Scalable Unified Data Analytics

by

Alex Watson

Bachelor of Computer Science,
University of New Brunswick, 2012-2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Suprio Ray, PhD, Computer Science
Examining Board:	David Bremner, PhD, Computer Science, Scott Bateman, PhD, Computer Science, Erik Scheme, PhD, PEng, Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April, 2019

Abstract

The volume of data that is being generated each day is rising rapidly. There is a need to analyze this data efficiently and produce results quickly. Data science offers a formal methodology for processing and analyzing data. The goal of processing and analyzing data is to produce actionable insights from this data. The insight gathered from the data enables individuals and businesses to make informed and important decisions. Data science involves a work-flow with multiple stages, such as data collection, data wrangling, statistical analysis, and machine learning. The first part of this thesis evaluates data analytics systems that support the data science work-flow by introducing a data science benchmark, Sanzu.

Further, we believe that data analysts and scientists would want to use a single system that can perform both data analysis tasks and SQL querying, without requiring data movement between different systems. Ideally, this system would have adequate performance, scalability, built-in data analysis functionality, and usability. Currently, relational databases and big data frameworks are used for SQL querying and complex data analysis respec-

tively. However, with relational databases, the data must be completely loaded into the database before performing any analysis, this is increasingly expensive and unnecessary as the volume of data continues to grow. Also, the data must be either analyzed in the database using UDFs, which are often not-optimized or moved to a data science system to be analyzed. On the other hand, big data frameworks tend to entail steeper learning curves and require a larger development effort. Therefore, these big data frameworks are often not the first choice of a data scientist, especially in the early phase of the data analysis, such as data exploration. The second contribution of this thesis is DaskDB, a unified data analytics and SQL querying system that we have developed. DaskDB has similar APIs as a popular Python data science distribution, Anaconda Python. So, it can be easily integrated with most existing data science applications that have been developed with Anaconda Python. Furthermore, in our experimental evaluation, we demonstrate that DaskDB scales out well and has comparable performance as other big data systems with similar capabilities.

Acknowledgements

I would like to thank my supervisor, Dr. Suprio Ray, for first recruiting me to do my Master's and then for the constant encouragement, freedom, guidance, and advice. Dr. Ray really contributed to making the last two years of my life a very enjoyable experience.

Next, I would like to thank my thesis committee, Dr. David Bremner, Dr. Scott Bateman, and Dr. Erik Scheme for their insightful comments and suggestions that helped greatly improve the quality of our work. Dr. Bateman had previously hired me as a summer research student, which gave me my first real hands-on research experience. He also has been very encouraging and helpful throughout the last two years of school.

I would like to thank my fellow labmates, Mahbub, Puya and Yoan for their support, friendship, long interesting off-topic conversations, and constant positive attitudes. They really made the lab a friendly, positive and enjoyable working environment.

I would like to thank the Natural Sciences and Engineering Research Council (NSERC), the New Brunswick Innovation Foundation (NBIF), UNB, and

David Beauvais for financially supporting my studies and research. Also a special thanks to David for providing the initial opportunity to do a Master's and the continued relationship for the past two years.

I am grateful to UNB and the Faculty of Computer Science (FCS), where I have spent most of the past 7 years"studying".My overall experience has been amazing here, starting from my undergraduate studies and all the way up to finishing graduate school. The support of numerous friends and the great experiences I had at UNB have helped in furthering both my academic and professional career.

Lastly, I would like to thank my family: Sophia, Mom, Brittannie, Robyn, Pam, Dad, Chris and Scott for their unconditional love and support. Also a special thanks to Sophia for listening to my presentations, correcting many assignments and being there every step of the way.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

API	Application Programming Interface
DBA	Database Administrator
DAG	Directed Acyclic Graph
$DBMS$	Database Management System
EDA	Exploratory Data Analysis
ETL	Extract, transform, load
GB	Gigabytes
$HDFS$	Hadoop Distributed File System
$RDBMS$	Relational Database Management System
RDD	Resilient Distributed Dataset
SF	Scale Factor
SQL	Structured Query Language

Chapter 1

Introduction

The digital information revolution is accelerating the growth of data. Data is everywhere and a significant part of our daily activities generate data. These activities include online shopping, browsing and searching, and even offline tasks like medical checkups at a clinic or check-out at our local grocery stores. Besides human activities, a large amount of data is generated from machine to machine (M2M) interaction. However, most of the collected data is raw (that has not been processed for use). To extract value from the data, it is necessary to process and analyze the data to enable the creation of actionable insights. With the recent advances in data collection and storage technologies, data analysis has become widely accepted as the fourth paradigm of science [38].

1.1 A Data Science Benchmark

Data science provides formal methods and techniques for processing and analyzing data. Data science is still evolving as a field and it derives skills from different disciplines, such as math, statistics and computer science. The data science process is characterized by a workflow with a feedback loop (described in Section 2.1.1). In this workflow different tasks are pipelined, which resemble the stages in the "big data pipeline" as described in [41]. However, data science not only includes notions of big data technical challenges, but also considerations that might arise even with smaller datasets [20].

To derive value from raw data, it needs to be cleaned, analyzed, manipulated, stored and retrieved. These actions can be performed in different ways. Historically, data were stored in the database, retrieved when needed and then operations were performed on it. This can be very useful in many situations. However, when there are large amounts of data that need to be analyzed together multiple times, this can result in inefficiencies. Some of these inefficiencies are due to constantly retrieving and storing data from the database. The others are due to a lack of support for one or multiple stages of the data science workflow. To address these challenges, a number of data systems and frameworks have been developed to analyze data. These data systems, libraries and platforms, are constantly evolving. The variety of systems offered by current commercial and open-source data analytics systems, differ significantly in terms of available features, functionality and

storage capacity. In computing, the purpose of a benchmark is to run one or more computer programs to assess the relative performance of an object, by running a number of standard tests against it [24]. Benchmarks have been developed to evaluate some of the big data systems ([12, 28, 27, 31, 48, 86]). They focus on either business model queries or more specific tasks and implementations, such as popular machine learning algorithms, genomics algorithms and smart grid analytics. However, to our knowledge, there is no benchmark to evaluate the functionality and performance of the systems for doing data science. It is unclear what system does this best, because there is no industry standard for evaluating or comparing the data science systems that are commonly used for these kinds of analyses.

To address this need, we introduce a data science benchmark called Sanzu. Our benchmark is intended to serve as a basis for an industry standard benchmark for evaluating systems and libraries for doing data science. The benchmark includes two components: a micro benchmark and a macro benchmark. The micro benchmark consists of several task suites that closely follow the stages in the data science workflow, including data loading, data wrangling, descriptive and inferential statistical analysis, and machine learning. For instance, Sanzu looks at the capacity for the systems to do simple statistical tasks, such as filtering and finding the central tendency, to more complex tasks with machine learning, such as linear regression and K-means clustering. It also includes time series tasks owing to its importance in different industries. The intent of the macro benchmark part of Sanzu is to incorpo-

rate applications that are representative of real-world data science use cases. To that end, the macro benchmark includes some real-world tasks in the field of sport analytics, particularly, ice hockey. It also includes a smart grid analytics application, with a focus on demand curves. Sanzu is an open-source project and the source code is publicly available [1].

We use Sanzu to evaluate five popular data science frameworks and systems. The five data analytic systems are R [72], Anaconda Python [70], Dask [14], PostgreSQL (MADlib) [69] and PySpark [83]. The research focus of this work is on the performance of each individual system in completing each particular task. The benchmark aims to expose strengths of the systems as well as particular limitations of each system. The micro benchmark tasks were first evaluated with a small dataset to ensure consistency of results across all systems. Then they were tested with three different datasets by varying the scale factor (SF: 1, 10, and 100, where SF 1=1 million rows) to evaluate their scalability on a single machine.

1.2 A System for Unified Data Analytics and SQL Query Processing

Through the process of evaluating data science systems with Sanzu, we found that some of these systems lack support for specific data science tasks and most of them do not scale well with larger data sets. These findings led us to

[1]Sanzu data science benchmark: http://bigdata.cs.unb.ca/projects/sanzu

think about how we could improve support for functionality, usability, and scalability for data science tasks within some of the systems. In the context of big data, the four V's (volume, variety, veracity, and velocity) are used to describe the increased complexity and challenges in dealing with the rising growth of data. Efficiently analyzing data is becoming increasingly difficult, as these data systems have to ensure scalability as well as provide a wide range of data analysis functionalities. We believe that there are scopes for significant improvement, as they lack one or more of the following: usability, scalability, performance or built-in data analysis capabilities. A data system must be able to scale out to a cluster of machines and support "big data" workflows. A data system needs to offer high usability for both novice and experienced analysts in order to support fast learning and shorter development cycles. Typically, usability refers to the ease of use and learnability of a system, and more formally, it is the ability of the users to achieve predetermined objectives with effectiveness, efficiency, and satisfaction in a quantified context of use [84]. Further, in the context of data analysis, we believe this implies that the system has support for one or more popular data analysis languages, such as Python, R, and SQL. With regards to the built-in capabilities, we believe that it is essential to be able to conduct exploratory data analysis (EDA) effectively. EDA refers to the process of performing initial investigations on the data in order to discover patterns, to spot anomalies, and to test a hypothesis [91]. Typically, this is done by generating summary statistics and graphical representations. Furthermore, through the process of

EDA, it is often found that parts of a dataset may be "bad data", and this "bad data" needs to be "cleaned" using data wrangling techniques before the data can be analyzed any further [53]. We believe that a data system that supports "big" data science tasks should have all of these characteristics: built-in data analytics functionality, scalability and performance, and usability.

At present, there are two primary categories of systems used for data analysis. The *first category* includes dedicated data analytic systems comprised of language frameworks, such as R, Python, and MATLAB. Usually, these frameworks offer a wide range of library APIs supported by a large community. NumPy [61], Pandas [67], and scikit-learn [78] are all packages belonging to Python's data science portfolio. In the *second category*, there are database management system (DBMS). Traditionally, a DBMS is utilized for data management tasks and SQL (a powerful declarative query language) is used to perform such tasks within DBMS. Since SQL is widely popular, increasingly data analytics and machine learning applications are being developed that are SQL-centric. An example of this is the MADlib analytics library [37] for PostgreSQL database.

However, both language framework and DBMS approaches have some drawbacks. The drawback with the language framework is that the data must be retrieved from DBMS into user space, if data resides in DBMS tables, which can happen in many cases. For large volumes of data, this can impose significant overhead. Moreover, unlike server-class machines operating DBMS

engines, user systems are usually not equipped with large memory capacity or processing power. Therefore, users are often forced to use smaller data sets that can comfortably fit inside the user system's memory.

The drawbacks of the DBMS approach is that typically SQL offers limited data analysis capabilities, which include join and aggregation. For more advanced analytics involving statistical analysis and machine learning, the application developer must either develop new User Defined Function (UDF) or use an existing UDF. The other possibility is to load the data from the DBMS into an outside language framework based data analytic system. The issues with UDFs is that in most DBMS systems the UDFs are not optimized. Typically the only way to interact with a UDF is through its input parameters and final output results and this imposes constraints in terms of the number and data types of its input and output [21]. This results in them being difficult to debug and to incrementally develop. Hence, many users tend to use outside language framework based data analytic systems, which inherit their limitations, as discussed above.

In response to the drawbacks of the two categories of systems for data analytics, a third category of systems have emerged in recent times. They are usually classified as "big data systems". Spark [83] and Hive [89] are prime examples of these. However, the main drawback with these big data systems frameworks is that they have complicated APIs and hence entail steeper learning curves. Thus for the general user, it takes much longer and more effort to become proficient with these systems, compared to the other two

categories of systems that perform data analytics.

The workflow involving executing SQL queries and performing data analytics on large datasets is where we think most existing systems lack the desired characteristics mentioned earlier. Given the vast ecosystem of SQL applications and large user base, we think that having SQL querying capabilities in a data science platform exposes their analytic potential to a wide variety of end users. These considerations led us to focus our attention on Dask. Dask is a system that performed the best on our Sanzu benchmark in terms of functionality and scalability on a single machine. Dask is a parallel computing library for analytic computing that extends and supports similar APIs as popular Python data-science packages NumPy, Pandas, and scikit-learn. Dask extends the ability of these packages to scale up to larger datasets on a single machine, as well as scale out to a cluster of machines.

The second part of this thesis presents our system DaskDB, which we developed. It combines the dedicated analytics functionalities of Dask with SQL querying capabilities. DaskDB supports all Dask features and enables SQL querying on raw data in a data science friendly environment, which has access to all of the packages available within the Anaconda Python ecosystem. We believe that DaskDB addresses the issues that existing data science systems face when performing SQL querying and data analytics together. Python and SQL are among the most popular data science languages used for data science, according to a survey from KDNuggets [43]. Also, we believe that DaskDB possesses all of the desired attributes (mentioned earlier):

performance, scalability, usability and built-in functionalities. We perform an extensive evaluation to show that our system remains competitive against current state-of-the-art big data systems. We present an evaluation that runs a subset of the micro benchmark from Sanzu and a subset of queries from the TPC-H benchmark on a cluster of 4 machines. Finally, we discuss why we believe that our system has much higher usability than the other current state-of-the-art big data frameworks with similar capabilities.

1.3 Contributions

The main contributions of this thesis are as follows:

(1) Sanzu: A Data Science Benchmark

- Sanzu is a novel data science benchmark that encapsulates the full workflow of common data analytic tasks. It includes a micro benchmark to test individual operations and a macro benchmark to represent real-world use cases. It is a performance evaluation of five popular data science systems: R, Anaconda Python, Dask, PostgreSQL (MADlib) and PySpark.

(2) DaskDB: A Unified Big Data System for Supporting SQL Queries and Data Analytics.

- We introduce DaskDB, a unified big data system that combines a data analytic system with SQL querying capabilities. We describe the approach we take to designing DaskDB and discuss the advantages it

has over current big data systems with similar capabilities. Lastly, we benchmark our solution DaskDB against Spark using TPC-H and several data science tasks from our Sanzu benchmark.

1.4 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, we introduce and discuss data science, data analytics, big data, and SQL querying and why they are important. We also introduce the systems we evaluated throughout our work. In Chapter 3, we discuss benchmarks related to data science as well as various other approaches that combine SQL querying and data analytics. We then introduce Sanzu in Chapter 4, we go into detail on the key features within the benchmark itself, such as the systems we evaluated, the datasets, the implementation, the experimental setup, the evaluation, and results. In Chapter 5 we introduce DaskDB and discuss the methodology, system architecture and evaluate and benchmark DaskDB against Spark. We finish with a discussion of possible future directions to build on the work, and a conclusion of the findings in Chapter 6.

Chapter 2

Background

In this chapter, we define data science and discuss why benchmarks are so important. Also, we introduce other key data science concepts and aspects. Lastly, we introduce the systems evaluated and benchmarked throughout our work.

2.1 Data Science and Data Analytics

Data science is still an evolving field and it provides tools, techniques and methodologies to turn data into knowledge [18]. Joel Grus has defined a data scientist as someone who extracts insight from messy data [33].

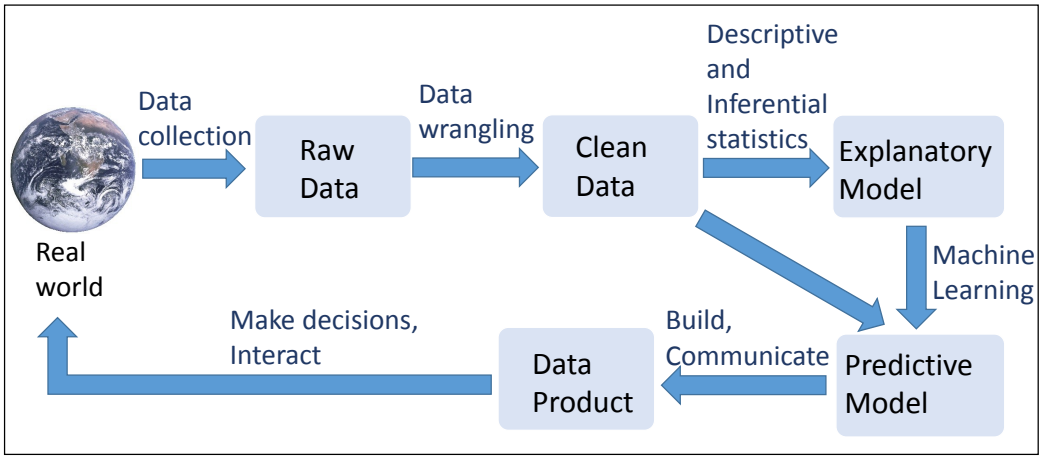


Figure 2.1: Data Science Workflow

2.1.1 Data Science Workflow

Data science draws various techniques and skills from different fields, including math and statistics, machine learning and computer science. In addition, a data scientist may possess domain expertise in her own areas. Invariably, data science tasks follow a series of steps, from data gathering to building analytical models. The data science process [77] can be conceptualized as a workflow.

