obtain the output traces we need for predictor training:

OT-SEQ is an output trace of executing the workload sequentially in a single-tenant environment. This is done by waiting for each query to complete before submitting another to the cluster.

OT-FIFO is the result from executing the workload on a multi-tenant Spark cluster with a FIFO queue. Queries are submitted with an exponentially distributed inter arrival time (as defined before). Multiple queries are executing in the cluster at the same time, whereas in OT-SEQ only one query is running in the cluster at any given time.

OT-FAIR is similar to OT-FIFO, but instead of using a FIFO queue, we assign execution time to each query in a round robin manner. This is done by configuring one query pool in the built-in Spark FAIR scheduler.

Figure 4.2 depicts the density distribution of query response times in each output trace, while Table 4.3 summarizes query response time statistics. As can be seen in the figure, the sequential execution yields the lowest average response time, followed by FIFO. The fair scheduler gives each query equal CPU time, but because of that a lot of queries are running concurrently and thus only limited resources can be used per query. This results in a far longer average response time. All of the output traces have stragglers, but OT-FAIR has by far the longest tail.

Table 4.3: Query Response Time Statistics

Output Trace	**Mean (s)**	**CV.**	**Max (s)**
OT-SEQ	4.13	1.16	80.16
OT-FIFO	5.33	0.62	23.99
OT-FAIR	15.05	0.75	101.82

Recall that we want to predict both query response time and execution time, and that we use the output traces to train the predictors. Although we can use the above output traces to train query *response* time predictors, we do not have any data for the query *execution* time. This does not mean we cannot predict the execution time at all. For the OT-SEQ and OT-FAIR traces the response time equals the execution

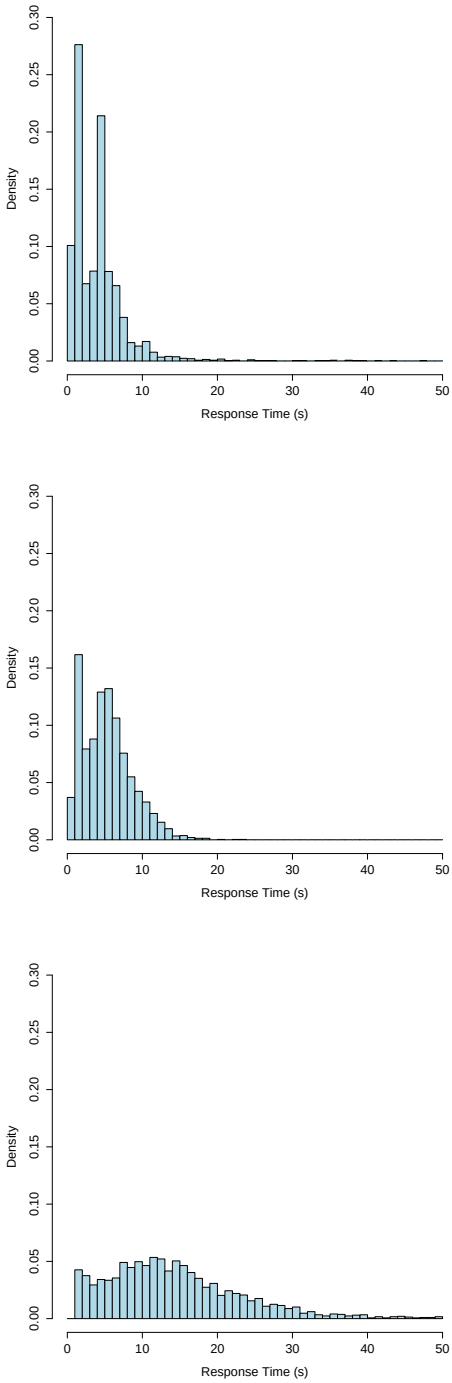


Figure 4.2: Query Response Time Densities for OT-SEQ (top), OT-FIFO (middle), and OT-FAIR (bottom).

time by definition, since no queue is present. OT-FIFO has a queue, and therefore the response time is different from the execution time. However, since the response time distribution of OT-FIFO is very similar to the one of OT-SEQ (see Figure 4.2), we can use the execution time of OT-SEQ as an estimation for the execution time of OT-FIFO. This results in the output trace OT-FIFO-EXEC, as defined below:

OT-FIFO-EXEC This is the OT-FIFO trace with its response times replaced with the response times of OT-SEQ. Recall that the response times of OT-SEQ equal its execution times. Thus, we created an output trace that approximates FIFO execution times, and that can be used for predicting query execution times in FIFO.

4.3.2 Performance Metrics

In order to be able to reason about the performance differences between the different predictors, we use the accuracy metrics commonly used in the field of machine learning. Each of them has certain benefits compared to the others, therefore we report all three of them. Given the predicted values of the response time in vector y' and the observed values in y in m observations, the metrics are defined as follows:

1. *Root Mean Square Error (RMSE)*: this metric is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y'_i - y_i)^2} \tag{4.2}$$

   The RMSE can be interpreted as the number of seconds the prediction is off on average. The lower the value the better the prediction, with 0 being a perfect prediction. Benefit of this metric is that it has the same units as the predicted quantity.

2. *Coefficient of Determination (R^2)*: this metric is defined as:

$$R^2 = 1 - \frac{\sum_i^m (y_i - y'_i)^2}{\sum_i^m (y_i - \bar{y})^2}, \tag{4.3}$$

   where the numerator is called the residual sum of squares and the denominator the total sum of squares, and $\bar{y}$ is equal to the mean of the values in the vector y. This metric shows the fraction of the variance that is explained by the model. If this metric returns 1, the prediction is perfect. If this metric returns a negative number, the prediction is actually worse than taking the mean of all observed values as prediction.

3. *Mean Absolute Percentage Error (MAPE)*: this metric is defined as:

$$\text{MAPE} = \frac{1}{m}\sum_{i=1}^{m}\left|100 \cdot \frac{y'_i - y_i}{y_i}\right| \tag{4.4}$$

   The MAPE is probably the easiest to interpret since it uses the most intuitive definition of error: the percentage a prediction is off (on average).

4.3.3 Evaluation Procedure

We evaluate the predictors *MLR*, *SVR* and *Last2* from Section 4.1 on the output traces OT-FAIR, OT-FIFO, and OT-FIFO-EXEC from Section 4.3.1. We measure the performance using the metrics from Section 4.3.2. OT-FAIR and OT-FIFO are used to assess query response time prediction capabilities, and OT-FIFO-EXEC to give insights in query execution time prediction performance. We describe how we evaluate *Last2* at the end of this section, but we start off by describing the evaluation procedure for *MLR* and *SVR*.

Evaluating MLR and SVR

For both *MLR* and *SVR* we apply the general method for training machine learning predictors as described in Section 2.4.1 (all computations are carried out in R [49]):

1. We start out with the output trace matrix X of $n = 53$ features (columns) and $m = 3000$ observations (rows). The vector y contains the 3000 corresponding query times.

2. We clean the dataset. We remove each feature vector x_i that has identical observations (constant). Redundant features are removed: we only retain one feature out of each possible feature pair for which holds that their absolute correlation is higher than 0.85. An example of this step is presented in Appendix D.

3. We perform column-wise normalization on the remaining feature vectors in X so all feature observations have a value between 0 and 1. The query times in vector y are not normalized.

4. (Optional) We use Principal Component Analysis to visualize the cleaned dataset (see Appendix D).

5. We perform stratified random sampling from X into a 80% training set $T_{training}$ and 20% test set T_{test} by splitting the dataset X into $\sqrt{m} = 55$ bins based on query time. Sampling using these bins makes sure $T_{training}$ and T_{test} have similar distribution of query times as the original dataset X. We fix the random seed to be equal to the iteration number so the computation is replicable and has equal data samples for all predictors. This is the best approach for our dataset. Did we have more query types, it would have been better to make sure some query types were only in the training set, and some of them were only in the test set.