As shown in Figure 2.1, the first step of the workflow is the collection of data. The second step is data wrangling, which involves loading, cleaning, transforming, and rearranging messy data for easier access and analysis. This data preparation step is vital for subsequent downstream processing. According to some estimates [62], on average about 50% to 80% of the time in the data science workflow is spent in data wrangling. The next step is to apply statistical analysis on the clean data to build explanatory models. The

statistical analysis could be descriptive or inferential. Descriptive statistics quantitatively describe the characteristics of a dataset. Essentially, it is used to characterize a population. Important descriptive statistical properties include measures of central tendency and measures of dispersion. Inferential statistics is used to make generalization about the population based on representative subsets of the population called samples. A common approach to investigate a claim about the population by analyzing samples is hypothesis testing. The statistical analysis techniques can be used to build explanatory models. An explanatory model establishes a causal effect from the observed data. On the other hand, a predictive model is used to predict new observations and can be constructed by machine learning algorithms. The choice of a particular machine learning algorithm depends on the type of problem at hand.

Once there is a model one could communicate our results or build a "data product", such as an email spam filter. The data product is usually incorporated back into the real world, thus enabling user interaction with that product. This process helps generate more data, and in turn creates a feedback loop. Schutt and O'Neil [77] noted that this feedback loop is a key distinguishing feature of the data science process.

2.1.2 Why a Data Science Benchmark?

We believe that a benchmark for data science should faithfully follow the data science process. Reflecting the ground reality, it should contain tasks from

every step of the workflow (as in Figure2.1). As described in Chapter 3, there are a few benchmarks that focus on a few domain specific analytics applications. However, no benchmark exists that captures the data science process. Given the growing importance of data science, we have developed our benchmark. We call it "Sanzu", which is a mythical river of life in Japanese tradition, to symbolize the workflow in data science.

2.1.3 Data Analytics

Data analytics is the process of examining large and varied data sets to gather relevant insightful information from the data. Before examining any data, a system must first deal with the collecting, cleaning and wrangling the data. First, a system must read it from various data formats and then perform common data wrangling and cleaning techniques such as sorting, filtering, merging, aggregating and removing duplicates. After collecting and cleaning, data analysis tasks can be performed to derive insights, which include uncovering hidden patterns, unknown correlations, trends and any other useful information that can help organizations and businesses make informed decisions. Common categories in analytics are descriptive statistics (central tendency, measures as dispersion), distribution and inferential statistics (correlation, skewness), time series analysis (auto-correlation and exponential weighted moving average) and machine learning (regression, clustering and classification algorithms) [93]. Formally, data analytics involves applying an algorithmic or mechanical process to derive insights. Data analytics

are used by big companies to make smart, informed decisions. Thus, having feature-rich data analytic toolkit is an essential part of having a productive and usable big data system.

2.1.4 Big Data Systems

With the amount of data that needs to be analyzed in this world, "Big Data" analytics is often necessary to deal with these large amounts of data. Big data is a term generally used to refer to data sets that are too large or complex for traditional data-processing systems. The big data systems were developed to deal with these large and more complex datasets and computation heavy analysis and algorithms. Thus, more time and effort is required to run, configure, and use these big data systems and frameworks, which in turn often requires a much steeper learning curve (lower usability) compared to the traditional data analytic systems. Typically, these big data frameworks are able to scale up to a cluster of machine that allows them to perform more computation heavy queries and query larger amounts of data.

2.1.5 SQL Querying

SQL is a standard going back to the 1970s and is a popular way to get information out of relational database systems. The SQL language is written to retrieve data or otherwise interface with a conventional RDBMS (Relational Database Management Systems). SQL is widely used in business and in

other types of database administration. It is the default tool used to create, alter table data, retrieve data or manipulate an existing data set. SQL has remained very popular over the years due to its simplicity and power and continues to remain essential in both academia and industry.

2.2 Systems Evaluated

Next we describe popular data analytics systems that we evaluate in our work. All of these systems are evaluated by the Sanzu benchmarks. Also, Spark and our proposed system DaskDB were compared in 5.5.

1. Anaconda: The Anaconda distribution is an open-source platform for the languages Python and R for data science and machine learning available on Linux, Windows, and Mac OS. It has over 11 million users worldwide [70] and it has a simple installation and configuration procedure that gives the user access to over 1500 data science packages for Python and R. It allows for easy management of libraries, dependencies and virtual environments with Conda [1]. Furthermore, they have functionalities for machine learning with packages such as scikit-learn [78] and TensorFlow [88]. Also Anaconda has packages that analyze data with scalability and performance, such as Dask[14], NumPy [61], Pandas[67], and Numba. Lastly, it has visualization packages with

[1]Conda: https://conda.io/

Matplotlib [52], Bokeh [2], Datashader[3], and HoloViews[4] . The packages used throughout this benchmark are Pandas, NumPy, Matplotlib and scikit-learn. Pandas is a package that provides data structures and tools for data analysis. NumPy is a package for scientific computing with Python. With a powerful N-dimensional array object, it supports linear algebra, and other capabilities. Matplotlib is a Python plotting library to produce figures and graphs. Scikit-learn is a machine learning library that is built on NumPy.

2. Dask: Dask [14] is a parallel computing library for analytic computing and part of the Anaconda Python ecosystem. It is composed of two components: dynamic task scheduling and "big data" collections (data structures). This collection includes parallel arrays, dataframes, and bags (to support unstructured data). Dask extends common data-science packages Pandas, NumPy and scikit-learn. Dask represents computation as task graphs with data dependencies and it supports out-of-core execution. It can be set up and run on a single machine or on a cluster with 1000s of cores using the extended 'dask.distributed' library. 'Dask.distributed' [15] is an extension of Dask that is needed to extend to a cluster of machines. Dask.distributed is necessary component implementation of DaskDB. Dask.distributed is composed of three main components, the client, scheduler, and worker. The client is the

[2]Bokeh: https://bokeh.pydata.org/
[3]Datashader: http://datashader.org/
[4]HoloViews: https://holoviews.org/

primary entry point for users of Dask. The scheduler coordinates the actions of all worker processes and the concurrent requests of clients. The worker has two main functions: to compute tasks as directed by the scheduler and store and serve computed results for other workers or clients.

3. PostgreSQL (with Apache MADlib): PostgreSQL [69] is an open-source object relational database management system. PostgreSQL is accompanied by MADlib [37], an in-database machine learning toolkit which performs statistics and analytics. It provides many SQL-based algorithms for machine learning, data mining and statistics, which run at scale within a database engine.

4. R: R is a language and environment for statistical computing and graphics [72]. R has the ability for data handling and storage facility, data manipulation, statistical analysis, time series analysis, and machine learning algorithms. R is extensible and includes a wide collection of open-source packages.

5. Spark: We chose to implement Spark using PySpark. PySpark is the Python API from Apache Spark [83]. Spark is a main-memory distributed data processing platform. It provides an application programming interface centered on a data structure called the resilient distributed dataset (RDD). Spark supports data manipulation, statistical analysis and machine learning analysis. Spark Core is the foundation

of the overall project. It provides distributed task dispatching, scheduling, and basic I/O functionality. Also, the package SparkSQL provides the ability to query using SQL Strings.

2.3 Summary

In this chapter, we introduced a definition of data science. We discussed the importance of benchmarks and introduced several relevant aspects. They included data analytics, big data systems, and SQL querying. Finally, we introduced the systems we evaluate throughout our work: Spark, Dask, Anaconda, PostgreSQL (MADlib) and R.

Chapter 3

Related Work

3.1 Introduction

In this chapter, we discuss the current work that is related to Sanzu and DaskDB respectively. In Section 3.2, we discuss current data science benchmarks and benchmarks that closely relate to data science. In Section 3.3, we describe and discuss several systems that combine SQL querying and (big) data analytics.

3.2 Data Science Benchmarks

To our knowledge, no benchmark exists that can be termed as a data science benchmark. However, there are several categories of benchmarks that are related. Relational database benchmarks belong in the first category. In

the second category are the big data systems benchmarks. Lastly, there are domain specific analytics benchmarks, these however, tend to focus on specific tasks, such as smart grid or genomics problems.

3.2.1 Database Benchmarks

There are a number of commercially available and free database benchmarks. The most well known is the Transaction Processing Performance Council (TPC) [90]. It is devoted to defining transaction processing and database benchmarks for diverse workloads. TPC-C is an On-line Transaction Processing (OLTP) benchmark to measure transactional throughput. TPC-H is a decision support benchmark characterized by the execution of complex SQL queries. Some of these benchmarks have evolved over the years. TPC-DS is TPC's latest decision support benchmark with a complex snowflake-like data model [45].

Besides transactional and decision support workloads, there have been attempts to develop database benchmarks for other kinds of workloads. For instance, Jackpine [75] is a benchmark to evaluate spatial database performance.

3.2.2 Big Data Systems Benchmarks

The growing popularity of big data systems, such as Hadoop [35] and Spark [83], led to the development of several benchmarks. These benchmarks address

the different aspects of big data such as the 4 V's: volume, variety, veracity and velocity. YCSB [12] is arguably the first among the big data benchmarks. The focus of this benchmark was to evaluate the performance of emerging NoSQL data stores and also to compare them against traditional relational databases. In the original paper, the authors executed three different workloads with four different data systems: HBase, Cassandra, PNUTs, and MySQL. TeraSort (or GraySort) [31] is another well-known benchmark to be run on commercially available hardware. It is a micro benchmark that sorts a large number of 100-byte records with the goal of stress testing processing, and storage I/O subsystems. BigBench [28] is a popular end-to-end big data benchmark. It provides a data model and synthetic data generator. Its data model was adopted from the TPC-DS benchmark, and extended that with semi-structured and unstructured data components. Recently, a modified version of BigBench, called BigBench V2 [27], has been introduced. It includes a new data model and an overhauled workload, that includes Web-logs modeled as key-value pairs. Perhaps the most recent benchmark is [5], they evaluate several big data systems using the TPC-H, TPC-DS and TPCx-BB benchmarks.

3.2.3 Domain Specific Analytics Benchmarks

Mehta et al. [54] evaluated several systems with scientific image data processing tasks. The uses cases were from neuroscience and astronomy. The systems they evaluated were Spark, Myria, Dask, SciDB and TensorFlow.

In [86] a genomics benchmark, GenBase, was developed to measure systems performance on data management and data analytics tasks. The systems that were evaluated were R, PostgreSQL, a commercial column store database, SciDB, and Hadoop. A benchmark to evaluate common smart grid analytic tasks was developed by Liu et al. [48].

3.3 Standalone Data Analytic Systems

Next, we discuss well-known systems that perform data analytics, SQL, or SQL-like queries. We look at dedicated data analytics systems, in-database analytic systems and big data frameworks.

3.3.1 Dedicated Data Analytics Systems

Dedicated analytic systems are used to perform data analysis tasks and execute queries that process the whole dataset or extensive subsets. These operations are often computation intensive and long-running. Some popular open-source data analytic systems and tools include, ELKI [76], Orange [16], and Tupleware [13]. Some popular commercial data analytic systems include Tableau [85], RapidMiner [40], MATLAB [51] and Sisense [82].

Many dedicated open-source data analytic tools, systems, frameworks, and environments traditionally use the R ecosystem [72]. More recently, Python has increased in popularity mostly because of the Anaconda distribution [70]. The Anaconda distribution contains a multitude of Python and R data

science and analytics packages. R is a language and environment for data analytics that provides many packages and libraries for common analytical tasks such as statistical computing, visualization, and machine learning. Python also has many data analytics libraries such as Pandas [67], SciPy [79], Matplotlib [52], and scikit-learn[78], these libraries are also readily available through the Anaconda distribution. These libraries provide machine learning, data visualization, scientific computing to this ecosystem. These systems are readily available, are easy to learn and have a lot of documentation. For these reasons, they are heavily used among data scientist for data analytics, and implementations of new algorithms are often integrated. Furthermore, these libraries also offer single-threaded and multi-core computation. R has a package 'parallel'[1] and Python has several packages that offer parallel computation support including Dask [14]. Further discussion in 5.2.1 explains how data analytic systems typically fall short in terms of scalability.

3.3.2 Big Data Analytic Frameworks

Big data frameworks were initially introduced to handle one or more of the 4Vs (volume, variety, velocity, veracity) of 'big data'. These frameworks typically are cluster-computing frameworks and have scale-up and scale-out capabilities. Programming interfaces like Java, SQL (or a SQL-like declarative language like Pig[2]), Python, Scala, and R are supported depending on

[1]https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf
[2]https://pig.apache.org/

the framework. The frameworks allow for the programming of entire clusters with data parallelism and fault tolerance. The most well-known big data framework is Spark [94]. Spark supports SQL like queries with SparkSQL [6] and machine learning with MLlib [55]. Some other well-known big data frameworks include Hadoop [35], Hive [89], HBase [36] and Storm [1]. Some drawbacks of big data frameworks include more complicated development and steeper learning curve than most other analytics systems and the difficulty in integration with SQL based applications (DBMS). Further discussion about these issues is in Section 5.2.3.

3.3.3 In-Database Analytics

If data is already loaded in a DBMS, they would likely need to be moved back and forth between external analytics environments and the DBMS, resulting in considerable inefficiencies and performance overheads. The performance becomes increasingly worse as the datasets grow due to the I/O constraints. Thus, an increasing number of the major databases now include data science and machine learning analytic tools. Most modern DBMSs have data analytics capabilities and functionalities. They include PostgreSQL (MADlib) [69], MySQL [57], Oracle Database [64], and SAP HANA [26]. There have also been parallel databases developed such as Pivotal Greenplum [32] derived from PostgreSQL and Amazon Redshift derived from MySQL [34]. While modern DBMSs do incorporate analytical components, interacting with a DBMS to implement analytics can be troublesome [25]. The end user requires

the knowledge of database specific languages such as SQL. Even though the SQL is a mature technology, database languages are not rich enough to be suitable for extensive data analysis. The drawback of most modern DBMS solutions is that they typically use UDF (User Defined Functions) to support analytics functionalities. A database system treat a UDF as a black box because no optimization can be performed on it, as it may execute any external code written in R, Python, C, C++, Java or T-SQL code.

There have been several in-database analytics solutions over the years, both open-source and commercial. One of the most popular open-source in-database analytic systems is Apache MADlib [37]. MADlib is an in-database machine learning toolkit supported by PostgreSQL and Greenplum, which performs statistical and analytics tasks. It provides many SQL-based algorithms for machine learning, data mining, and statistics, and runs within a database engine. It provides support for analytical algorithms as user-defined functions (UDF) written in C++ and Python that are called from SQL queries. Other academic efforts include MauveDB [17], which integrates regression models into a RDBMS and [23], which integrates and supports in-RDBMS analytics and integrates and supports least squares regression models.

As for commercial databases, Microsoft SQL Server [56] has machine learning services, predictive analytics, and data science engine that can execute R and Python code within a SQL Server database as stored procedures. SAP HANA [22] offers multi-threaded C++ implementations of analytical algorithms with an extension to a predictive analytics library [26]. SAP HANA

uses UDFs to perform this functionality with parallel execution along with relational operators. IBM DB2 warehouse formerly DashDB [47], allows for optimizations for parallel execution of relational operators and UDFs. Lastly, Alteryx [3] and tableau [4] are business analytics solutions, in-database analytics also and uses well-known in database query compiler of Hyper [60].

Even with all these optimizations and progress made to increase functionality and performance, the same issue still persists in DBMS. They have the constraint of ETL (extract, transform, load). The data must be completely loaded into the database to run any analysis. ETL is a time-consuming process and is not necessary in a growing number of cases. This is further discussed in Section 5.2.2.

3.4 Integrating, Data Analytics, DBMS and Big Data Frameworks

There have been several attempts at creating more efficient systems that perform data analytics and run SQL and SQL-like queries. These solutions take vastly different approaches, but generally combine two or more of either dedicated data analytic systems, DBMS or big data frameworks.

[3] https://www.alteryx.com/

[4] https://www.tableau.com/

3.4.1 Database Connectors

Database connectors are typically packages or libraries that integrate data analytics or big data framework systems directly into the database. Some examples include RIOT-DB [95], a system that makes R programs I/O-efficient and uses a database system as a backend. Others include PostgreSQL with RPostgreSQL [5] and Oracle with RORACLE [6]. Others integrate with Spark ecosystems, such as Oracle (OTA4H) and IBM DB2 [47]. IBM DB2 also connects with the Python interface via ibmdbpy[7] library, for database administration and data analytics and uses data analytics library Pandas and machine learning library scikit-learn. Lastly, [42] is a big SQL system that supports UDFs and any big machine learning system that can support the Hadoop *InputFormat* [8]. The disadvantage of the "database connectors" systems is ETL, as the data needs to be loaded into a database and this can be a waste of resources depending on the workflow.

3.4.2 Directly Integrated

The "directly integrated" solutions integrate two or more system types together into one system. AIDA [21] integrates the Python client directly to use the DBMS memory space, eliminating the bottleneck of transferring data.

[5]https://cran.r-project.org/web/packages/RPostgreSQL/RPostgreSQL.pdf
[6]https://rdrr.io/cran/ROracle/
[7]https://pypi.org/project/ibmdbpy/
[8]Hadoop InputFormat: https://hadoop.apache.org/docs/r2.7.5/api/org/apache/hadoop/mapreduce/InputFormat.html

In [44] the authors present a prototype system that integrates a columnar relational database (MonetDB) [9] and R together using a same-process zero-copy data sharing mechanism. Lastly, in [50] they attempt to combine R, MapReduce (a well-known big data framework), and DBMS technologies to try to take advantage of each individual system. Again, the drawback of the "directly integrated" systems is ETL, since the data needs to be loaded into a database and this can be a waste of resources depending on the workflow.

3.4.3 Cross-Platform Data Processing

In cross-platform data processing, front-end frameworks are decoupled from the back-end execution engines, which allow workflows to be written once and mapped to many different systems. In Musketeer [29] different systems can be combined within a workflow and existing workflows seamlessly ported to new execution engines. In RHEEM [4] there is a cost-based optimizer in the executor to find the most efficient platform for specific tasks. Cross-platform solutions are still in the infancy as they only show a few use cases. These are not viable options in practice at this stage for EDA (exploratory data analysis). They still need to address several issues such as inter-platform communications and fault tolerance while moving data across platforms.

[9] MonetDB: https://www.monetdb.org/

3.4.4 Middleware and Polystore Systems

Middleware and Polystore systems expose multiple storage engines through a single middle interface, which allow the system to pick which engine performs best on specified workflow. BigDawg [63] middleware chooses the best engine for each subquery and optimizes the joins. In [49] they propose a polystore that relies on query augmentation, and a data manipulation operator, it allows results to be stored locally with related data stored on the polystore. The constraint of the "polystore" systems is ETL, the data needs to be loaded into a database(s) to do any sort of EDA, which is a waste of resources.

3.4.5 In-Memory Optimizations

These solutions aim to improve performance by optimizing the workloads in-memory on single machines. Level-Headed [3] is an in-memory query processing engine that compromises between relational and statistical systems, by unifying their query workloads. It uses a novel (Worst-Case Optimal Join) WCOJ query architecture. Weld [65, 66] and Tupleware [13] use a common intermediate representation to combine SQL, data analytics, machine learning, and graph analytics into one workload. Weld can be integrated incrementally into existing frameworks like TensorFlow, Apache Spark, NumPy and Pandas without changing their APIs. These systems focus more on optimized code that runs on the applications in-memory data.

3.4.6 DBMS Query Plan Optimizers (Optimizing UDFs in Query Plan)

These solutions aim to optimize the UDFs by converting them into relational operators so that the optimizations already present in the DBMS compiler can optimize the UDFs. In both, [68] and Froid [73] the authors use relational main-memory database systems and they are capable of executing analytical algorithms by adding a new operator in which the UDFs can be optimized by the DBMS in their query execution plan. In Froid, they focus their optimizations on scalar UDFs and in [68] they extend with optimizing predictive analytics algorithms such as PageRank, KMeans and Naive Bayes.

3.4.7 DBMS SQL Query Code Generation

In these solutions, they convert SQL into executable code. LegoBase [80] is a query engine written in Scala. It first converts SQL statements into Scala code and then compiles it into low-level C code. SimSQL [10] compiles SQL queries into Java code that run on top of Hadoop. Nagel et al. [58] integrate database queries with a Language Integrated Query (LINQ) [10] to convert SQL and LINQ code into C and C# code. SimSQL and LegoBase are systems that focus on running SQL queries and do not consider other aspects of data analytics such as machine learning. The Q*cert [8] platform includes some support for SQL and OQL, and for code generation to Java, Spark and

[10]https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

Cloudant. Q*cert is not integrated with any big data system, but the idea is similar to the code generation component of DaskDB called DaskPlanner, explained in Section 5.3.2. None of these code generations system are viable solutions to do a full data analytic workload or EDA.

3.4.8 Creating a New Big Data Framework

The Myria [92] stack is a big data framework for big data management and analytics created from scratch. It was built to test novel database research ideas. It was created because they wanted the freedom to decide on all components of the systems design rather than extend other big data systems such as Hadoop [35] or Spark [94]. In our system DaskDB, we extended Myria's Relational Algebra compiler (RACO). It was used to create an initial physical plan, described in detail in Section 5.3.2. Myria has recently concluded development and it is not as mature compared to other big data frameworks.

3.4.9 Other Implementations

MLOG [46] program is similar to a SQL program but it extends relational algebra relations to linear algebra over tensors [11]. MLOG supports machine learning models that are not currently available with in-database analytics systems and can read directly from a file. However, MLOG is in its initial stages and is not currently implemented in any DBMS.

[11] https://www.tensorflow.org/guide/tensors

3.4.10 Why these Solutions Fall Short

In the following section, we discuss the limitations of existing systems that attempt to unify SQL-querying and data analytic systems. The systems described in the Sections 3.4.1, 3.4.2, 3.4.4, and 3.4.6, all store data in DBMS, and so their main constraint is ETL. The data needs to be loaded into a database(s) to do any kind of EDA, which is a wasteful and time-consuming step in EDA, as the analyst usually does not know in advance how much or if any data is not useful. The systems mentioned in Sections 3.4.3, and 3.4.9 are still in their infancy and lack EDA functionality to make them viable options. The systems described in Section 3.4.7 also lack EDA capabilities, while Myria big data framework, mentioned in Section 3.4.8, is no longer in development.

3.5 Summary

In this chapter, we presented different benchmarks related to data science, described popular systems that perform analytics and SQL querying, and explored various systems that unify and improve SQL querying and data analytics. We then discuss the most popular dedicated analytic systems, big data framework, and DBMS, that perform data analytics.

Chapter 4

Sanzu: A Data Science Benchmark

4.1 Introduction

This chapter introduces and describes our data science benchmark called Sanzu [93]. This benchmark contains two components: a micro benchmark and a macro benchmark. With Sanzu, we evaluate five popular data science frameworks and systems. We test the performance of each individual system completing each particular task. The micro benchmark tested three different datasets by varying the scale factor (SF: 1, 10, and 100, where SF 1=1 million rows). The macro benchmark tested two representative tasks in the industry with smart grid and sports analytics. We discuss the implementation overview and challenges, as well as the experimental setup. The

Table 4.1: Software configuration

Software	**Version**	**Language**	**Key libraries**
Anaconda	4.4.0	Python	Pandas, NumPy, Matplotlib, scikit
Dask	0.15.1	Python	
PySpark	2.11	Python	
PostgreSQL	9.6	SQL	MADlib
R	3.4.1	R	

benchmark results and discussion examines performance in terms of running time and scalability.

4.2 Systems Evaluated

The five analytics systems that we have evaluated are all publicly available open-source software. The software version, programming language and key libraries of these systems are shown in Table 4.1. We chose these five analytic systems because we believe that they are a good representation of current state-of-the-art tools used in data science. Moreover, they were used in other benchmarks mentioned in Chapter 3. PostgreSQL is a popular relational database with support for machine learning in SQL (MADlib) and it has been used by other benchmarks such as ([48] [75], [86]). R is one of the most popular statistical computing libraries and is also used in other benchmarks [86]. Anaconda is the most popular Python-based data science platform. The libraries we used from Anaconda in Sanzu were Pandas, NumPy, Matplotlib and scikit-learn. Dask extends Anaconda and adds support for out-of-core

execution and parallelism to some of the libraries within Anaconda. Dask was used in the benchmark in [54]. Spark is one of the leading candidates among the big data systems with its distributed data processing framework and it was used in a few benchmarks ([27], [48], [54]).

4.3 The Benchmark

Our benchmark, Sanzu, consists of two different components: a micro benchmark and a macro benchmark. The micro benchmark consists of several task suites, which involve data wrangling, statistical, time series and machine learning analyses. The macro benchmark contains two applications that are modelled after real-world use cases.

4.3.1 Micro Benchmark

The micro benchmark is intended to test the basic tasks across multiple data analytics systems. These tasks are common and are used everyday in solving real-world and industry problems. They are modeled after the data science workflow (in Section 2.1.1). The micro benchmark consists of 6 task suites: basic file I/O, data wrangling, descriptive statistics, distribution and inferential statistics, time series analysis and machine learning. Each suite represents a different stage in the data science workflow. Each suite includes a number of tasks. Usually, some of the operations involving statistical distribution, such as correlation and skew, are considered as part of descriptive

Table 4.2: Micro benchmark tasks and feature matrix (NF=no built-in functionality)

Operation	**TaskId**	**Description**	**Support in the evaluated systems**				
			Anaconda	**Dask**	**PostgreSQL**	**PySpark**	**R**
Basic File I/O							
Read	read	Read from data1	Y	Y	Y	Y	Y
Write	write	Write data1 to a CSV file	Y	Y	Y	Y	Y
Data Wrangling							
Sorting	sort	Sort data based on the column uni in data1	Y	Y	Y	Y	Y
Filtering	filter	Filter the column rand1 of data1, if smaller than a given value	Y	Y	Y	Y	Y
Merging	merge	Merge on column rand1 from both data1 and data2	Y	Y	Y	Y	Y
GroupBy	groupby	Group by column city in data1	Y	Y	Y	Y	Y
Removing Duplicates	remdup	Remove duplicates from column words in data1	Y	Y	Y	Y	Y
Descriptive Statistics							
Central Tendency	centend	Find the mean of uni, mode of rand1 and median of rand2 in data1	Y	Y	Y	Y	Y
Measure of dispersion	dispers	Finds the range of exp and standard deviation of uni column in data1	Y	Y	Y	Y	Y
Rank	rank	Add a new column based on rank of column uni in data1	Y	NF	Y	Y	Y
Outliers	outlier	Remove the outliers of column exp in data1	Y	Y	Y	Y	Y
Scatter Plot	scatter	Draw the scatterplot based on column uni and nor in data1	Y	Y	NF	NF	Y
Distribution and Inferential Statistics							
Probability Density Function (PDF)	pdf	Find the Pdf of the data series on column rand1	Y	Y	NF	Y	Y
Skewness	skew	Calculate the skewness of the data1 series on column rand2	Y	NF	Y	Y	Y
Correlation	corr	Calculate the correlation between uni and exp columns in data1	Y	Y	Y	Y	Y
Hypothesis Testing	hypo	Hypothesis testing (shuffling method)	Y	Y	Y	NF	Y
Time Series							
Exponentially-weighted moving average (EWMA)	ewma	Find the exponentially-weighted moving average on column rand1	Y	NF	NF	NF	Y
Autocorrelation	autocorr	Find the autocorrelation of the nor column in data1 time-series	Y	Y	NF	NF	Y
Machine Learning							
Linear Regression	linreg	Linear Regression between columns rand1 and rand2 in data1	Y	Y	Y	Y	Y
Clustering	kmeans	K-means Clustering with two clusters between columns rand1 and rand2	Y	Y	Y	Y	Y
Classification	naivebayes	Naive Bayes (rand1 as the target and rand2 and uni in data1 as features)	Y	Y	Y	Y	Y

statistics. But we put them under the task suite distribution and inferential statistics, to have a sizable number of tasks in that suite. Ideally, these tasks should be based on built-in functions or should have simple implementation in each of the systems. We implemented the tasks using built in functionality when available. However, in some cases this was not possible. The tasks included in the micro benchmark are summarized in Table 4.2. The table also indicates the level of support for these tasks in each data system.

4.3.1.1 Micro Benchmark Task Suites

- **Basic File I/O**: An important goal of the micro benchmark is to demonstrate weaknesses and strengths on a specific task for each of the systems. File I/O operations are the basic tasks that are necessary to do any data analysis. In our benchmark reading and writing operations are performed with CSV files. This file format was chosen for consistency, because each data system has the ability to read and write from CSV.

- **Data Wrangling**: The data wrangling tasks are: sorting, filtering, merging, groupby and removing duplicates. All the five data systems provide built-in functionality to implement all of these tasks.

- **Descriptive Statistics**: The descriptive statistical analysis tasks are: finding central tendency, finding measures of dispersion, ranking, eliminating outliers and plotting a scatter plot. However, the outcome of breaking ties in sorting is different depending on the default data sys-

tem. Dask does not have the functionality to rank a column based on value. For plotting tasks, Dask, Anaconda (Matplotlib) and R have the built-in support for plotting.

- **Distribution and Inferential Statistics**: All the systems are able to calculate correlation. However, when it comes to estimating the PDF (probability density function), hypothesis testing and skewness some of the systems do not have the functionality. The PDF is calculated using the kernel density estimator. The PostgreSQL (MADlib) does not have any available functionality for PDF. Dask has a rolling skew function, but the max window size is the size of a partition, which is smaller than one million. In our experiments, PySpark did not support efficient hypothesis testing for datasets with over one million rows.

- **Time Series**: Time series is a sequence of data points indexed by time order. The two tasks in this suite are autocorrelation and EWMA (exponential weighted moving average). Anaconda, Dask and R have built-in functionality to calculate autocorrelation. PostgreSQL or MADlib does not have either feature built-in and we have not found an efficient user defined function (UDF). Dask has built-in functionality for autocorrelation, but not for EWMA. PySpark does not have built-in functionality for either of the time series tasks.

- **Machine Learning**: The machine learning tasks are: linear regression, K-means Clustering and Naive Bayes Classification. All the the

systems or libraries used by the systems have built-in functionality to complete these tasks. Dask and Anaconda use the same library scikit-learn, PostgreSQL use MADlib, whereas, R and PySpark have built-in functionality in the library.

4.3.1.2 Data Model for the Micro Benchmark

We created a synthetic dataset generator that can generate data tables (as text files) with different scale factors. As shown in Table 4.3, each dataset consists two data tables: data1 and data2. For a given scale factor, both the data tables contain the same number of rows. For instance, with scale factor (SF) 1, each data table has 1 million rows and with SF 100, each has 100 million rows. Three different scale factors were used for the scalability experiments: SF 1, 10, and 100.

The schema and the number of columns of data1 and data2 differ. Table data1 contains one time series, two string, two integer and three float columns. The time series column is sequential and starts at the year 1970 and the time interval between each row shrinks as the scale factor grows. One string column values are chosen uniformly from a list of 1,000 elements and the other string column values are chosen from a Zipf distribution of a list of 100,000 elements. One integer column values are randomly chosen between 0 and 1,000,000 and the other between 0 and 2^{31}. The values of the three float columns are selected from a normal distribution, an exponential distribution and a uniform distribution respectively.