6. Parameter tuning is performed using 5-fold cross validation on T_{train} using R^2 as performance metric. For MLR we conduct feature selection in this phase using best first search [50], not adding an additional feature to the selected features if the R^2 does not improve by at least 0.005. For SVR we use this phase for tuning epsilon kernel function parameters cost

($C \in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2\}$), gamma ($\gamma \in \{\frac{1}{32}, \frac{1}{16}, \ldots, \frac{1}{2}, 1\}$), and epsilon ($\epsilon \in \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1\}$) using grid search [27]. We use the same features for SVR that were selected for MLR. No individual feature selection is performed for SVR since kernel parameter tuning already takes longer than the feature selection for MLR.

7. The best parameters from the parameter tuning phase are used to infer a model h from $T_{training}$.

8. We test the accuracy of the predictor h using RMSE, R^2 and MAPE.

9. Steps 5 to 8 are carried out 5 times and we report the minimum, median and maximum for each predictor. This shows how much predictor performance depends on the data partitioning in step 5.

Evaluating Last2
For the *Last2* predictor we simply do stratified random sampling in each iteration (iteration number equals random seed), like with the machine learning algorithms, and compute the prediction accuracy on the 20% test set (see step 5). This way, the performance of all predictors is measured on the same data partitioning.

4.4 Experimental Results

The results of executing the evaluation procedure from Section 4.3.3 on output traces OT-FIFO, OT-FIFO-EXEC and OT-FAIR are presented in this section. We trained the predictors 5 times using different data partitioning, and report the minimum, median, and the maximum value of RMSE, R^2 and MAPE observed for each predictor. The results per output trace are in Tables 4.4, 4.5 and 4.6. The coloured cells mark the predictor that performs best on a specific metric in terms of the median. We discuss the predictor performance on each output trace below, after first listing the key findings.

Key Findings:

- All predictors are able to predict response time better than execution time based on comparing the results of OT-FIFO with OT-FIFO-EXEC.
- They overall perform best on the OT-FIFO output trace.
- The Last2 predictor has the worst performance, whereas both MLR and SVR's performance is acceptable, but not extremely good.
- MLR and SVR have very similar performance.
- As we foresaw in step 4 of the evaluation procedure (Section 4.3.3), features characterising the query type are among the most important features MLR

and SVR use. Spark run-time features come second, and resource utilization features are often discarded.

Table 4.4: Predictor accuracy for the FIFO output trace trained on query response time. The colored background highlights the predictor with the best median.

OT-FIFO	**MLR**			**SVR**			**Last2**		
	Min	*Median*	*Max*	*Min*	*Median*	*Max*	*Min*	*Median*	*Max*
RMSE (seconds)	1.78	1.86	2.16	1.74	1.83	2.20	3.80	4.12	4.61
R^2 ($-\infty$ to 1)	0.63	0.645	0.69	0.63	0.647	0.69	-0.62	-0.61	-0.56
MAPE (%)	34.67	35.08	37	36.34	37.04	38.63	85.91	86.60	90.07

Table 4.5: Predictor accuracy for the FIFO output trace trained on query execution time. The colored background highlights the predictor with the best median.

OT-FIFO-EXEC	**MLR**			**SVR**			**Last2**		
	Min	*Median*	*Max*	*Min*	*Median*	*Max*	*Min*	*Median*	*Max*
RMSE (seconds)	1.66	3.11	7.21	1.31	2.99	7.21	4.04	5.25	6.86
R^2 ($-\infty$ to 1)	0.21	0.41	0.61	0.21	0.46	0.74	-1.54	-0.67	0.29
MAPE (%)	37.17	42.11	42.77	28.33	33.38	41.95	111.31	117.57	132.11

Table 4.6: Predictor accuracy for the FAIR output trace trained on query response time. The colored background highlights the predictor with the best median.

OT-FAIR	**MLR**			**SVR**			**Last2**		
	Min	*Median*	*Max*	*Min*	*Median*	*Max*	*Min*	*Median*	*Max*
RMSE (seconds)	5.66	8.17	9.33	5.58	8.38	9.74	13.34	15.72	18.56
R^2 ($-\infty$ to 1)	0.49	0.55	0.59	0.47	0.53	0.61	-1.26	-0.88	-0.75
MAPE (%)	40.70	41.11	46.58	32.23	34.50	37.18	80.96	84.25	89.70

1) OT-FIFO: As seen in Table 4.4, both MLR and SVR have acceptable performance with a median RMSE around 1.85 seconds, R^2 around 0.65, and a MAPE of $\leq 38\%$. This means it is expected that these predictors will be off by approximately 2 seconds in predicting response time, compared to more than 4 seconds for the naive Last2 predictor. Recall that the R^2 metric is positive when the prediction is better than simply taking the mean of all observations in the test set, and it is negative when it is worse than this mean. Last2 always models the test set worse than its mean, while MLR and SVR have at least 65% ($R^2 = 0.65$) of the performance of a perfect predictor ($R^2 = 1$).

2) OT-FIFO-EXEC: As seen in Table 4.5, SVR clearly has superior performance compared to the other predictors. While having a better MAPE (33%) compared to the OT-FIFO trace (37%), the R^2 is much lower (0.65 compared to 0.46).

3) OT-FAIR: As seen in Table 4.6, the median RMSE is much higher for all predictors compared with the results on other workloads. However, the median R^2

and MAPE are still acceptable for MLR and SVR. Performance of Last2 is still sub-par.

Overall MLR and SVR perform similarly using different output traces. If instead of just taking the features selected for MLR, SVR would do its own feature selection, it might be better than MLR in all cases at the cost of very long training time. Last2 is not good enough to be used as reliable predictor, while both MLR and SVR would do reasonably well.

4.5 Summary

In this chapter we have evaluated the performance of three query time predictors on three different realistic query output traces we generated. The predictors are Multiple Linear Regression (MLR), Support Vector Regression (SVR), and Last2. It turned out that both MLR and SVR have similar acceptable performance with a median (MAPE) error around 35% and a maximum error never bigger than 47%. However, the Last2 predictor is not suitable for predicting query time at all with median errors of 87% or more, and maximum errors of 132%! It turned out that all predictors are more accurate in predicting query response time than query execution time. This has likely to do with the fact that we did not have a real output trace for execution time, but constructed one using some assumptions. The machine learning algorithms heavily base their prediction on the values of the features that characterise the query, whereas Spark run-time and resource utilization features have a less significant impact.

The evaluation of these query time predictors answered the second and last research question of this thesis (**RQ2**): *What is the performance of query time predictors that utilize machine learning techniques?*

Chapter 5

Conclusion and Future Work

Big Data processing has become possible for almost any company, mainly due to lower prices of computing resources in the cloud. However, especially for SMEs that do not have the expertise to conduct in-depth performance comparisons between the Distributed SQL Query Engines on the market, it is challenging to select an appropriate engine. Besides this, DSQEs generally do not provide performance guarantees, while the companies that use these systems *do* have to give guarantees to their customers in the SLA. We formulated research questions based on these two challenges, and answered them in our work. In this chapter we start off by listing our findings in Section 5.1, followed by presenting possible directions for future work in Section 5.2.

5.1 Conclusion

We provide our conclusions of this work by summarizing all the research questions listed in our problem statement in Section 1.1:

RQ1 *What is the performance of state-of-the-art Distributed SQL Query Engines in a single-tenant environment?*

We have defined an empirical evaluation method to assess the performance of different query engines. Despite not covering all the methodological aspects of a scientific benchmark, this micro-benchmark gives practical insights for SMEs to take informed decisions when selecting a specific tool. Moreover, it can be used to compare current and future engines. The method focuses on three performance aspects: processing power, resource utilization, and scalability.

Using both a real world and a synthetic dataset with representative queries, we have evaluated the query engines' performance. We found that different query engines have widely varying performance. Although Hive is almost always outperformed by the other engines, it highly depends on the query type and input size whether Impala or whether Shark is the best performer.

Shark can also perform well on queries with over 500 GiB in input size in our cluster setup, while Impala starts to perform worse for these queries. Overall Impala is the most CPU efficient, and all query engines have comparable resource consumption for memory, disk and network. A remarkable result found is that query response time does not always improve when adding more nodes to the cluster. Our detailed key findings can be found with every experiment in Section 3.4.

RQ2 *What is the performance of query time predictors that utilize machine learning techniques?*

We have evaluated the performance of three query time predictors on three different realistic query output traces we generated. The predictors are Multiple Linear Regression (MLR), Support Vector Regression (SVR), and Last2. It turned out that both MLR and SVR have similar acceptable performance with a median (MAPE) error around 35% and a maximum error never bigger than 47%. However, the Last2 predictor is not suitable for predicting query time at all with median errors of 87% or more, and maximum errors of 132%! It turned out that all predictors are more accurate in predicting query response time than query execution time. This has likely to do with the fact that we did not have a real output trace for execution time, but constructed one using some assumptions. The machine learning algorithms heavily base their prediction on the values of the features that characterise the query, whereas Spark run-time and resource utilization features have a less significant impact.

The work regarding **RQ1** has resulted in the submission of the article:

[57] Stefan van Wouw, José Viña, Dick Epema, and Alexandru Iosup. *An Empirical Performance Evaluation of Distributed SQL Query Engines*. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE), 2015.

5.2 Future Work

Although our work already yields promising results, it should be extended. We suggest the following directions for future work:

1. We were not able to also evaluate the performance of Drill, Presto, and Hive-on-Tez in our work, due to them lacking features at the time of our query engine selection. Other DSQEs such as SparkSQL are also in development. Evaluating these platforms with our micro-benchmarking suite would give SMEs more options to choose from.

2. Our micro-benchmarking suite assumes a single-tenant environment. In order to evaluate the performance of DSQEs in a multi-tenant environment,

workload traces are required. However, at the time of writing no such traces are publicly available. Synthetic workloads such as the ones we created for evaluating query predictor performance could be used to also test the performance of query engines.

3. Due to time constraints we did not implement Perkin into Spark. In order to assess whether the proposed scheduling policies improve the query response times, they need to be implemented in Spark and evaluated using workloads similar to the ones we designed in Chapter 4.

4. Although the time predictors have acceptable performance on our set of five query types, we do not know if these predictors also have this performance when used on queries outside of our output traces. As long as we only have a handful of different query types we know by hard, we can simply retrain the predictor (offline) every time a new query type is added to the production system. This way we can ensure accuracy. This is the best approach when only having access to a limit number of query types (which was the case, as no output traces are publicly available for DSQEs).

   However, if public output traces became available, we would have access to thousands of every changing query types. In this case a better approach would be not to retrain offline every time we encounter a new query type. We could retrain using online machine learning while the predictor is active in the production system. The sampling of the training set also needs to change in this case. We should also have some queries in the test set that are not in the training set so as to detect overfitting in the cross validation phase. Future work is required to evaluate the effectiveness of this approach.

Bibliography

[1] Amazon Elastic MapReduce (EMR). http://aws.amazon.com/emr/. [Online; Last accessed 1st of September 2014].

[2] Amazon Elastice Compute Cloud (EC2). http://aws.amazon.com/ec2/. [Online; Last accessed 1st of September 2014].

[3] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/. [Online; Last accessed 1st of September 2014].

[4] AMPLab's Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark. [Online; Last accessed 1st of September 2014].

[5] Apache Cassandra. http://cassandra.apache.org. [Online; Last accessed 1st of September 2014].

[6] Apache Hadoop. http://hadoop.apache.org. [Online; Last accessed 1st of September 2014].

[7] Apache Tez. http://tez.apache.org. [Online; Last accessed 1st of September 2014].

[8] Cloud Central Cloud Hosting. http://www.cloudcentral.com.au. [Online; Last accessed 1st of September 2014].

[9] Collectl Resource Monitoring. http://collectl.sourceforge.net. [Online; Last accessed 1st of September 2014].

[10] Digital Ocean Simple Cloud Hosting. http://www.digitalocean.com. [Online; Last accessed 1st of September 2014].

[11] Dropbox. http://www.dropbox.com. [Online; Last accessed 1st of September 2014].

[12] Gmail. http://www.gmail.com. [Online; Last accessed 1st of September 2014].

[13] GoGrid Cloud Hosting. http://www.gogrid.com. [Online; Last accessed 1st of September 2014].

[14] Google BigQuery. http://developers.google.com/bigquery/. [Online; Last accessed 1st of September 2014].

[15] Impala. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/. [Online; Last accessed 1st of September 2014].

[16] Impala Benchmark. http://blog.cloudera.com/blog/2014/05/new-sql-choices-in-the-apache-hadoop-ecosystem-why-impala-continues-to-lead/. [Online; Last accessed 1st of September 2014].

[17] Lambda Architecture. http://lambda-architecture.net. [Online; Last accessed 1st of September 2014].

[18] MLLib. http://spark.apache.org/mllib. [Online; Last accessed 1st of September 2014].

[19] Presto. http://www.prestodb.io. [Online; Last accessed 1st of September 2014].

[20] RackSpace: The Open Cloud Company. http://www.rackspace.com. [Online; Last accessed 1st of September 2014].

[21] SAP. http://www.sap.com. [Online; Last accessed 1st of September 2014].

[22] Storm. http://www.storm-project.org. [Online; Last accessed 1st of September 2014].

[23] Windows Azure. http://www.windowsazure.com. [Online; Last accessed 1st of September 2014].

[24] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, 2013.

[25] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[26] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96. ACM, 2005.

[27] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[28] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[30] Menno Dobber, Rob van der Mei, and Ger Koole. A prediction method for job runtimes on shared processors: Survey, statistical analysis and new avenues. *Performance Evaluation*, 64(7):755–781, 2007.

[31] Harris Drucker, Chris JC Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. *Advances in neural information processing systems*, 9:155–161, 1997.

[32] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 689–692, 2012.

[33] Avrilia Floratou, Umar Farooq Minhas, and Fatma Ozcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[34] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 international conference on Management of data*, pages 1197–1208, 2013.

[35] Michael Hausenblas and Jacques Nadeau. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data*, 2013.

[36] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208, 2011.

[37] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 104–113, 2011.

[38] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[39] Kamal Kc and Kemafor Anyanwu. Scheduling hadoop jobs to meet deadlines. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 388–392. IEEE, 2010.

[40] Xuan Lin, Ying Lu, Jitender Deogun, and Steve Goddard. Real-time divisible load scheduling for cluster computing. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 303–314. IEEE, 2007.

[41] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. Boosting algorithms as gradient descent in function space. NIPS, 1999.

[42] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. `http://csrc.nist.gov/publications /nistpubs/800-145/SP800-145.pdf`, September 2011. [Online; Last accessed 1st of September 2014 in Google's Cache].

[43] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[44] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[45] Andrew Ng. Stanford Machine Learning Coursera Course. 2014.

[46] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[47] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178, 2009.

[48] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. Why you should run TPC-DS: a workload analysis. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1138–1149. VLDB Endowment, 2007.

[49] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[50] Piotr Romanski. *R FSelector Package*, 2014.

[51] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, page 2. ACM, 2012.

[52] Yassine Tabaa, Abdellatif Medouri, and M Tetouan. Towards a next generation of scientific computing in the Cloud. *International Journal of Computer Science (IJCSI)*, 9(6), 2012.

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

[54] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth,

et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[55] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[56] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1):37–52, 1987.

[57] Stefan van Wouw, José Viña, Dick Epema, and Alexandru Iosup. An Empirical Performance Evaluation of Distributed SQL Query Engines (under submission). In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2015.

[58] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[59] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM International Conference on Management of Data*, pages 13–24, 2013.

[60] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. Towards machine learning-based auto-tuning of mapreduce. In *IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 11–20, 2013.

[61] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2, 2012.

[63] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[64] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10, 2012.

Appendix A

Detailed Distributed SQL Query Engine Performance Metrics

In this appendix detailed performance metrics of the experiments are reported as-is, extending on the experimental results in Section 3.4. This information can be used to get more precise insights into performance differences across query engines. For example, if an SME has good disk performance but not so much memory in its cluster, it can focus on these aspects and select the query engine that does not consume a lot of memory.