Table 4.3: Micro benchmark dataset: cardinality for various scaling factors (Mil.=Million)

Data table	**Scale factor**					
	1		**10**		**100**	
	Rows (Mil.)	Size (MB)	Rows (Mil.)	Size (MB)	Rows (Mil.)	Size (GB)
data1	1	92.6	10	926.3	100	9.3
data2	1	43.9	10	438.9	100	4.4

Table 4.4: Macro benchmark dataset (K=thousand)

Data table	**Rows (K)**	**Size (MB)**	**Benchmark application**
weather	35,064	0.863	Smart Grid Analytics - Demand Forecast
consumption	35,064	0.875	Smart Grid Analytics - Demand Forecast
hockey_stats	888	1.2	Sports Analytics - Corsi in Ice Hockey

Table data2 contains one string, one integer and two float columns. The string column values are chosen from a uniform distribution from a list of 1000 strings, the integer column values are randomly distributed between 0 to 1,000,000 and the float columns values are selected from a normal distribution and a uniform distribution between 0 and 2^{31}.

4.3.2 Macro Benchmark

With the rapidly rising volume of data, the importance of data science is growing in many application domains. The goal of our macro benchmark is to model some of these applications and assess the performance of the evaluated data systems with each. In our macro benchmark we have included two real-

world applications. Each application consists of a series of data science tasks that are executed in sequence. Table 4.5 summarizes these applications and which data systems have support for which tasks from these application. In the next section, we briefly describe each application.

4.3.2.1 Macro Benchmark Applications

- **Smart Grid Analytics - Demand Prediction**: The demand prediction application focuses on predicting power consumption over a year for a particular substation based on the HDD (heating degree day) and CDD (cooling degree day) curves. The tasks for the demand prediction are: reading data from multiple CSV files, create 5 minute time series intervals from 2013 to 2016, merge all the data, add two separate columns, filter the data based on two values, predict the consumption for the years and correlate the real results with the predicted results.

- **Sport Analytics - Corsi in Ice Hockey**: The hockey analytics application focuses on calculating well-known and used statistics called Corsi and Fenwick. These are used to calculate the player effectiveness while on the ice. The tasks of finding Corsi and Fenwick are reading the data from a CSV, calculate Corsi, as well as Fenwick, based on several statistics, split the players based on position and group the players by team's so one could evaluate each team puck possession skills.

4.3.2.2 Data Model for Macro Benchmark

The details of the datasets in the macro benchmark are shown in Table 4.4. The data used for the smart grid demand prediction came from two sources: historical weather data for New Brunswick from Environment Canada [30], and historical New Brunswick power demand from NB Power [59]. The data used for the sports analytics came from the hockey analytic database [39].

Table 4.5: Macro benchmark tasks and feature matrix

Operation	**TaskId**	**Description**	**Support in the evaluated systems**				
			Anaconda	**Dask**	**Postgre-SQL**	**PySpark**	**R**
Smart Grid Analytics - Demand Prediction							
Reading table data	reading	Read data from CSV files	Y	Y	Y	Y	Y
Create data	create	Create five minute time series data	Y	Y	Y	Y	Y
Join tables	merge	Join consumption and weather tables on time columns	Y	Y	Y	Y	Y
Added HDD Heating Degree Day	addcol	Calculated heating degree day from outside temperature	Y	Y	Y	Y	Y
Filter	filter	filter on columns HDD and on the hour of the day	Y	Y	Y	Y	Y
Find curves	2linereg	Use linear regression to find heating and cooling curve from HDD and consumption	Y	Y	Y	Y	Y
Predict consumption	predict	Predict power consumption for 3 years from the curves and outside temperature	Y	Y	Y	Y	Y
Correlation	corr	Use correlation to see if curves predicted realistic results	Y	Y	Y	Y	Y
Sport Analytics - Corsi and Fenwick in Ice Hockey							
Reading data	reading	read the data from a CSV	Y	Y	Y	Y	Y
Corsi statistics	corsi	Add multiple columns for each player (row) for Corsi	Y	Y	Y	Y	Y
Fenwick statistics	fenwick	Add multiple columns for each player (row) for Fenwick	Y	Y	Y	Y	Y
Split players on position	split	Filter the players based on forwards and defense	Y	Y	Y	Y	Y
Group players	groupby	Group payers based on their team	Y	Y	Y	Y	Y

4.4 Implementation

In this section we discuss about the benchmark implementation and the challenges we encountered along the way.

4.4.1 Implementation Overview

The details of the benchmark tasks can be found in the Description columns of Table 4.2 and 4.5. The source code of Sanzu can also be inspected to provide additional details. While implementing a task, we tried to use as much built-in functionality as possible. If the built-in functionality was absent, no effort was made to implement it in a given system. However, if open source libraries were available with that functionality, it was implemented. An issue with using the built-in software was the different built-in defaults. For example, while sorting data if there was a tie, each system would arrange the tied values based on a different criteria.

4.4.2 Challenges

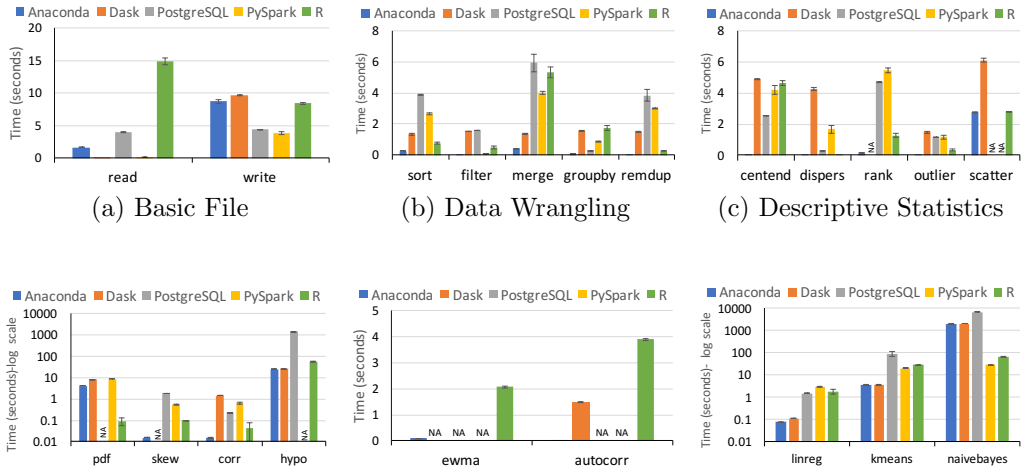
Anaconda and R were similar in terms of implementation effort, as they both have sufficient documentation Dask extends libraries used in Anaconda, but only partially. As a result, implementation in Dask was straightforward until a certain functionality was missing. This missing functionality resulted in an incomplete task. PySpark and PostgreSQL required more development effort than the other systems. Sometimes, otherwise simple operations needed

significant coding effort. For example, in PySpark there are only two ways to get rows by index while using a dataframe. The first is to add a new column for the index manually and filter based on this. The second is to convert to another data-type or structure, perform slicing and indexing operations there, and then convert back to a dataframe. This issue presented itself when developing the hypothesis testing, as there was no efficient way to shuffle (randomize) data and then split it based on the index, into a PySpark dataframe. Furthermore, when working with the macro benchmark (i.e. real-world messy data), cleaning the data up is much more difficult while using PostgreSQL or PySpark, as opposed to Anaconda or R.

While running tasks involving certain aggregate functions (e.g. finding the median and mode), PostgreSQL 9.3 took too long even with dataset scale factor 1. After upgrading PostgreSQL to the later version (9.6), the issues were resolved.

4.5 Experimental Setup

During the experiments, for each task and for a specific data system we loaded the data first. Then each task was run four times. We exclude the first run from the results as this can be affected by the loading of the data. We report execution times of each task with error bars. In-house data science tasks are usually run on a single machine so we limited all systems to one machine even though Dask and PySpark could scale out to a cluster. Each



(a) Basic File (b) Data Wrangling (c) Descriptive Statistics

(d) Distribution and Inferential Statistics (e) Time Series (f) Machine Learning

Figure 4.1: Execution times of the tasks in the Micro Benchmark suites (scale factor 1: 1 million)

machine has 8 GB memory, 1 socket with 4 cores (per socket) and each 4 Intel i5-2400 CPUs running at 3.10 GHz. The machines run Ubuntu 14.04 OS.

4.6 Benchmark Results and Discussion

In this section we present our evaluation of the systems with the benchmark. We introduce the micro benchmark results first, and then the macro benchmark results.

4.6.1 Micro Benchmark Results

We conducted two different series of experiments to evaluate the data systems with the micro benchmark task suites. The first series of experiments (Section 4.6.1.1) were run with dataset scale factor (SF) 1 i.e. 1 million rows per data table and the goal was to determine relative performance and feature support. The aim of the second series of experiments (Section 4.6.1.2) was to evaluate scalability of the systems with increased data size. So for these we used all three datasets with SF 1, 10 and 100.

4.6.1.1 Performance and Functionality

As can be seen in Figure 4.1, there is no system that is the best across-the-board in all tasks. On the basic file I/O, R is the slowest at reading, however it is competitive in writing, while PySpark is the fastest at both. In the data wrangling suite, Anaconda is consistently the fastest and PostgreSQL is the slowest in all but the groupby, and the others vary depending on the task. In descriptive statistics suite, Anaconda is consistently the fastest, R is in second in all but one tasks. PostgreSQL, Dask and PySpark lack some functionalities (shown as 'NA'), and PostgreSQL outperforms Dask and PySpark in every task it has functionality. In the distribution and inferential statistics suite Anaconda and R are the fastest in all cases; Dask, PySpark and PostgreSQL are slower and lack different functionalities for certain tasks. In the time series suite, Anaconda is the fastest. R also has functionality for both time series tasks, but it is significantly slower. In contrast, Dask only supports

autocorrelation, and PySpark and PostgreSQL do not have functionality for either task. As for the machine learning task suite, Anaconda is the fastest in two out of the three tasks, while PostgreSQL is the slowest in two out of the three. These results suggest that at dataset scale factor 1, Anaconda's performance is the most consistent and it is the best choice for most tasks. R also supports all the features, but it is slower than Anaconda in all but two tasks. As for Dask, PySpark and PostgreSQL, their performance and functionality depend on specific tasks. Thus at SF 1, whenever the data fits in memory, Anaconda is the best choice for the combination of functionality and performance.

4.6.1.2 Scalability

Table 4.6 summarizes information about scalability of the data systems for each task of all the micro benchmark suites. In Figure 4.2 the scalability results are presented for some selected tasks.

In almost all cases all the systems scale up to SF 10 (ten million rows per data table). The only exceptions are the merge operation, which joins both data tables. As the memory of the systems start to run out, R and PostgreSQL throw memory errors, whereas Anaconda is significantly slower than Dask and PySpark. Naive Bayes Classification did not scale up on any of the systems except for R, while hypothesis testing scaled for all except for PostgreSQL.

For the SF 100 (100 million rows per data table), the R data loading was

taking over two hours to read, PostgreSQL ran out of memory that caused the machine to restart and there was a memory error thrown during the read for Anaconda. As a result, the rest of the tasks for these systems were not evaluated. It is important to note that if the task failed or took too long on a lower row count, it was not evaluated again on a higher SF. For the systems that we were able to run at SF 100, their results mostly stayed consistent with the results at a lower SF. Some tasks, however, did not scale up from SF 10 to 100. For Dask, it was only with K-means Clustering that did not scale. Whereas for PySpark, central tendency, measure of dispersion and K-means Clustering were all unable to scale. These results suggest that in terms of scalability up to SF 100, Dask scales better than others with regards to functionality.

4.6.2 Macro Benchmark Results

In Figure 4.3 the macro benchmark results are presented. As can be seen, in the Demand Prediction application, there is no single system that is the best across-the-board. Anaconda is consistently competitive and is the quickest in the task reading, linear regression, and prediction. The others vary in competitiveness and performance depending on the task.

In the sport analytics application, Anaconda has the best performance in all of the tasks and R consistently has the second best performance. PostgreSQL is also very competitive with R but is much slower in the split task. PySpark and Dask are consistently slower in overall performance than the others.

4.6.3 Ranking the Systems

Historically, benchmark results were summarized and presented in several different ways. The authors in [24] discuss how to correctly summarize benchmark results. In [75] geometric mean, normalized to a reference system, was used for ranking. We considered using this approach, but it proved to be challenging. In our benchmark a number of tasks were not supported by some of the systems either due to lack of functionality or because of failure to complete at higher scale factors. Therefore, we decided not to rank the systems.

4.7 Summary

In this chapter, we presented Sanzu, a benchmark for data science. It includes a micro benchmark to test individual operations and a macro benchmark to represent real-world use cases. We have presented a performance evaluation of five popular data science frameworks and systems: R, Anaconda Python, Dask, PostgreSQL (MADlib) and PySpark. Our evaluation suggests that some of these systems lack support for specific data science tasks. Furthermore, we have found that several systems do not scale with larger data sets (scale factor 10 or 100). We hope that our findings may lead to improved support for functionality and scalability for the data science tasks within some of the systems.

Table 4.6: Scalability of the Micro benchmark tasks with dataset scale factors: 1, 10 and 100 [NA=Not Applicable (the dataset could not be loaded), NF = No built in functionality available, F=failed (Out of memory error or crashed the machine), L=took too long]

Operation	**Support in the evaluated systems**														
	Anaconda			**Dask**			**PostgreSQL**			**PySpark**			**R**		
	scale factor			scale factor			scale factor			scale factor			scale factor		
	1	**10**	**100**	**1**	**10**	**100**	**1**	**10**	**100**	**1**	**10**	**100**	**1**	**10**	**100**
Basic File I/O															
Read	Y	Y	F	Y	Y	Y	Y	Y	F	Y	Y	Y	Y	Y	F
Write	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Data Wrangling															
Sort	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Filtering	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Merging	Y	Y	NA	Y	Y	Y	Y	F	NA	Y	Y	Y	Y	F	NA
Grouping	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Removing Duplicates	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Descriptive Statistics															
Central Tendency	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	F	Y	Y	NA
Measure of dispersion	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	F	Y	Y	NA
Rank	Y	Y	NA	NF			Y	Y	NA	Y	Y	Y	Y	Y	NA
Outliers	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Scatter Plot	Y	Y	NA	Y	Y	Y	NF			NF			Y	Y	NA
Distribution and Inferential Statistics															
PDF	Y	Y	NA	Y	Y	Y	NF			Y	F	NA	Y	Y	NA
Skewness	Y	Y	NA	NF			Y	Y	NA	Y	Y	Y	Y	Y	NA
Correlation	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Hypothesis Testing	Y	Y	NA	Y	Y	Y	Y	L	NA	NF			Y	Y	NA
Time Series															
EWMA	Y	Y	NA	NF			L	NA	NA	NF			Y	Y	NA
Autocorrelation	Y	Y	NA	Y	Y	Y	NF			NF			Y	Y	NA
Machine Learning															
Linear Regression	Y	Y	NA	Y	Y	Y	Y	Y	NA	Y	Y	Y	Y	Y	NA
Clustering	Y	Y	NA	Y	Y	F	Y	Y	NA	Y	Y	F	Y	Y	NA
Classification	Y	L	NA	Y	L	NA	Y	F	NA	Y	F	NA	Y	Y	NA