For all tables holds that the green coloured cells are marking the system that has the best performance for a query for a certain metric, and that cells indicating a 0 are actually presenting values *close* to 0 (because of rounding).

Query 1 (cold)	**Hive**			**Impala**			**Shark**		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	154	1.58	666	53	0.08	60	78	0.16	104
Disk Read (MiB/s)	8	1.82	110	17	2.05	204	13	1.40	119
Disk Read Total (MiB)	1,229	1.17	5,211	970	0.32	1,202	1,138	0.25	1,871
Disk Write (MiB/s)	24	1.35	172	53	1.05	245	35	1.11	204
Disk Write Total (MiB)	3,455	0.30	5,721	2,954	0.10	3,653	2,989	0.11	3,682
Memory (MiB)	5,320	0.51	16,171	11,346	0.27	17,059	9,696	0.37	16,494
Network In (MiB/s)	21	0.97	97	37	0.88	141	25	1.22	152
Network In Total (MiB)	3,015	0.16	3,637	2,070	0.14	2,863	2,105	0.20	2,914
Network Out (MiB/s)	20	1.21	117	36	0.84	141	24	1.20	134
Network Out Total (MiB)	2,919	0.57	5,452	1,996	0.13	2,681	2,047	0.15	2,842
Response Time (s)	142	0.13	191	54	0.12	69	81	0.21	108

Table A.1: Statistics for Query 1 (cold), Data Scale: 100%, Number of Nodes: 5.

Query 1 (warm)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	149	1.69	674	54	0.08	63	59	0.07	65
Disk Read (MiB/s)	2	3.51	78	1	7.09	146	5	2.98	117
Disk Read Total (MiB)	202	2.68	2,674	67	1.47	384	60	1.46	341
Disk Write (MiB/s)	29	1.27	180	63	1.04	248	0	2.97	0
Disk Write Total (MiB)	3,272	0.31	5,609	2,971	0.10	3,809	0	0.56	0
Memory (MiB)	17,041	0.27	27,134	66,865	0.02	68,249	53,384	0.13	67,168
Network In (MiB/s)	26	0.91	93	44	0.73	142	1	2.15	11
Network In Total (MiB)	2,929	0.11	3,642	2,102	0.14	2,911	11	0.44	23
Network Out (MiB/s)	25	1.20	113	43	0.70	132	1	2.91	14
Network Out Total (MiB)	2,837	0.62	5,597	2,029	0.12	2,603	10	0.97	33
Response Time (s)	110	0.08	129	45	0.06	50	11	0.04	12

Table A.2: Statistics for Query 1 (warm), Data Scale: 100%, Number of Nodes: 5.

Query 2 (cold)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	2,852	0.41	4,807	417	0.14	638	1,346	0.08	1,563
Disk Read (MiB/s)	12	1.67	116	30	1.42	193	28	0.87	101
Disk Read Total (MiB)	10,680	0.94	38,117	10,237	0.16	11,491	11,145	0.12	13,330
Disk Write (MiB/s)	11	1.87	184	10	2.97	235	14	1.46	148
Disk Write Total (MiB)	9,502	0.38	17,079	3,499	0.08	4,209	5,554	0.08	6,488
Memory (MiB)	20,100	0.65	63,388	28,471	0.35	55,062	31,317	0.36	62,393
Network In (MiB/s)	10	1.59	185	14	1.10	85	11	2.52	171
Network In Total (MiB)	8,747	0.46	14,998	4,758	0.14	6,449	4,163	0.05	4,580
Network Out (MiB/s)	10	1.88	151	14	1.18	102	10	2.49	148
Network Out Total (MiB)	8,687	1.10	35,624	4,648	0.16	6,508	4,035	0.07	4,493
Response Time (s)	891	0.08	1,004	340	0.06	379	389	0.07	438

Table A.3: Statistics for Query 2 (cold), Data Scale: 100%, Number of Nodes: 5.

Query 2 (warm)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	2,648	0.49	5,106	399	0.02	425	938	0.08	1,102
Disk Read (MiB/s)	3	2.49	96	2	4.11	116	3	1.47	35
Disk Read Total (MiB)	2,439	1.17	11,615	433	0.80	1,373	1,129	0.53	2,250
Disk Write (MiB/s)	12	1.84	187	13	2.65	250	9	1.65	111
Disk Write Total (MiB)	8,779	0.48	17,612	3,512	0.09	4,176	2,861	0.06	3,178
Memory (MiB)	47,685	0.25	68,255	62,154	0.07	68,249	68,206	0.00	68,252
Network In (MiB/s)	11	1.87	203	16	1.09	86	7	2.30	209
Network In Total (MiB)	8,013	0.53	16,480	4,457	0.07	5,178	2,458	0.21	3,857
Network Out (MiB/s)	10	2.02	129	16	1.05	93	7	2.50	155
Network Out Total (MiB)	7,620	1.12	30,953	4,358	0.06	5,083	2,381	0.36	6,491
Response Time (s)	800	0.01	811	276	0.02	283	333	0.13	410

Table A.4: Statistics for Query 2 (warm), Data Scale: 100%, Number of Nodes: 5.

Query 3 (cold)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	6,647	0.39	10,853	551	0.01	569	1,726	0.08	1,924
Disk Read (MiB/s)	3	3.56	143	30	1.47	192	15	1.29	125
Disk Read Total (MiB)	12,624	0.95	45,355	11,959	0.03	12,449	12,645	0.09	14,605
Disk Write (MiB/s)	18	1.74	250	0	20.41	0	9	0.72	46
Disk Write Total (MiB)	64,007	0.30	106,898	0	1.45	1	7,353	0.09	8,384
Memory (MiB)	38,205	0.38	68,250	34,437	0.42	55,775	33,857	0.48	68,250
Network In (MiB/s)	11	1.75	198	21	1.30	102	5	4.40	216
Network In Total (MiB)	39,728	0.17	53,539	8,260	0.01	8,391	4,414	0.08	5,157
Network Out (MiB/s)	11	1.92	160	21	1.37	152	5	3.48	152
Network Out Total (MiB)	38,552	0.44	95,724	8,080	0.04	8,466	4,293	0.08	4,991
Response Time (s)	3,699	0.06	3,947	390	0.03	422	829	0.04	882

Table A.5: Statistics for Query 3 (cold), Data Scale: 100%, Number of Nodes: 5.

Query 3 (warm)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	6,347	0.36	13,671	561	0.02	606	1,413	0.10	1,664
Disk Read (MiB/s)	2	3.87	118	8	2.63	170	4	1.31	27
Disk Read Total (MiB)	6,704	1.03	24,295	2,786	0.88	9,649	4,554	0.26	5,906
Disk Write (MiB/s)	19	1.70	249	0	6.42	0	6	1.25	53
Disk Write Total (MiB)	62,028	0.24	100,819	0	0.14	0	7,194	0.08	8,074
Memory (MiB)	53,394	0.20	68,253	59,443	0.16	68,249	67,823	0.03	68,257
Network In (MiB/s)	12	1.79	197	24	1.29	112	4	2.52	202
Network In Total (MiB)	39,201	0.18	57,428	8,460	0.00	8,575	4,974	0.25	7,825
Network Out (MiB/s)	12	1.90	161	23	1.32	132	4	2.79	154
Network Out Total (MiB)	37,796	0.40	92,610	8,268	0.03	8,578	4,812	0.33	10,326
Response Time (s)	3,359	0.06	3,725	350	0.05	384	1,168	0.06	1,253

Table A.6: Statistics for Query 3 (warm), Data Scale: 100%, Number of Nodes: 5.

Query 4 (cold)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	29,615	0.05	32,528	15,022	1.48	78,620	10,741	0.03	11,321
Disk Read (MiB/s)	6	1.11	79	1	2.96	105	18	0.26	78
Disk Read Total (MiB)	28,445	0.82	77,034	79,334	0.38	159,037	27,229	0.06	29,998
Disk Write (MiB/s)	0	6.29	95	0	6.55	0	1	2.32	20
Disk Write Total (MiB)	1,245	0.15	1,618	21	0.26	29	1,205	0.04	1,297
Memory (MiB)	34,162	0.41	68,251	41,967	0.50	68,250	45,285	0.32	68,250
Network In (MiB/s)	2	0.78	67	0	23.04	112	1	6.79	143
Network In Total (MiB)	11,011	0.47	16,226	4,158	2.90	40,329	1,175	0.15	1,510
Network Out (MiB/s)	2	2.04	85	0	21.58	65	1	6.46	125
Network Out Total (MiB)	10,843	1.76	50,992	4,287	1.88	25,404	1,133	0.43	2,212
Response Time (s)	4,727	0.12	5,809	63,936	0.27	81,178	1,489	0.01	1,521

Table A.7: Statistics for Query 4 (cold), Data Scale: 100%, Number of Nodes: 5.

Query 4 (warm)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	58,536	0.05	66,008	13,182	0.02	13,579	11,276	0.02	11,733
Disk Read (MiB/s)	6	0.86	60	1	3.59	103	10	0.49	48
Disk Read Total (MiB)	50,883	0.51	87,299	40,925	0.09	45,915	16,687	0.17	24,282
Disk Write (MiB/s)	0	8.95	205	0	6.26	0	1	2.50	21
Disk Write Total (MiB)	2,753	0.14	3,796	16	0.01	16	1,205	0.04	1,322
Memory (MiB)	67,780	0.02	68,251	68,239	0.00	68,250	67,998	0.02	68,251
Network In (MiB/s)	2	0.86	127	0	17.10	18	1	3.64	48
Network In Total (MiB)	21,062	0.28	29,399	173	0.68	407	1,188	0.16	1,680
Network Out (MiB/s)	2	1.38	131	0	8.29	21	1	4.12	74
Network Out Total (MiB)	21,095	1.02	51,920	368	0.11	428	1,148	0.41	2,022
Response Time (s)	9,507	0.13	10,957	45,396	0.00	45,396	1,668	0.02	1,738

Table A.8: Statistics for Query 4 (warm), Data Scale: 100%, Number of Nodes: 5.

Query 5 (cold)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	1,470	1.76	7,351	269	0.16	445	1,355	0.11	1,664
Disk Read (MiB/s)	2	1.16	40	3	4.37	189	4	0.77	38
Disk Read Total (MiB)	1,808	0.57	4,628	703	0.36	1,064	999	0.10	1,245
Disk Write (MiB/s)	3	4.91	248	9	3.08	234	8	1.90	144
Disk Write Total (MiB)	3,577	1.39	16,997	1,900	0.11	2,293	2,164	0.09	2,662
Memory (MiB)	4,510	0.70	16,746	13,045	0.30	26,812	23,039	0.25	30,706
Network In (MiB/s)	1	7.80	168	15	1.04	78	6	3.30	177
Network In Total (MiB)	872	1.14	2,902	3,242	0.12	4,825	1,601	0.06	1,791
Network Out (MiB/s)	1	7.80	156	15	1.06	79	6	3.02	129
Network Out Total (MiB)	854	1.17	2,923	3,173	0.14	4,840	1,551	0.10	1,882
Response Time (s)	1,116	0.01	1,134	208	0.03	221	269	0.02	277

Table A.9: Statistics for Query 5 (cold), Data Scale: 100%, Number of Nodes: 5.

Query 5 (warm)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	4,452	0.94	15,631	263	0.03	278	1,338	0.11	1,656
Disk Read (MiB/s)	1	1.10	9	1	1.53	53	1	1.04	8
Disk Read Total (MiB)	1,743	0.22	3,611	188	0.18	281	295	0.42	772
Disk Write (MiB/s)	4	3.52	186	9	3.46	251	5	1.79	81
Disk Write Total (MiB)	6,324	0.86	22,651	1,793	0.12	2,466	1,302	0.15	1,795
Memory (MiB)	62,400	0.07	68,252	27,611	0.16	38,695	63,480	0.05	68,252
Network In (MiB/s)	1	5.37	195	15	1.08	78	4	4.55	194
Network In Total (MiB)	2,389	0.82	6,272	3,136	0.07	3,692	982	0.10	1,108
Network Out (MiB/s)	1	5.20	150	15	1.10	92	4	4.07	160
Network Out Total (MiB)	2,339	0.54	4,718	3,068	0.06	3,591	953	0.15	1,335
Response Time (s)	1,775	0.08	2,045	200	0.02	208	250	0.05	281

Table A.10: Statistics for Query 5 (warm), Data Scale: 100%, Number of Nodes: 5.

Query 6 (cold)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	426	1.29	1,908	122	0.07	132	332	0.10	385
Disk Read (MiB/s)	2	1.39	27	4	3.82	132	4	1.09	22
Disk Read Total (MiB)	890	0.46	2,012	418	0.12	468	415	0.17	551
Disk Write (MiB/s)	3	3.84	172	22	1.68	231	11	1.84	148
Disk Write Total (MiB)	1,441	1.14	5,886	2,205	0.07	2,384	1,318	0.11	1,647
Memory (MiB)	3,030	0.39	7,151	12,716	0.28	17,151	12,758	0.49	23,980
Network In (MiB/s)	1	6.08	165	34	0.69	89	9	2.24	125
Network In Total (MiB)	650	0.83	1,859	3,378	0.03	3,523	1,035	0.07	1,186
Network Out (MiB/s)	1	5.98	162	33	0.77	108	9	2.10	146
Network Out Total (MiB)	631	0.89	1,733	3,276	0.14	4,085	1,008	0.11	1,234
Response Time (s)	547	0.01	561	96	0.00	96	113	0.06	124

Table A.11: Statistics for Query 6 (cold), Data Scale: 100%, Number of Nodes: 5.

Query 6 (warm)	Hive			Impala			Shark		
	Mean	*CV.*	*Max*	*Mean*	*CV.*	*Max*	*Mean*	*CV.*	*Max*
CPU Seconds	427	1.38	1,944	121	0.07	139	333	0.08	381
Disk Read (MiB/s)	1	1.09	7	1	4.22	121	1	1.08	7
Disk Read Total (MiB)	536	0.09	827	123	0.61	340	106	0.31	231
Disk Write (MiB/s)	3	4.07	167	22	1.70	234	6	1.59	48
Disk Write Total (MiB)	1,437	1.23	5,968	2,218	0.12	2,700	598	0.18	787
Memory (MiB)	6,974	0.29	12,306	15,839	0.23	22,259	19,358	0.23	29,049
Network In (MiB/s)	1	5.83	162	34	0.72	90	5	3.32	103
Network In Total (MiB)	582	0.88	1,837	3,381	0.10	4,045	465	0.08	534
Network Out (MiB/s)	1	5.84	150	33	0.79	117	5	3.07	124
Network Out Total (MiB)	568	0.97	1,809	3,289	0.13	4,159	449	0.18	612
Response Time (s)	536	0.01	543	97	0.04	103	96	0.06	106

Table A.12: Statistics for Query 6 (warm), Data Scale: 100%, Number of Nodes: 5.

Appendix B

Detailed Distributed SQL Query Engine Resource Utilization

This appendix gives more insights in how resources are utilized over time. We normalized the response time on the horizontal axis of the figures, and calculated the mean resource utilization over all 10 experiment iterations. See Section 3.4.3 for accompanying explanations.