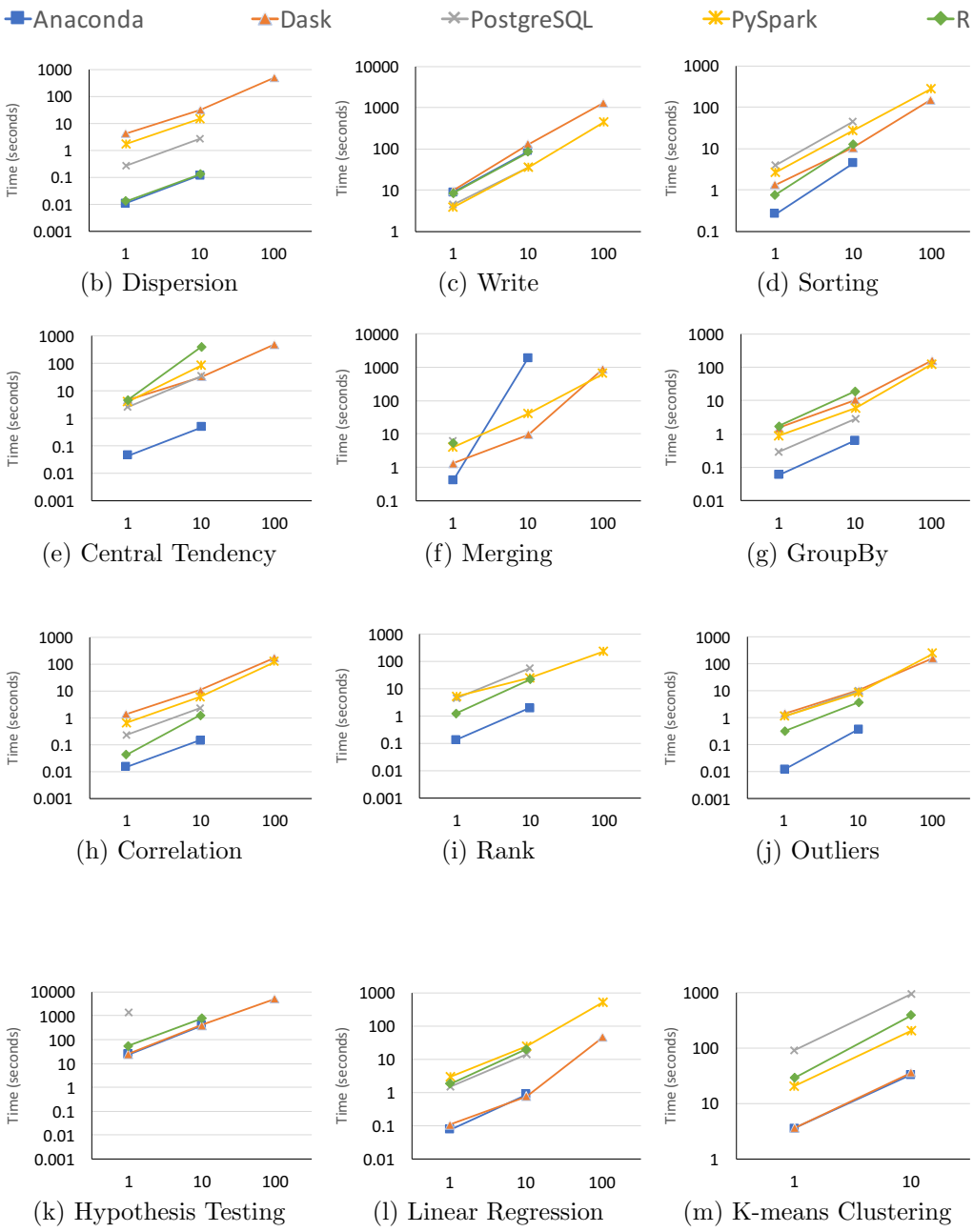
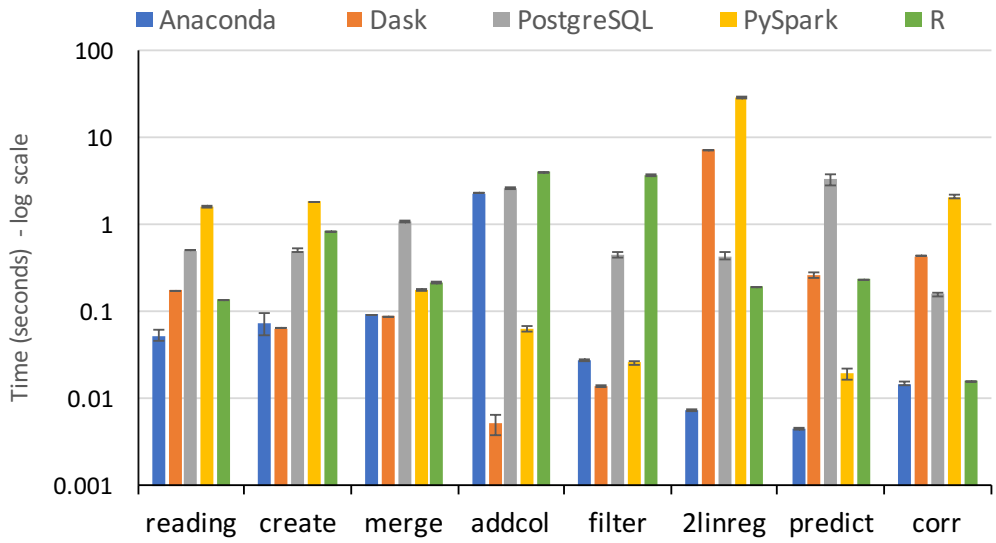
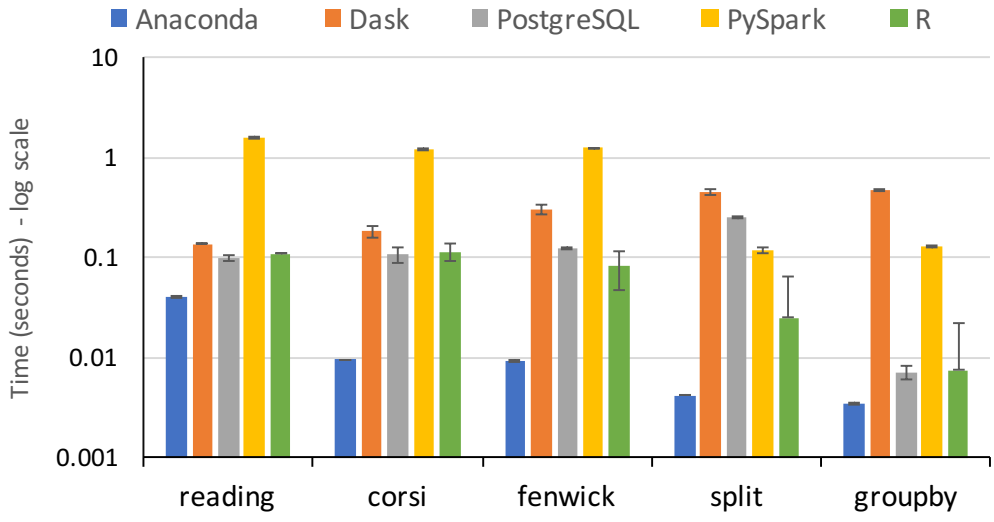


Figure 4.2: Scalability of **selected** Micro benchmark tasks with dataset scale factors: 1, 10 and 100 (All y-axis are in log scale)



(a) Smart Grid Analytics - Demand Prediction



(b) Sport Analytics - Corsi and Fenwick in Ice Hockey

Figure 4.3: Execution times of the tasks in the Macro Benchmark applications

Chapter 5

DaskDB: Unifying Data Analytics and SQL querying

5.1 Introduction

In this chapter, we introduce and describe the unified big data system, DaskDB, which we have developed for supporting SQL queries and data analytics. We discuss why we believe that there is an opportunity to develop this type of system and why current state-of-the-art systems that perform analytics fall short. Next, we present the system architecture of DaskDB, and discuss the system design principles, including why we chose each aspect and how we implement different components of DaskDB. Finally, we evaluate and compare our system against a state-of-the-art big data system, Spark and discuss the results.

5.2 Methodology

As the volume of data continuously grows, the data science community are constantly searching for simpler, quicker and more efficient ways to analyze and process this data. Consider a scenario, in which an analyst wants to perform some analytics on a 100 GB (Gigabytes) dataset. However, the data is stored as a raw text file. This raw text file may be formatted in different ways, such as a .txt (text), CSV (comma-separated values) or JavaScript Object Notation (JSON). The three popular options to perform this type of analysis include a dedicated data analytic system comprised of language frameworks, a DBMS, or a big data framework. In the next subsections, we will explore the strengths and weaknesses of each of these options.

5.2.1 Scalability of Dedicated Data Analytic Systems

The first option to process and analyze data is using a dedicated data analytic system. They are typically based on language frameworks, such as R, MATLAB and Python. The upside to this option is that these systems are usually simple to install and require limited configuration. Furthermore, they have large ecosystem of data analytic libraries supporting a diverse set of tasks, such as machine learning, data visualization, scientific computing, graph libraries, multidimensional data structures, file formats, and database connectors.

The Anaconda distribution of Python and R ecosystem are the most popular

dedicated analytic systems currently being used. Performing different types of analysis is possible with these systems, due to the availability of large body of libraries mentioned in Section 2.2. However, since the data is relatively large (100 GB - in the scenario mentioned above), it would not fit within the main memory (2 to 32 GB) of a typical personal computer. This issue has been mentioned in Section 4.6.1.2, as the Anaconda and R systems cannot load the 9.3 GB dataset into a machine's main memory (8GB RAM). Until recently, the way all dedicated data analytic systems worked was that all data was needed to be loaded into the main memory before performing an analysis task. So, a supercomputer or a workstation with main memory larger than 100GB would be needed to perform an analysis task. In recent times, several dedicated analytic systems have been extended with parallel computing libraries. They automatically keep the data in the disk and load parts of the data into memory as needed for processing. An example of one of these libraries is Dask, as mentioned in Section 4.6.1.2. Dask is the only framework, supporting Python data science library ecosystem, which we found that has the ability to scale out to a cluster of machines. However, Dask does not have SQL querying capabilities.

5.2.2 The Challenge of ETL with DBMS

The second option is to use a DBMS and in particular utilize the in-database analytic functionality to perform the analysis. Most modern DBMS have data analytic capabilities and functionalities such as mentioned in Section

3.3.3. Traditionally, database application developers need to create user-defined functions (UDF) for analytics, but they treat UDFs as black boxes with little or no optimizations. Depending on the DBMS, it is possible to execute UDFs written in R, Python, C, C++ or T-SQL languages. However, the only way to interact with UDFs are through their input parameters and final output results. Furthermore, they are often constrained in terms of the number and data types of its input and output. This results in them being difficult to debug and to incrementally develop [21]. Further, the other major issue is the bottleneck involved with extracting, transforming and loading (ETL) the data into the DBMS. ETL is a time-consuming process and is not viable or necessary in many big data workflows. For example, only a subset of the data might be actually valuable or useful, so loading it into the database is a waste of time and resources. Also, with the data volume growing exponentially, a greater portion of the data for analysis is not being stored in databases.

5.2.3 Usability in Big Data System Frameworks

The last option is to perform the analysis using a big data system framework. These frameworks can scale-up processing to multiple machines (if machines are available and configured), partition the data into chunks and keep data in disk, until chunks of the data are brought into main memory to process. Furthermore, they enable automatically programming entire clusters with data parallelism and built-in fault tolerance. Some popular frameworks include,

Hadoop [81], Spark [94], Storm [1] and Hive [89], and often a combination of these frameworks are used in real-world applications. Their programming interfaces include Java, Python, Scala, SQL (or a SQL-like language) and R, and their functionalities include SQL querying capabilities, machine learning, streaming and graph libraries. The downside of these types of frameworks is that they often entail more complicated APIs and a steeper learning curve than the other two options. In general, big data frameworks are more complex, take larger development effort and have comparatively lower usability. Furthermore, many programmers often have knowledge of high-level programming languages, but they usually lack domain knowledge to use these frameworks effectively. For example, even though these frameworks have popular data science language interfaces like Python and R, they require an analyst to use completely new APIs native to the big data framework. This results in an increase in time for non-experts to become proficient at using such a system. We ran into this issue specifically while performing experimental evaluation with Spark. It took multiple iterations to find an optimal configuration by reading the documentation.

5.2.4 Our Approach: DaskDB

To address the above-mentioned issues, we introduce DaskDB. Before DaskDB, if a user (analyst or DBA) wanted to perform SQL queries with analysis on a large dataset, they only had three practical options.

We summarize these options (as discussed in Sections 5.2.1, 5.2.2 and 5.2.3)

next. First, the user could choose a subset of the data and pick whatever system they preferred to perform analytics. However, this data sub-setting increase algorithm complexity and decrease accuracy compared to the full dataset [19]. This is a bad choice as it will produce non-realistic results. Second, the user could choose an in-database analytic system, however, these systems have high ETL cost. Additionally, as data continues to grow, the ETL time of data will only get larger. If a user wants to perform any additional analysis not supported by SQL, she either has to use a UDF or load the data out of the DBMS into her analytic system. This loading process constrains the user to option three: use a big data system framework like Spark or Hive. However, these frameworks have complicated APIs, steeper learning curves and it takes a considerable amount of time and effort to become proficient in them. This leaves the average user with a decision: either allocate a significant amount of time to properly learn the framework, or pay a trained user to do it or try to convert the SQL code using the APIs supported by a big data analytics system like Dask. These considerations provided the inspiration for DaskDB, which in addition to supporting all Dask features, also enables SQL querying on raw data in a data science friendly environment. We believe that DaskDB or a system like it gives the ability for more potential users to face the challenges of big data analytics.

5.3 Our System

In this section, we explain the approach we took to implement our system of DaskDB. Our goal is to create a unified big data system that supports SQL query, and big data analytics with competitive performance and scalability, high usability and built-in EDA capabilities. Fortunately, due to the selection of our implementation environment of Anaconda distribution, a significant part of our goal is already met due to the supported API ecosystem, as we discuss in Section 5.3.1 below. Next, we explain the components of DaskDB and the design decisions we made in Section 5.3.2.

5.3.1 Why Python?

One of the main reasons we chose to work with Python is the growing popularity of it. From the StackOverflow.com question views and trends in Figure 5.1[1], Python has grown steadily to become the most popular viewed question tag over the past 6 years. A major reason for this growth is the large number of data science libraries currently available. The Anaconda distribution is the main data science platform for Python. Furthermore, Dask is part of this distribution and is configured out of the box with various data science packages. We have mentioned about a few of them in Section 2.2. Some of the other frameworks include Datashader, a graphics pipeline system that visualizes large datasets and machine learning libraries scikit-learn and

[1]https://stackoverflow.blog/2017/09/06/incredible-growth-python/

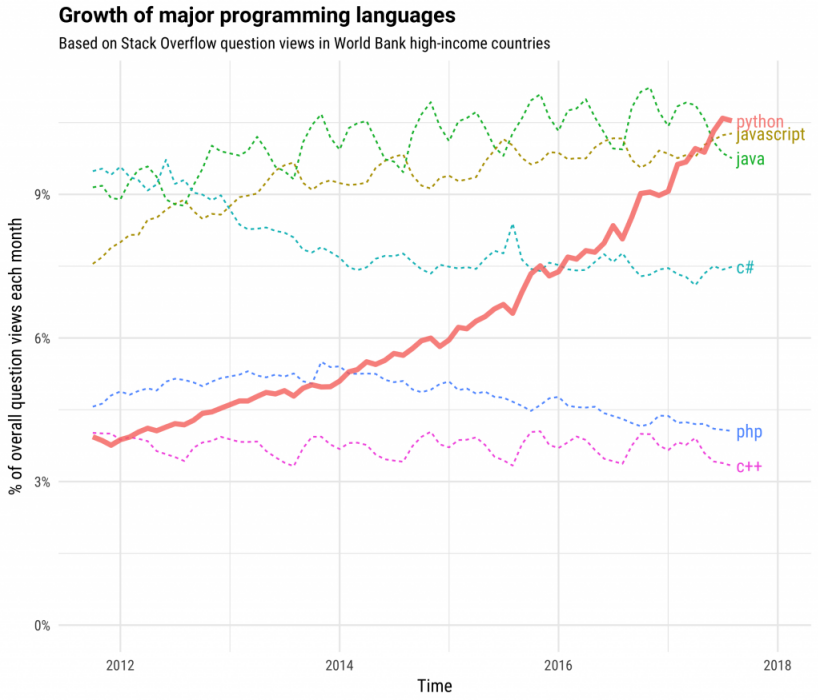


Figure 5.1: StackOverflow Question Tags Views

DaskML [2]. Lastly, the number of available data science packages in Anaconda distribution is growing.

5.3.2 System Architecture

The system architecture of DaskDB incorporates five main components: the SQLParser, QueryPlanner, DaskPlanner, DaskDB Execution Engine, and HDFS. They are shown in Figure 5.2. First, the SQLParser gathers metadata information pertaining to the SQL query, such as the names of the tables, columns and functions. This information is then passed along to the

[2]http://ml.dask.org/

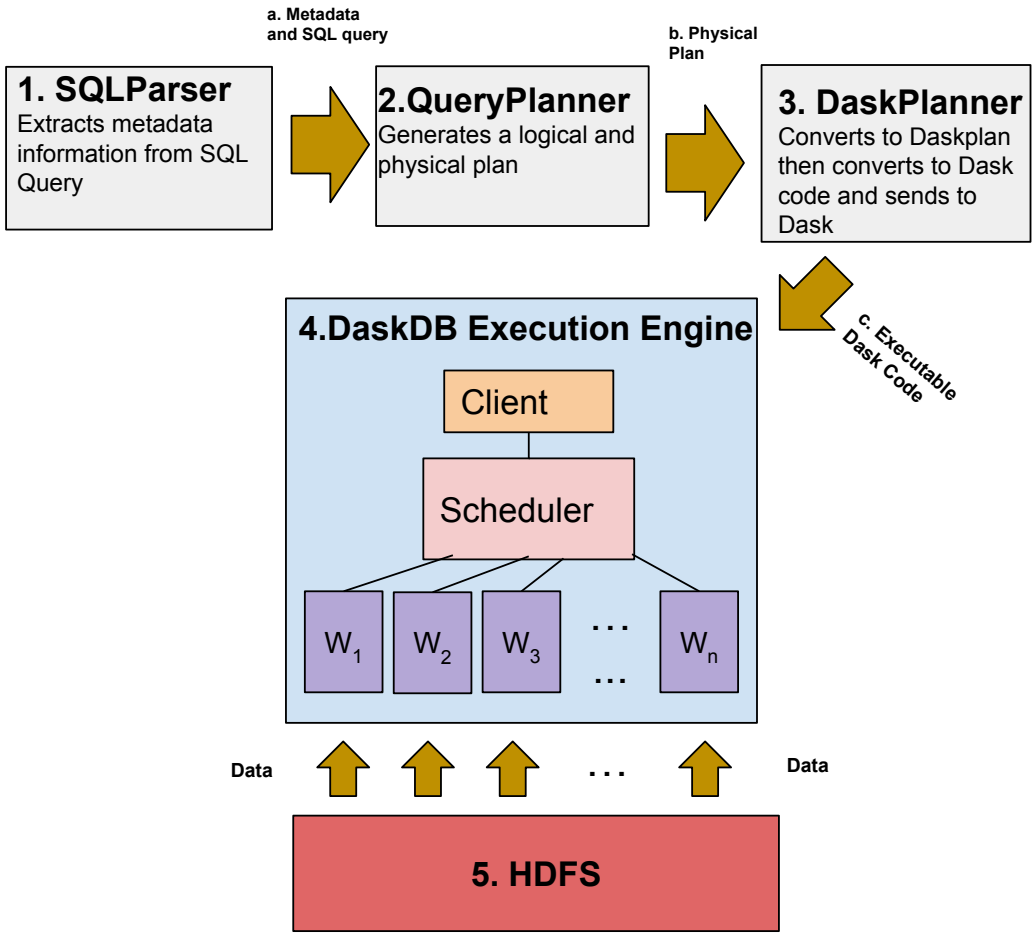


Figure 5.2: DaskDB System Architecture

QueryPlanner. Next, in the QueryPlanner component, physical plan is generated from the information sent by SQLParser about the SQL query. The physical plan is an ordered set of steps that specify a particular execution plan for a query and how data would be accessed. The QueryPlanner then sends the physical plan to the DaskPlanner. In the DaskPlanner, a plan is generated, which closely resembles the Dask API, called the Daskplan. The

Daskplan is produced from the physical plan, and it is then converted into Python code and sent to DaskDB Execution Engine. Note that when we refer to Dask, we refer to both Dask and Dask.distributed packages. DaskDB Execution Engine then executes the code and gathers the data from the HDFS then computes the SQL query. Further details are provided in the sections below.

5.3.2.1 SQLParser

The SQLParser is the first component of DaskDB that is involved in query processing. The input for the SQLParser is the original SQL query. It first checks for syntactical errors and creates a parse tree with all the metadata information about the query. We then process the parse tree to gather the metadata information needed by the QueryPlanner. This metadata information includes table names, column names and UDFs. We then check if the table(s) exist (e.g. if there is a .txt file with that name) in the default directory. If the table exists, we collect the relevant table name, column names and data types, and dynamically generate a schema. The schema contains information about all the tables and column names and data types used in the SQL query. The schema, UDFs (if any) and the original SQL query are then passed to the QueryPlanner.

Note that our system can treat any file (with a supported file format) as a data table, whereas a traditional DBMS requires an ETL process to load data into its native storage. Thus, DaskDB can support in-situ heterogeneous

data source querying [11].

5.3.2.2 QueryPlanner

The QueryPlanner, shown in Figure 5.2, is the second component of DaskDB designed to support query execution. The QueryPlanner creates logical and preliminary physical plans. The schema and UDFs produced by SQLParser, along with the SQL query, are passed into the QueryPlanner. The QueryPlanner uses these to first create a logical plan and then a optimized preliminary physical plan. This plan is then sent to the DaskPlanner.

5.3.2.3 DaskPlanner

The DaskPlanner is used to transform the preliminary physical query plan from the QueryPlanner into Python code that is ready for execution. The first step in this process is for the DaskPlanner to go through the physical plan obtained from QueryPlanner and convert it into what we call the 'Daskplan'. This maps the operators from the physical plan into operators that more closely resemble the Dask API. This Daskplan also associates relevant information with each operator from the physical plan. This information includes columns and tables involved and specific metadata information for a particular operator. We also keep track of each operator's data dependency. This is needed to pass intermediate results from one operation to the next. The pseudo code of how DaskDB converts the physical plan into the Daskplan is shown in Algorithm 1.

Input: A physical plan (P) is given as input. P contains ordered groups of dependent operators (G). Each G consists of an ordered list of tuples (k, o, d), where k contains the unique key of a operation, o is the operation type and d contains the operation metadata information.

Output: The final result is a Daskplan DP, which consists of an ordered list of operators.

```
DP ← list();
for sorted(G ∈ P) do
    for sorted(k, o, d ∈ G) do
        dp ← dict() //create Daskplan operator;
        dp[o] ← convertToDaskPlanOperator(o) ;
        dp[d] ← getMetadataInfo(d) ;
        dp[key] ← k //adds key (used to get data dependencies) ;
        DP.add(dp) //adds dp to Daskplan (order is important) ;
    end
end
//Each operation (aside from the table scans) needs intermediate data
 (tables) results from the previous operations. Note that the join
 operation has two children (c), so we need to check if a particular
 operation has multiple children.
for sorted(dp ∈ DP) do
    while dp has children (c) do
        dp[t_i] ← get table information from c_i;
    end
end
return DP;
```

Algorithm 1: DaskPlanner: Conversion of Physical Plan to Daskplan

In the next step, the DaskPlanner converts the Daskplan into the Python code, which utilizes the Dask API. The Algorithm 2 shows pseudo code of how DaskDB performs this. All of the detail about each table and their particular column names and indexes are tracked and maintained in a dictionary throughout the execution of a query, this is because multiple tables and columns may be created, removed or manipulated in any given query. For example, columns often become unnecessary after a particular filter or join operation is executed. These columns are dropped for optimization to avoid unneeded data movement. Columns may also be added due to aggregation or groupby operators. For these reasons, the names of the tables, columns, and indexes are dynamically maintained while transforming the Daskplan into Python code to execute it. The resultant code can be executed by Dask over a cluster of machines.

5.3.2.4 DaskDB Execution Engine

As shown in Figure 5.2, there are three main components of the DaskDB execution engine: the client, scheduler and workers. The client transforms the Dask Python code into a set of tasks. The scheduler creates a DAG (directed acyclic graph) from the set of tasks, automatically partitions the data into chunks, while taking into account data dependencies. The scheduler sends a task at a time to each of the workers according to several scheduling policies. The scheduling policies[3] for task and workers depend on various

[3]http://distributed.dask.org/en/latest/scheduling-policies.html

Input: A Daskplan (DP) is given as input. DP contains an ordered list of tuples (o, d, t), where o is the operation type, d contains the operation metadata information and t contains the table dependency from previous operation(s).
Output: The final result *exec* is Python executable code.
exec ← list();
cols ← dict() // maintains the columns indexes for all tables ;
for *sorted(*$o, d, t \in DP$*)* **do**
    O ← get(o) // get operation in Dask API
    T ← get(t) // get table(s) from previous operation(s)
    D ← set(O,d) // set the operation specific Dask parameters
    E ← convertToDaskCode(O,T,D)
    columns ← update(T,cols) // update tables and columns indexes
    exec.add(E) // add to final executable Python code.
end
return **exec**;

Algorithm 2: DaskPlanner: Conversion of DaskPlan to Executable Code

factors including data locality. A worker computes and stores a data chunk until it is not needed anymore and is instructed by the scheduler to release it.

The DaskDB Execution Engine extends the libraries, Dask and Dask.distributed. Both these libraries were introduced and briefly explained in Section 2.2. We refer to both Dask and Dask.distributed as Dask. Enhancements to the Dask library were necessary for a few SQL features to work properly within the execution engine, as the original library does not support sorting with a limit by multiple columns in different directions (one column ascending, one column descending). Adjustments were made to the Pandas and Dask APIs for 'nLargest' and 'nSmallest' methods. Furthermore, a bug needed to be fixed for Dask to properly interact with HDFS.

5.3.2.5 HDFS

The Hadoop Distributed File System (HDFS) [81] is the primary data storage system used by Hadoop applications. It is one of the more popular open source distributed file systems and many big data frameworks including Spark, Hive and Storm utilize it. Its basic architecture include the components: 'NameNodes' and 'DataNodes'. HDFS provides high-performance and access to data across highly scalable Hadoop clusters.

5.3.2.6 Implementation

DaskDB was implemented in Python. A few tools and frameworks were utilized to develop DaskDB. The SQLParser utilizes a tool called 'sqlparser'[4]. This tool was chosen for its simplicity and very little overhead. This tool checks for syntactical errors and returns a parse tree from a SQL query as input. The QueryPlanner extends a tool called Raco [92], which is used in the MyriaL data language as a relational algebra compiler. We decided to use Raco because it was natively developed in Python, and hence there is very little integration overhead. Other options such as Q*cert [8] and Calcite [9] were considered, but the difficulty in integrating separate runtime environments rendered them impractical. We also modified Raco to accept a few more keywords such as 'interval' and 'date', as well as the binary operators, so that we could support date strings in a SQL query. We made these modifications so that it is compatible with the TPC-H benchmark

[4]https://github.com/TwoLaid/python-sqlparser

queries.

5.3.3 Support for SQL Query with UDFs in DaskDB

DaskDB supports UDFs in SQL querying. A UDF enables a user to create a function using Python code and embed it into the SQL query. Since DaskDB converts the SQL query, including the UDF back into Python code, the UDFs can reference and utilize features from any of the data science packages available in Anaconda. This is another advantage that DaskDB has over Spark. Spark can only execute UDFs that can be applied to one data tuple at a time. In contrast, DaskDB has the ability to pass in UDFs with code involving machine learning, data visualization, numerous other functionalities.

5.4 Illustration: DaskDB query execution

In this Section, we show an example to illustrate how DaskDB executes an SQL query. The Listing 5.1 is the example query used in our illustration. This query contains two tables: *lineitem* and *orders*. The operations in this query include filter, join, aggregation and orderby. This query is a simplified version of the type of queries in the TPC-H benchmark. We visualize the query as it gets transformed from the logical to physical plan and then to the Daskplan shown in Figure 5.3. The logical and physical plan use standard relation algebra describe in [74] to represent the operations of a SQL query.

We then show the Python code the DaskPlanner creates for the Daskplan in Figure 5.4. This code is then sent to the DaskDB Execution Engine. The resulting execution is illustrated in Figure 5.5a in a per worker per core fashion. We also include part of the corresponding progress bar that shows how the colors relate to specific tasks in the execution in Figure 5.5b.

Listing 5.1: SQL Query associated with Figure 5.3

```
SELECT
        l_orderkey,
        sum(l_extendedprice * (1-l_discount)) as revenue
FROM
        orders, lineitem
WHERE
        l_orderkey = o_orderkey
        and o_orderdate >= '1995-01-01
GROUP BY l_orderkey
ORDER BY revenue
LIMIT 5
```

5.4.1 SQL Query Plan Transformation

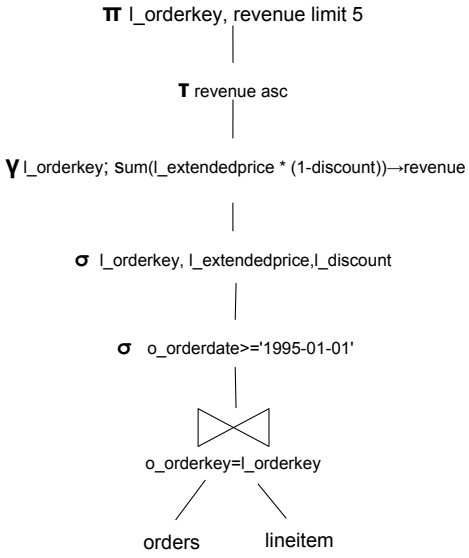
In Figure 5.3, three plans are shown which are generated from the SQL query in Listing 5.1. However, this SQL query must be scanned, parsed and validated by the SQLParser before it can be interpreted as query execution

plan. The first plan generated is the logical plan, as shown in Figure 5.3a. The logical plan is an unoptimized internal representation of the query within the QueryParser. The logical plan is the initial plan and is presented in the form of a relation algebra tree. The QueryParser then performs query optimization to try to find efficient query execution strategies. This results in a second plan generated by the QueryParser called the physical plan, shown in Figure 5.3b. The physical plan shows a specific join operator with the 'hash join'. Also, the filter condition is now pushed down below the join operation. This is done to minimize the data transfer done during the execution of the join and reduce the cost of the join.

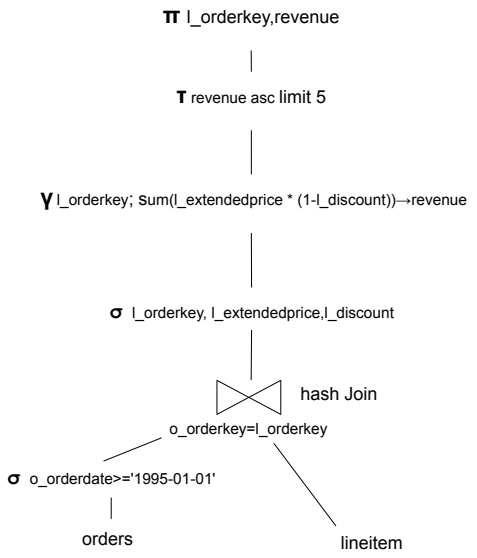
The Daskplan is the third plan generated show in Figure 5.3c. The Daskplan is generated from the physical plan in the DaskPlanner component. The Algorithm detailing how this is done is shown in Algorithm 1. The Daskplan does not use relational algebra, instead, the operators are converted to tasks that more closely resemble the Dask API. For example, these include the 'read_csv' and 'filters' methods shown on the tree. This Daskplan is then converted into executable Python code shown in Figure 5.4.

5.4.2 Executable Python Code

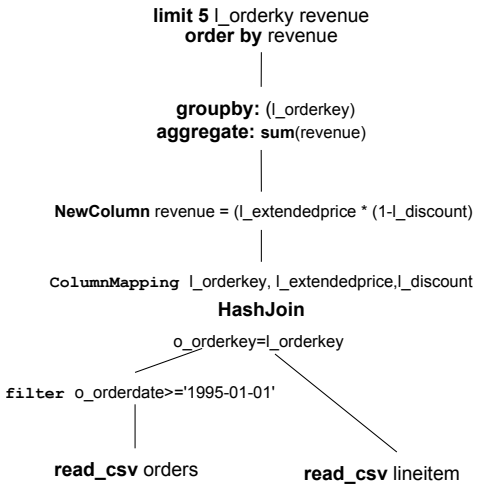
The Python code that is generated from the DaskPlanner can now be sent to the DaskDB Execution Engine. DaskDB keeps track of the column indexes for each table and the table indexes. The method 'add_columns_index' appears when we manipulate a table, particularly when there is a need to



(a) Logical Plan Relational Algebra



(b) Physical Plan Generated by Query-Parser



(c) Generated Daskplan

Figure 5.3: Plan Transformations

check for changes on the given table columns. For example, when tables are joined or filtered, often times columns are not needed anymore and can be

dropped, which minimizes data movement. The generated executable code is shown in Figure 5.4. The code is then sent to the execution engine. For this example, we executed it on a cluster of 4 machines each having 4 cores. Thus, there are 16 processes where a task may be executed. This is shown in Figure 5.5, as there are 16 processes shown on the 'y' axis. The 'x' axis is the time elapsed. This illustration shows a typical workflow within DaskDB. Each task is represented by a bar graph, and the colour of the task corresponds to a task type shown in Figure 5.5b. For example, yellow represents 'merge' and so is 'getitem'. Note that 'getitem' and 'merge' are related and hence they are coloured the same. The only type of task not shown in the legend are the data transferring tasks in red. This happens when data needs to be transferred from one machine to another. To summarize, we use this visualization to demonstrate the progress of the different task steps involved in an SQL query execution on our distributed data analytics system.