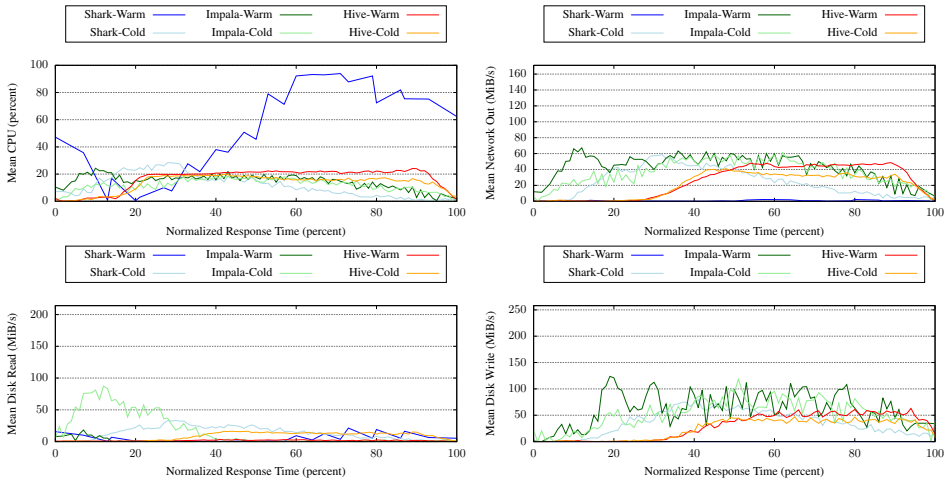


Figure B.1: CPU utilization (top-left), Network Out (top-right), Disk Read (bottom-left), Disk Write (bottom-right) for query 1 over normalized response time.

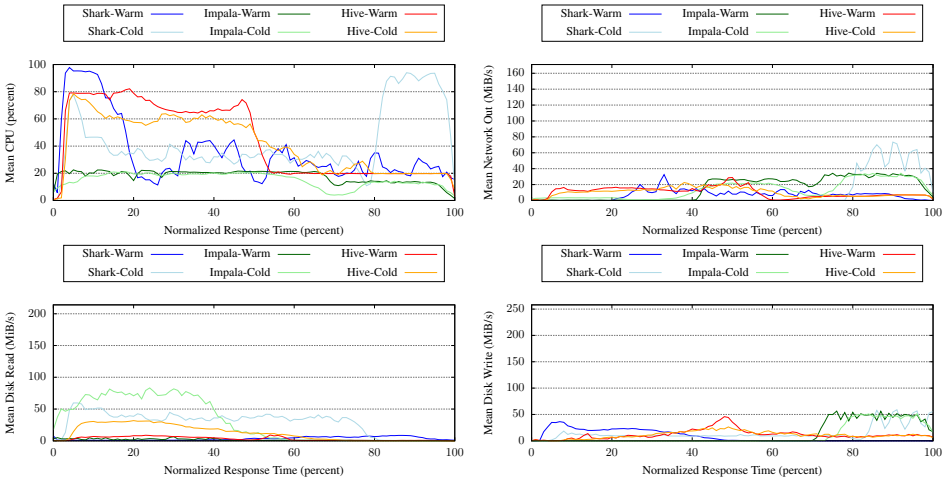


Figure B.2: CPU utilization (top-left), Network Out (top-right), Disk Read (bottom-left), Disk Write (bottom-right) for query 2 over normalized response time.

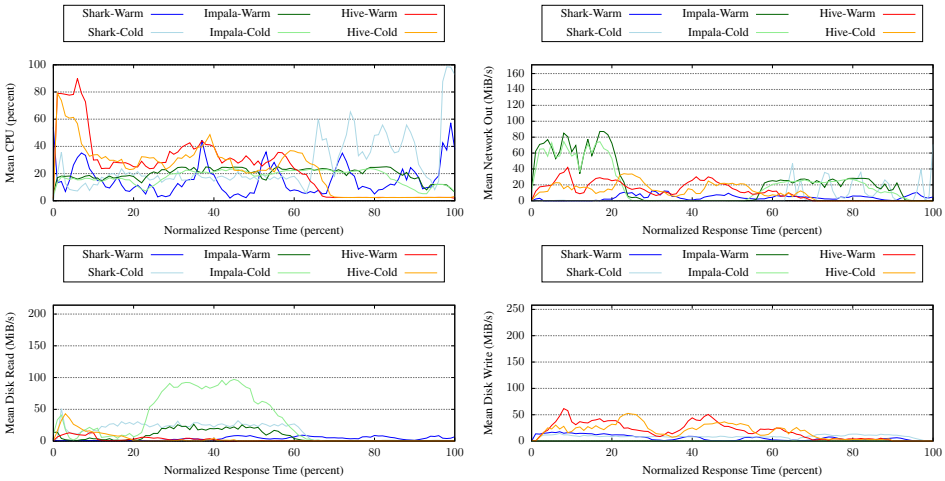


Figure B.3: CPU utilization (top-left), Network Out (top-right), Disk Read (bottom-left), Disk Write (bottom-right) for query 3 over normalized response time.

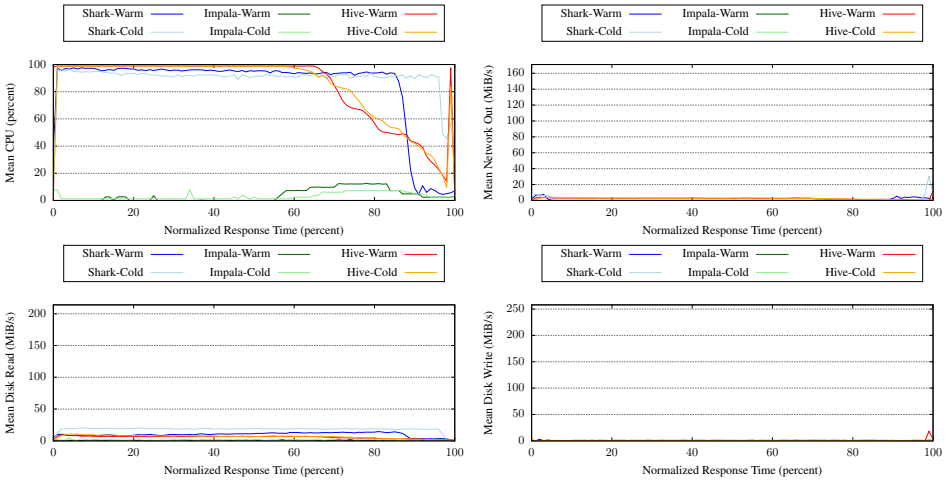


Figure B.4: CPU utilization (top-left), Network Out (top-right), Disk Read (bottom-left), Disk Write (bottom-right) for query 4 over normalized response time.

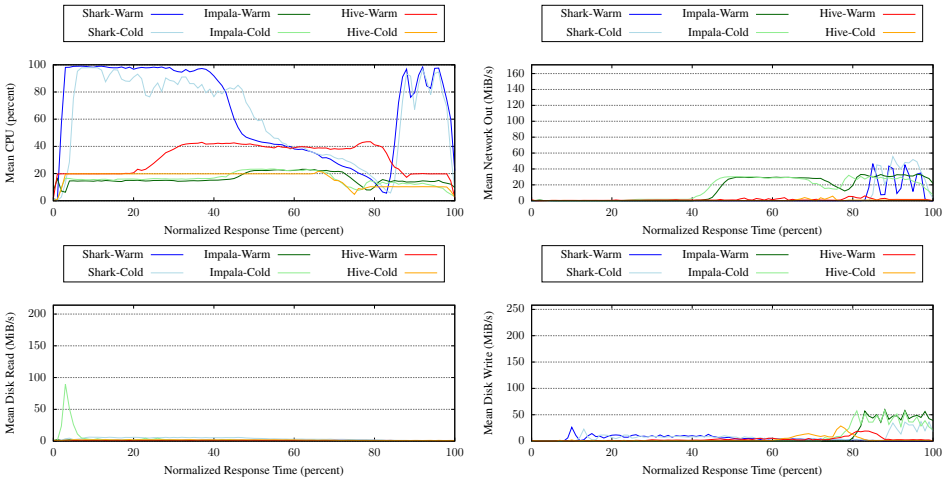


Figure B.5: CPU utilization (top-left), Network Out (top-right), Disk Read (bottom-left), Disk Write (bottom-right) for query 5 over normalized response time.