5.4.3 DaskDB with UDFs

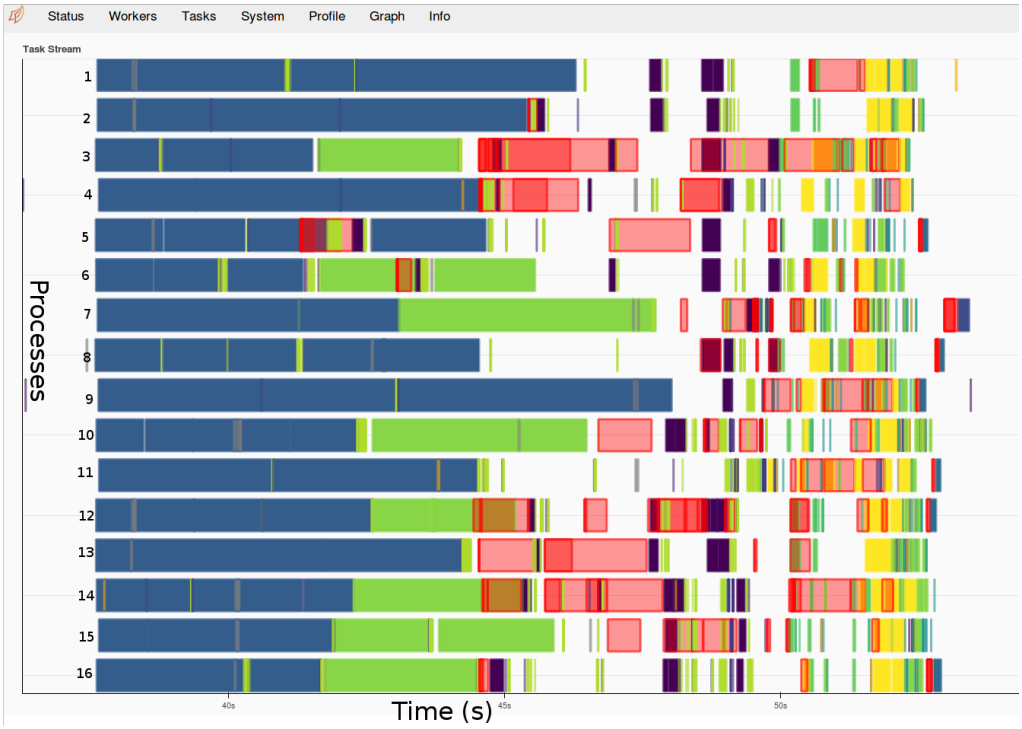
In this section, we show two examples of DaskDB using UDFs in SQL queries. The first example is of a UDF that removes duplicates. As can be seen in the code in Figure 5.6, the UDF, 'remove_dup', is referenced in the same way as a typical SQL based application. The SQL query combines two tables *customer* and *orders*, filters the data with dates prior to "1995-03-21" and then removes all duplicates. The query returns a Dask dataframe structure containing three columns. The 'query' method sends the SQL statement to

```
add_columns_index(orders,'orders')
orders=orders[orders[col_map['orders'][1]]>=pd.to_datetime('1995-01-01')]
orders=orders.rename(columns={col_map['orders'][0]:'o_orderkey'})
add_columns_index(orders,'orders')
orders=orders.loc[:,['o_orderkey']]
add_columns_index(orders,'orders')
divisions = get_table_division('lineitem',col_map['lineitem'][0])
lineitem=lineitem.set_index(col_map['lineitem'][0], divisions=divisions)
set_table_division('lineitem',col_map['lineitem'][0],lineitem.divisions)
indexed_col2=col_map['lineitem'][0]
merge_table=orders.merge(lineitem,left_on=col_map['orders'][0],right_index=True)
len_col1 =len(col_map['orders'])
add_columns_index(merge_table,'merge_table')
merge_table=merge_table.loc[:,[col_map['merge_table'][
1+len_col1-1],col_map['merge_table'][2+len_col1-1]]]
merge_table=merge_table.reset_index()
merge_table=merge_table.rename(columns={'index':indexed_col2})
add_columns_index(merge_table,'merge_table')
merge_table=merge_table.loc[:,[col_map['merge_table'][0],
col_map['merge_table'][1],col_map['merge_table'][2],]]
add_columns_index(merge_table,'merge_table')
merge_table=merge_table.rename(columns={col_map
['merge_table'][0]:'l_orderkey'})
add_columns_index(merge_table,'merge_table')
merge_table['_COLUMN1_']=merge_table[col_map['merge_table'][1]] \
*(1-merge_table[col_map['merge_table'][2]])
merge_table=merge_table.loc[:,['l_orderkey','_COLUMN1_']]
add_columns_index(merge_table,'merge_table')
gb=merge_table.groupby([col_map['merge_table'][0]])
merge_table=gb.agg({col_map['merge_table'][1]:'sum'})
merge_table=merge_table.reset_index()
add_columns_index(merge_table,'merge_table')
merge_table=merge_table.rename(columns={col_map['merge_table'][0]:'l_orderkey',
                                        col_map['merge_table'][1]:'revenue'})
add_columns_index(merge_table,'merge_table')
merge_table=merge_table.loc[:,['l_orderkey','revenue']]
add_columns_index(merge_table,'merge_table')
merge_table=merge_table.nsmallest(5,columns=['revenue']).compute()
```

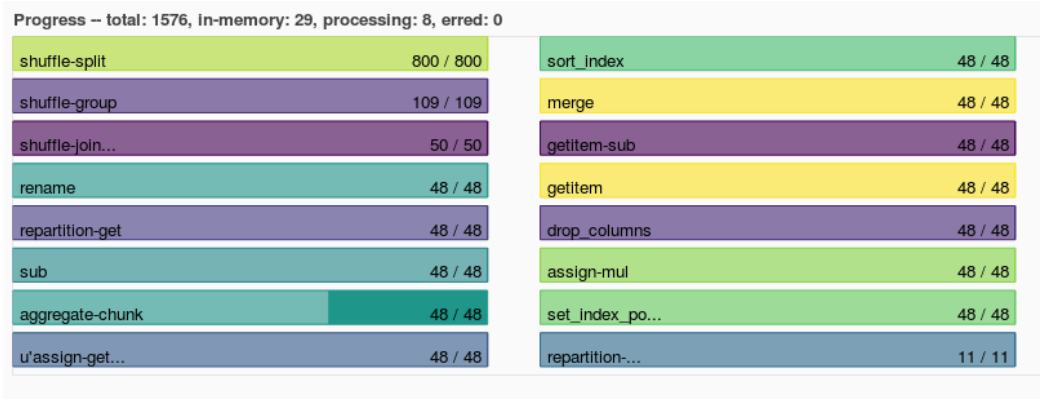
Figure 5.4: Executable Python Code

DaskDB to be executed. The resulting DAG is shown in Figure 5.7.

Next we present an illustration of a linear regression example using a Dask machine learning package (dask_ml). As shown in Figure 5.8, the UDF is again called in the same way as it would be in a typical DBMS system. In the SQL query, we invoke the 'lin_regress_fit' method to create and fit the



(a) Query Execution on the DaskDB Engine



(b) Progress Bar for Cooresponding Task execution

Figure 5.5: Query Execution on the DaskDB Engine

```
In [5]: sql_rem = '''SELECT
                        remove_dup(c_custkey),
                        o_custkey,
                        o_orderdate
                    FROM
                        customer,orders
                    WHERE
                        c_custkey = o_custkey
                        and o_orderdate < date "1995-03-21";
                    '''
        rem_dup = query(sql_rem)
        rem_dup.visualize()
```

Figure 5.6: Removing Duplicate Code

linear regression. The query result can be further manipulated using native Python code to create and plot a prediction for the test data.

5.5 Evaluation

In this section, we discuss our experimental evaluation. All systems evaluated are publicly available open-source software. The software version, programming language and key libraries of these systems are shown in Table 4.1.

5.5.1 Experimental Setup

During the experiments, for each task, and for a specific data system we loaded the data first. Then each task was run four times. We exclude the first run from the results as this can be affected by the loading of the data. We report execution times of each task with error bars. We ran experiments

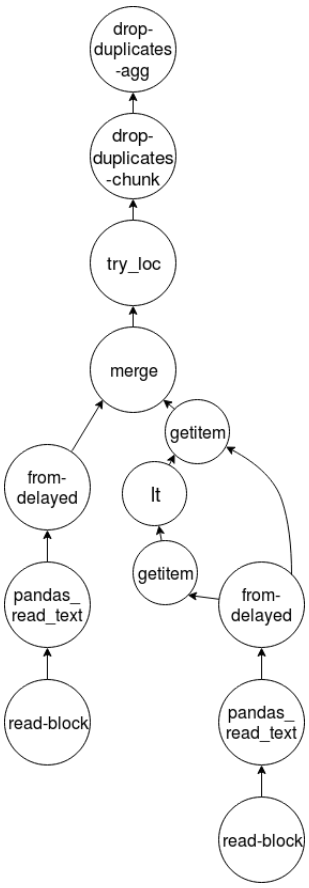


Figure 5.7: Removing Duplicate DAG Plan

on a cluster of 4 machines each having 8 GB memory and 4 Intel i5-2400 CPUs running at 3.10 GHz and each machine ran Ubuntu 14.04 OS. HDFS was used to store the datasets for each system on all the tasks.

We chose to compare DaskDB against Spark because it has similar data science functionalities as DaskDB. Furthermore, Spark is considered the state-of-the-art and is one of the most popular big data frameworks. Spark was used in other benchmarks mentioned in Sections 3.2.2 and 3.2.3. Spark,

```
In [13]: auto = fetch_data('auto')
         auto, auto_test = train_test_split(auto)
         test_length = auto_test.loc[:,['length']].values
         test_target = auto_test.loc[:,'target'].values

         sql_lin_reg = """SELECT  lin_regress_fit(length,target)
                             FROM auto
                           WHERE
                             length > 0;
                       """
         md = query(sql_lin_reg)

         target_pred= md.predict(test_length)
         plt.scatter(test_length,test_target , color='black')
         plt.plot(test_length, target_pred, color='blue', linewidth=3)

         public:adhoc:auto

Out[13]: [<matplotlib.lines.Line2D at 0x7f76d01efc50>]
```

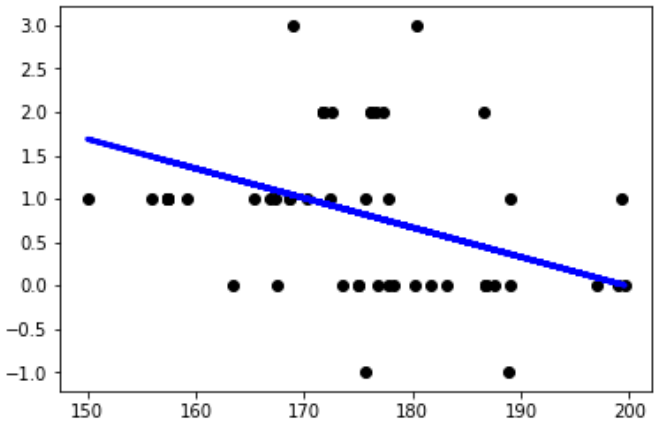


Figure 5.8: Resulting Linear Regression Graph

SparkSQL or PySpark have appeared in many other benchmarking studies involving the TPC-H such as [87, 71, 5, 7].

Table 5.1: Software configuration

Framework	Version	Language	Key Package(s)
DaskDB	1.0.0	Python 2.7.3	Dask, dask_ml dask.distributed
Spark	2.4.0	Python 2.7.3	PySpark, SparkSQL
Hadoop	2.9.1	Java 1.8.0	HDFS, Yarn

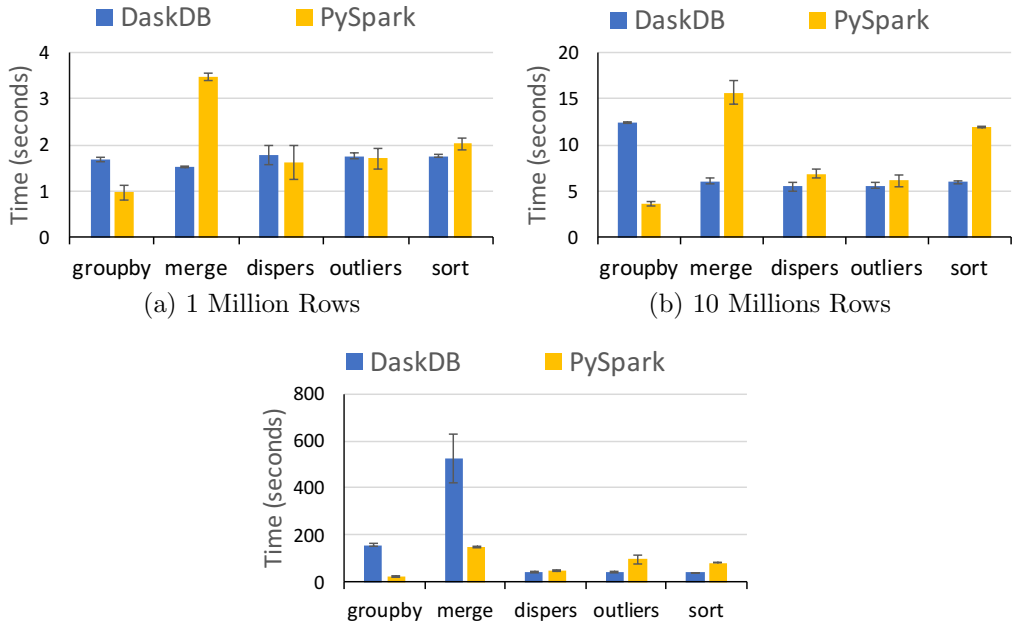
5.5.2 Benchmarks

5.5.2.1 Extension of Analytic Benchmark

For this evaluation, we ran a subset of the micro benchmark tasks used in the Sanzu benchmark describe is Chapter 4. A few changes were made to both the Spark and Dask code to extend task execution to a cluster of machines, but the code for each task remained the same. The tasks we extend are groupby, join (merge), dispersion, outliers, and sorting. These tasks show the strength and weaknesses of each of the systems as well as, highlight some of the results in the TPC-H benchmark.

5.5.2.2 TPC-H Benchmark

The TPC-H benchmark is a suite of ad hoc analytical (OLAP) SQL queries [2]. It consists of 8 datasets (tables) and 22 queries typically evaluated with DBMS systems that have SQL or SQL-like query execution capabilities. The queries and the database tables have industry-wide relevance. This benchmark illustrates a decision support system that examines and processes large volumes of data. It executes queries with a high degree of complexity. The



(a) 1 Million Rows

(b) 10 Millions Rows

(c) 100 Millions Rows

Figure 5.9: Execution times for the Analytic Tasks

data tables are generated as text files using dbgen [5], and they are loaded into HDFS. We used 3 scale factors (SF): 1, 4, and 10, where SF 1 indicates that the dataset size is 1GB.