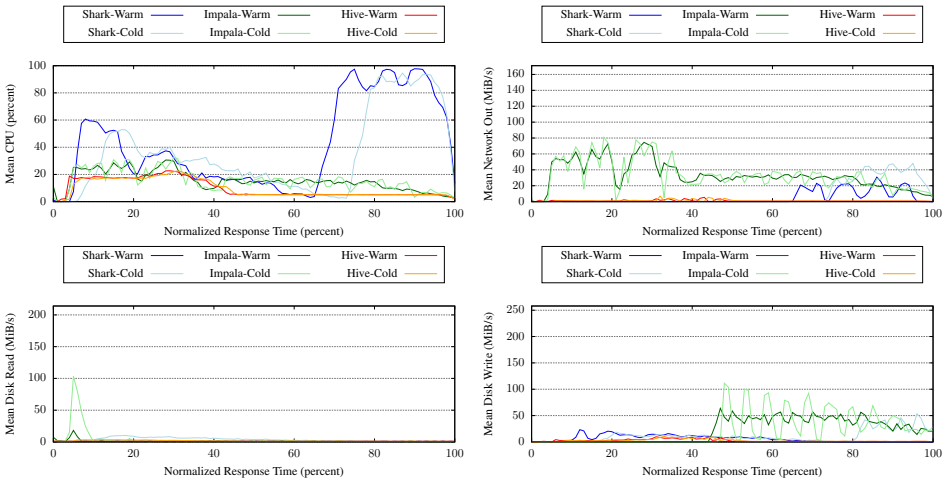


Figure B.6: CPU utilization (top-left), Network Out (top-right), Disk Read (bottom-left), Disk Write (bottom-right) for query 6 over normalized response time.

Appendix C

Cost-based Analytical Modeling Approach to Prediction

An alternative to the machine learning approach we used for our query time predictors in Chapter 4 is a cost-based analytical approach. We illustrate the cost-based analytical approach in order to get more insights in Spark's internal workings, but we will not use it because of its downsides listed at the end of this section. For the cost-based analytical approach we view the system as a whitebox, where we try to obtain a model based on the system's internal workings. We build upon the work performed in [39], where a deadline constrained job scheduler for Hadoop MapReduce is proposed. In this work distinct map and reduce phases are assumed where reduce phases start after all map tasks have completed. Spark's equivalent to this are stages. Each stage contains one or more RDD data transformation operations, and the last stage of a job also contains exactly one action operation (see Table C.1[1]). These operations are chained into an anonymous function to be executed by each task in the BoT.

For example, a query which groups by a certain field could get translated to a job with two stages: The first stage will execute tasks that *map* the input from HDFS, then the second stage will execute tasks that chain *groupByKey* and *saveAsTextFile* into one anonymous function in order to group by key (shuffling data) and save the result back to HDFS.

Adopting the notation of [39] and [40] we get the following:

- $Q = (A, \sigma, D)$: A query with arrival time A, relative deadline D (in seconds), and data input size σ (in KiB).

- $J = (A, \sigma, D, (S_1, S_2, \ldots, S_m))$: A Spark job that was converted from a Shark query Q, with m stages. Each stage S_j contains a Bag of Tasks that execute one anonymous function consisting of a chain of Spark Operations.

[1] Full explanation at: `http://spark.apache.org/docs/latest/scala-programming-guide.html`

Table C.1: Spark Operations (Transformations are Independent of Actions).

Spark RDD Transformations		
map	filter	flatMap
mapPartitions	mapPartitionsWithIndex	sample
union	distinct	groupByKey
reduceByKey	sortByKey	join
cogroup	cartesian	
Spark RDD Actions		
reduce	collect	count
first	take	takeSample
saveAsTextFile	saveAsSequenceFile	countByKey
foreach		

- n: Amount of total task slots available in the cluster (using the default settings of Spark, this equals the amount of total CPU cores).
- $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_u)$: The distribution vector for the data input to the first stage S_1, where u is the amount of tasks in S_1. Assuming the data is evenly distributed over the worker nodes, each of these tasks will get to process about $\frac{1}{u}\sigma$ KiB of data.
- f_o: Filter ratio for each Spark Operation o, which is the fraction of the input to o that will be outputted by o. For example, when $o = map$: $0 \leq f_{map} \leq 1$.
- c_o: Worst-case cost for each Spark Operation o, which is the amount of time it takes to process one unit of data (e.g. c_{map} could be 100ms for each KiB of data). This parameter needs to be trained or averaged over time.
- c_d: Time cost of shuffling one unit of data over the network.
- c_r: Time cost of loading one unit of data from disk.
- c_w: Time cost of writing one unit of data to disk.

Each jth stage will generate a certain amount of output data:

$$\sigma_{out}^{j} = \sigma_{out}^{j-1} \prod_{o \in O_j} f_o \quad \text{with } \sigma_{out}^{0} = \sigma, \tag{C.1}$$

where the sequence O_j contains the chain of Spark Operations in stage j.
The total execution time of each jth stage then becomes:

$$\epsilon_j = \frac{\sigma f_0 c_1 + \sigma f_1 c_2 + \sigma f_1 f_2 c_3 + \ldots + \sigma f_1 f_2 f_3 .. f_{p-1} c_p}{n_j}, \tag{C.2}$$

where $f_0 = 1$, n_j is the amount of task slots assigned to stage j, f_1 to f_p and c_1 to c_p are filter ratios and costs of all operations $o \in O_j$ respectively. This rewrites to the following general equation:

$$t_j = \sigma \sum_{i=1}^{p} c_i \prod_{k=0}^{i-1} f_k \quad \text{with } f_0 = 1, \tag{C.3}$$

The total task execution time for a job J then becomes:

$$T_J = \underbrace{\frac{\sigma c_r}{n_1}}_{\text{Disk Read}} + \underbrace{\sum_{j=1}^{m} t_j}_{\text{Computation}} + \underbrace{\sum_{j=1}^{m-1} \frac{\sigma_{out}^{j} c_d}{n_j}}_{\text{Data Shuffle}} + \underbrace{\frac{\sigma_{out}^{m} c_w}{n_j}}_{\text{Disk Write}} \tag{C.4}$$

One downside of this approach is that equation C.4 will only hold when all cost and filter parameters are correctly set, which is very difficult to do in such a complex system. It is also required that the system will dedicate a certain amount of task slots to each stage. However, in Spark it is possible that a stage will get paused halfway and its task slots assigned to another stage of a higher priority job. Another downside of this approach is that we do not model queue wait times. If we would include these the model would become even more complicated.

Appendix D

Evaluation Procedure Examples

This appendix extends the evaluation procedure of Section 4.3.3 with examples of the OT-FIFO output trace. The steps below correspond to the steps in the evaluation procedure section.

Example of step 2: A feature correlation matrix is depicted in Figure D.1 for OT-FIFO, showing the absolute correlations between all 37 non-constant features. After cleaning with threshold 0.85, 21 features remain in X (see Figure D.2).

Example of step 4: We visualize X to get more insights into its structure (this is optional). For OT-FIFO we still have 21 features after cleaning. To visualize these, we plot the first two principal components of this dataset, using PCA (see Section 2.5 on PCA). These two components (PC1 and PC2) are linear combinations of the 21 original features. They are are selected by the PCA algorithm such that they cover the largest part of the variance in the original dataset. Figure D.3 shows the first two principal components of OT-FIFO plotted against the query response time (gradient color of the data points). Each line (from left to right) corresponds to query type 4, 2, 3, and 5 respectively. The three lines on the right all correspond to query type 1, but with three different parameter values for the check-in date range. It seems that only knowing the query type can aready give a quite accurate prediction for the response time. It also shows that the predictors will probably not be able to accurately predict queries outside our set with 5 query types. We do not have any data in our output trace for a query which has a PC1 value of 0.5 for example.

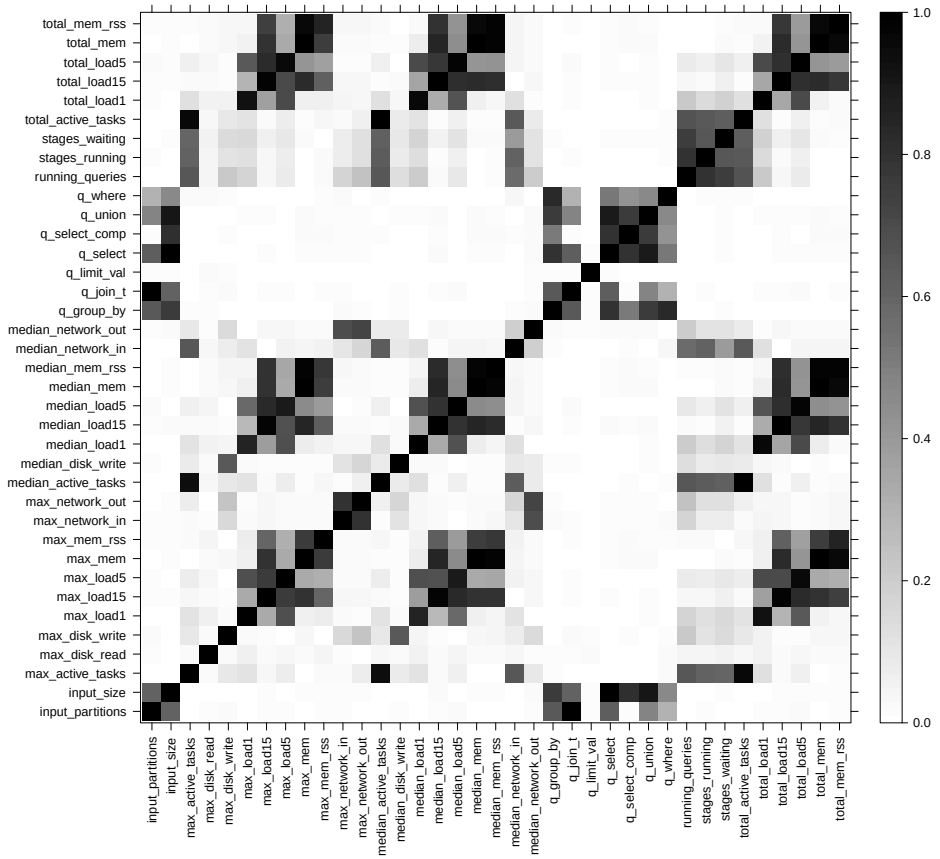


Figure D.1: Feature Correlation Matrix **before** feature reduction. Darker cells correspond to higher correlation.

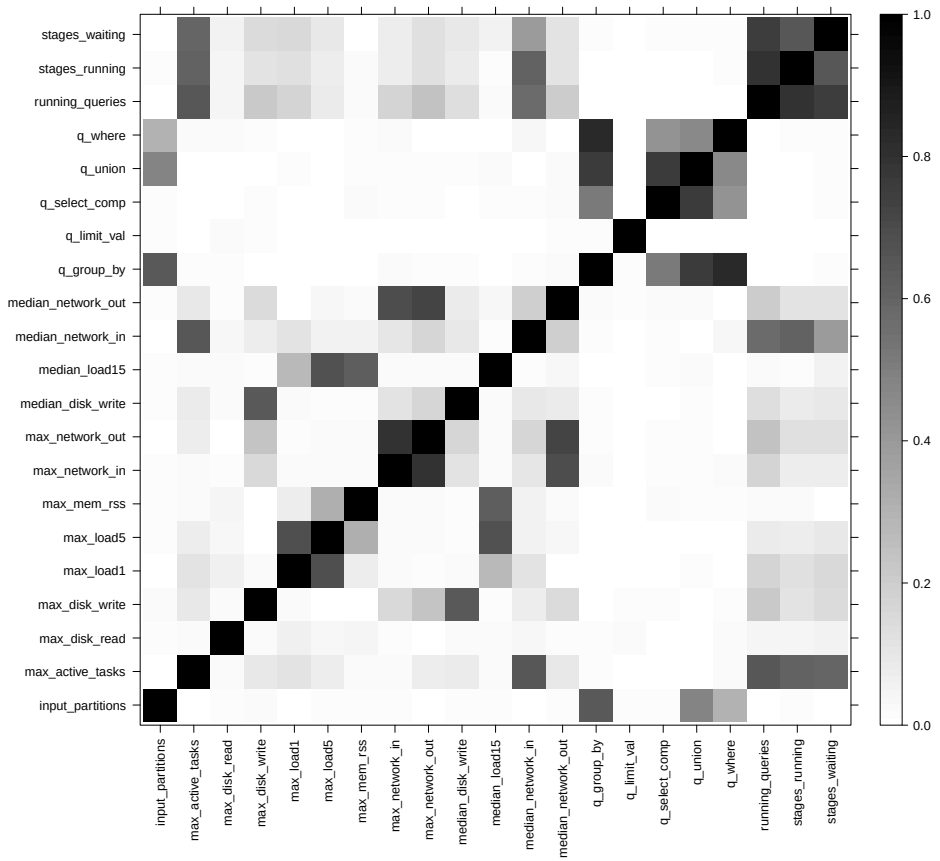


Figure D.2: Feature Correlation Matrix **after** feature reduction. Darker cells correspond to higher correlation.

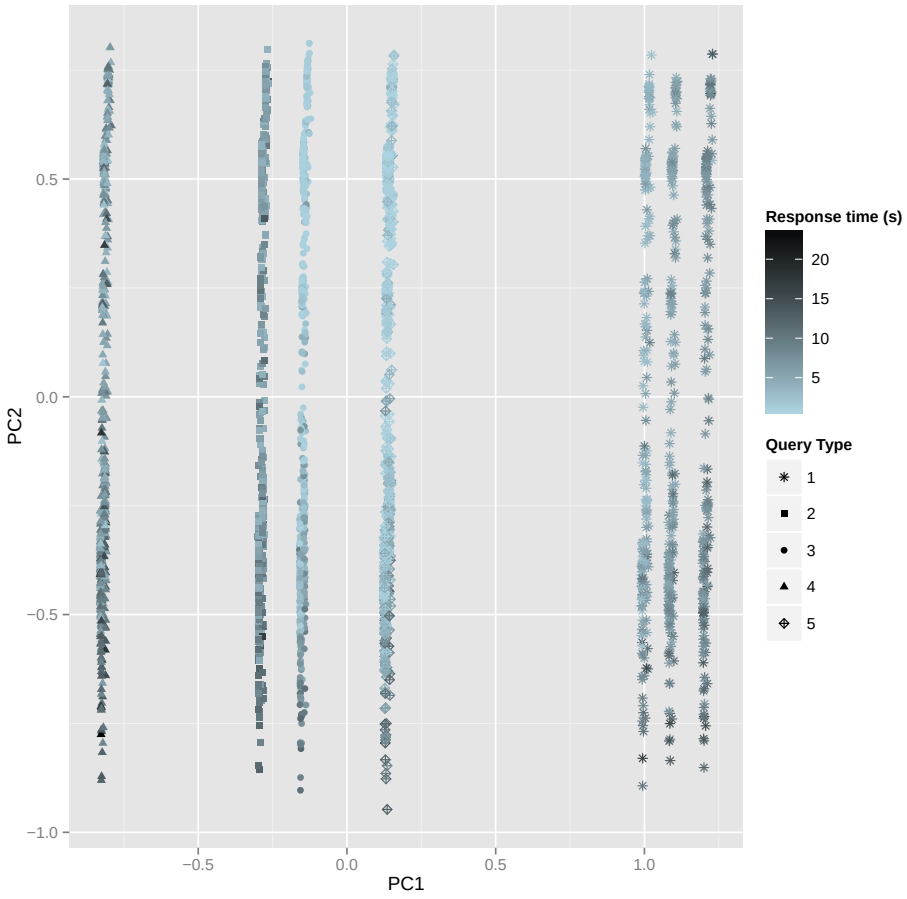


Figure D.3: Principal Component Analysis for the OT-FIFO output trace. Darker colors depict longer response times.