5.5.3 Results

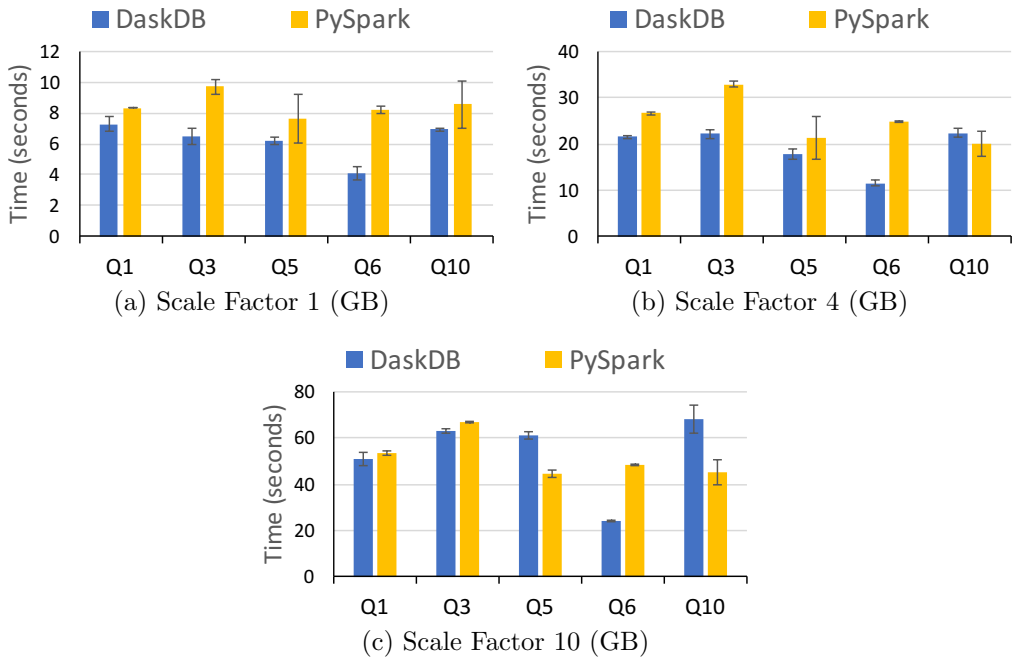
In this section we present our results of the evaluation of the systems. We present the Sanzu (analytic) results first, and then the TPC-H benchmark results.

[5]dbgen: https://github.com/electrum/tpch-dbgen

5.5.4 Analytic Benchmark Results and Discussion

In Figure 5.9, we see that neither DaskDB nor Spark is better across all tasks or scales factors. Looking at Figure 5.9a, we see that DaskDB shows better performance on the merge task, and Spark has better performance on the groupby, while in three other tasks the performance of the two systems is relatively close. As we scale up to 10 million rows, in Figure 5.9b, we see that the groupby task has faster completion time on Spark than DaskDB, while the other four tasks are all faster on DaskDB than on Spark. Scaling up to 100 million rows in Figure 5.9c, we see that four out of the five tasks have the same trend as 10 Million, while the performance of merge has completely changed. We see Spark has better performance in the groupby and merge, while DaskDB has better performance on sort, measures of dispersion (dispers) and removing outliers.

Looking back to Figure 4.2, we observe that the results from the Sanzu benchmark remain mostly consistent on all of the tasks, except for two, as we scaled out to a cluster of machines. The main differences are with the merge and groupby tasks, as DaskDB becomes comparatively worse than Spark. DaskDB does not scale out as well as Spark on groupbys and merge, although they were fairly close in terms of performance on a single machine. DaskDB was able to the scale out to a cluster effectively on the other three analytical tasks.



(a) Scale Factor 1 (GB)

(b) Scale Factor 4 (GB)

(c) Scale Factor 10 (GB)

Figure 5.10: Execution times of the query's in the TPC-H

5.5.5 TPC-H Benchmark Results and Discussion

The benchmark consists of SQL query tasks. As can be seen in Figure 5.10, neither DaskDB nor Spark is better across the board at scale factor 1. DaskDB has faster completion times on Q1, Q3, Q6, and Q10, while Spark is only faster on Q5, as shown in Figure 5.10a. At scale factor 4, shown in Figure 5.10b, we see that the trends remain the same for Q1, Q3, Q6, and Q5, however, Spark is much faster on Q10. In Figure 5.10c, at scale factor 10, we see that with Q5 and Q10, Spark further improve its completion time over DaskDB, while on Q1 and Q6 still has a consistently faster completion time on DaskDB. However, on Q3 Spark seems to have closed the gap in

completion time slightly.

Looking at the queries makeup specifically, queries 5 and 10 have more table joins and groupbys than the other queries. This is where Spark held the advantage over DaskDB. Q6 has no groupby or joins, while Q1 has a groupby but no joins and Q3 has both joins and groupbys. These results suggest that in terms of scalability up to SF 10, DaskDB scales better on more analytical queries, but Spark holds advantages over the DaskDB on queries with more groupby and joins.

5.6 Challenges and Limitation of DaskDB

The results of the evaluation indicate that DaskDB is not as optimized as Spark on joins and groupbys. One issue is that by default Dask combines all groupby aggregate chunks into a single Pandas dataframe. It is up to the user to manually make sure that the output does not get too large. DaskDB also automatically combines the resulting aggregation chunks (every 8 chunks by default). This value is fine for most cases, but if the aggregation chunks are large, it could take a substantial amount of time to combine them, especially when transferring chunk from worker to worker. Another reason may be that DaskDB cannot parallelize within individual tasks nor does it automatically re-partition data. For example, if a task gets too large like on a join, it may overwhelm any particular worker, and this could force a restart, slowing down the overall completion time. Furthermore, Dask also

assigns tasks to workers heuristically, and this type of task scheduling usually makes the right decision. However, non-optimal situations do occur, and they can lead to unneeded data transfers between workers. A limitation of our implementation of DaskDB is that it only has limited subquery (query placed within another query) execution. Lastly, the machine learning packages for Dask are still in their infancy. We believe that there are opportunities for significant improvements in many different aspects of DaskDB.

5.7 Summary

DaskDB is a novel approach to addressing the question of how to unify query processing and data analysis capabilities in a single system. It is an integrated big data system that combines the functionalities of a data analytics system with the SQL querying features of a DBMS. It brings SQL querying to a data science platform in a way no other existing system, which we are aware of, can match in terms of usability, performance, scalability and built-in capabilities. We have introduced and described each of the components of DaskDB and how they seamlessly work together to convert a SQL query into executable Python code using the Dask APIs. Furthermore, DaskDB also has the ability to incorporate any UDF into the input SQL query, where the UDF could invoke any Python library call. The other systems, (DBMS and big data frameworks) with UDF capabilities, that we looked into, have some sort of restriction with the UDF. These include only allowing scalar

UDFs (restricting output results) and restricting initial input parameters. Lastly, we evaluated our solution DaskDB against Spark using the TPC-H benchmark and several data science tasks from our Sanzu benchmark and demonstrated its competitive performance.

Chapter 6

Conclusions, Challenges and Future Work

6.1 Summary of Contributions

With the rapid growth in data that is being generated from diverse sources, the need to efficiently analyze data has become paramount. As a result, data science is rising in importance. Data science provides a systematic approach for processing and analyzing data. Although a number of frameworks and data systems have emerged to support the data science work-flow, there is no standard benchmark to evaluate them. We have presented Sanzu, a benchmark for data science. It includes a micro benchmark to evaluate individual operations and a macro benchmark to represent real-world use cases. We conducted a systematic performance evaluation of five popular data science

frameworks and systems: R, Anaconda Python, Dask, PostgreSQL (MADlib) and PySpark. Our evaluation suggests that some of these systems lack support for specific data science tasks. Furthermore, we have found that several systems do not scale with larger datasets (10 or 100 million rows). We hope that our findings may lead to improved support for functionality and scalability for the data science tasks within some of the systems.

While evaluating data science frameworks with the Sanzu benchmark, we found that many popular systems lack support for specific data science tasks and they do not scale with larger data sets. We then discussed how there is room for improvements in most state-of-the-art systems that perform data analytics and SQL querying as part of the same workflow. We addressed why current big data frameworks and DBMS fall short in terms of scalability, performance, usability, and built-in functionalities when performing these workflows. Thus, we developed DaskDB, a unified big data system that combines data analytics with SQL querying capabilities. We described the design principles and system architecture of DaskDB and discuss the benefits it has over current state-of-the-art systems. Finally, we evaluated our solution DaskDB against Spark using TPC-H, and an extended version of the Sanzu benchmark.

6.2 Future Work

For our future work on Sanzu, we plan to test the ability of each system to process other data models, by incorporating semi-structured and unstructured datasets. We would like to include additional macro benchmark applications. We would also like to evaluate other data science systems, and further scale-out Sanzu to evaluate data science systems that run on a cluster of machines.

With DaskDB, we think there are a plenty of potential opportunities for future work and at the top of the list are the limitations mentioned in Section 5.6. We would like to optimize joins and groupbys within Dask, look into particularly at adaptive indexing, dynamic partitioning resizing, and different task scheduling strategies and policies. We would also like to incorporate the ability to process subqueries inside DaskDB. Lastly, we also like to develop and evaluate and test the benefit of a distributed cache within DaskDB framework.

References

[1] *Apache storm*, http://storm.apache.org/.

[2] *Tpc-h*, http://www.tpc.org/tpch/.

[3] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré, *Levelheaded: A unified engine for business intelligence and linear algebra querying*, 2018 IEEE 34th International Conference on Data Engineering (ICDE) (2018), 449–460.

[4] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi, *Rheem: Enabling cross-platform data processing: May the big data be with you!*, Proc. VLDB Endow. **11** (2018), no. 11, 1414–1427.

[5] Victor Aluko and Sherif Sakr, *Big sql systems: an experimental evaluation*, Cluster Computing (2019).

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia, *Spark sql: Relational data processing in spark*, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '15, ACM, 2015, pp. 1383–1394.

[7] Jason Arnold, Boris Glavic, and Ioan Raicu, *A High-Performance Distributed Relational Database System for Scalable OLAP Processing*, Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium, 2019.

[8] Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon, *Q*cert: A platform for implementing and verifying query compilers*, SIGMOD Conference, 2017.

[9] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire, *Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources*, Proceedings of the 2018 International Conference on Management of Data (New York, NY, USA), SIGMOD '18, ACM, 2018, pp. 221–230.

[10] Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J. Haas, and Christopher Jermaine, *Simulation of database-valued markov chains using simsql*, Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '13, ACM, 2013, pp. 637–648.

[11] Javad Chamanara, Birgitta König-Ries, and H. V. Jagadish, *Quis: In-situ heterogeneous data source querying*, Proc. VLDB Endow. **10** (2017), no. 12, 1877–1880.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears, *Benchmarking Cloud Serving Systems with YCSB*, SoCC, 2010, pp. 143–154.

[13] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur etintemel, and Stanley B. Zdonik, *Tupleware: "big" data, big analytics, small clusters*, CIDR, 2015.

[14] *Dask*, https://dask.pydata.org/en/latest/.

[15] Dask Development Team, *Dask.distributed*, 2016.

[16] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan, *Orange: Data mining toolbox in python*, Journal of Machine Learning Research **14** (2013), 2349–2353.

[17] Amol Deshpande and Samuel Madden, *Mauvedb: Supporting model-based user views in database systems*, Proceedings of the 2006 ACM

SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '06, ACM, 2006, pp. 73–84.

[18] Vasant Dhar, *Data science and prediction*, Commun. ACM **56** (2013), no. 12, 64–73.

[19] Pedro Domingos, *A few useful things to know about machine learning*, Commun. ACM **55** (2012), no. 10, 78–87.

[20] Bonnie J. Dorr, Peter C. Fontana, Craig S. Greenberg, Marion Le Bras, and Mark A. Przybocki, *Evaluation-driven research in data science: Leveraging cross-field methodologies*, IEEE Big Data (2016), 2853–2862.

[21] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme, *Aida: Abstraction for advanced in-database analytics*, Proc. VLDB Endow. **11** (2018), no. 11, 1400–1413.

[22] Franz Färber, Norman May, Wolfgang Lehner, Philipp Grosse, Ingo Müller, Hannes Rauhe, and Jonathan Dees, *The sap hana database – an architecture overview*, IEEE Data Eng. Bull. **35** (2012), 28–33.

[23] Xixuan Feng, Arun Kumar, Ben Recht, and Christopher R, *Towards a unified architecture for in-rdbms analytics*, Proceedings of the ACM SIGMOD International Conference on Management of Data (2012).

[24] Philip J. Fleming and John J. Wallace, *How not to lie with statistics: The correct way to summarize benchmark results*, Commun. ACM (1986).

[25] Edouard Fouché, Alexander Eckert, and Klemens Böhm, *In-database analytics with ibmdbpy*, SSDBM, 2018.

[26] Franz Frber, Norman May, Wolfgang Lehner, Philipp Grobe, Ingo Mller, Hannes Rauhe, and Jonathan Dees, *The sap hana database - an architecture overview*, IEEE Data Eng. Bull. **35** (2012), 28–33.

[27] Ahmad Ghazal, Todor Ivanov, Pekka Kostamaa, Alain Crolotte, Ryan Voong, Mohammed Al-Kateb, Waleed Ghazal, and Roberto V. Zicari, *BigBench V2: The New and Improved BigBench*, ICDE, 2017, pp. 1225–1236.

[28] 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*, SIGMOD, 2013, pp. 1197–1208.

[29] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand, *Musketeer: all for one, one for all in data processing systems*, EuroSys, 2015.

[30] *Historical Climate Data*, http://climate.weather.gc.ca.

[31] *Graysort benchmark*, http://sortbenchmark.org.

[32] *Greenplum*, https://greenplum.org/.

[33] Joel Grus, *Data science from scratch*, O'Reilly Media, Inc., 2015.

[34] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan, *Amazon redshift and the case for simpler data warehouses*, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '15, ACM, 2015, pp. 1917–1923.

[35] *Apache Hadoop*, http://hadoop.apache.org/.

[36] *Apache HBase*, https://hbase.apache.org/.

[37] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar, *The MADlib Analytics Library: Or MAD Skills, the SQL*, Proc. VLDB Endow. **5** (2012), no. 12, 1700–1711.

[38] Tony Hey, Stewart Tansley, and Kristin Tolle, *The fourth paradigm: Data-intensive scientific discovery*, October 2009.

[39] *Hockey stats*, http://www.hockeyabstract.com.

[40] Markus Hofmann and Ralf Klinkenberg, *Rapidminer: Data mining use cases and business analytics applications*, Chapman & Hall/CRC, 2013.

[41] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi, *Big data and its technical challenges*, Commun. ACM **57** (2014), no. 7, 86–94.

[42] Nick R. Katsipoulakis, Yuanyuan Tian, Fatma Özcan, Hamid Pirahesh, and Berthold Reinwald, *A generic solution to integrate sql and analytics for big data*, EDBT, 2015.

[43] *Python eats away at R: Top Software for Analytics, Data Science, Machine Learning in 2018: Trends and Analysis*, https://www.kdnuggets.com/2018/05/poll-tools-analytics-data-science-machine-learning-results.html.

[44] Jonathan Lajus and Hannes Mühleisen, *Efficient data management and statistics with zero-copy integration*, Proceedings of the 26th International Conference on Scientific and Statistical Database Management (New York, NY, USA), SSDBM '14, ACM, 2014, pp. 12:1–12:10.

[45] Mark Levene and George Loizou, *Why is the snowflake schema a good data warehouse design?*, Information Systems **28** (2003), no. 3, 225 – 240.

[46] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang, *Mlog: Towards declarative in-database machine learning*, Proc. VLDB Endow. **10** (2017), no. 12, 1933–1936.

[47] Sam Lightstone, Russ Ohanian, Michael Haide, James Cho, Michael Springgay, and Thomas. Steinbach, *Making big data simple with dashdb local*, 2017 IEEE 33rd International Conference on Data Engineering (ICDE), April 2017, pp. 1195–1205.

[48] Xiufeng Liu, Lukasz Golab, Wojciech Golab, Ihab F. Ilyas, and Shichao Jin, *Smart meter data analytics: Systems, algorithms, and benchmarking*, ACM Trans. Database Syst. **42** (2016), no. 1, 2:1–2:39.

[49] Antonio Maccioni and Riccardo Torlone, *Augmented access for querying and exploring a polystore*, 2018 IEEE 34th International Conference on Data Engineering (ICDE) (2018), 77–88.

[50] Dennis Marten and Andreas Heuer, *A framework for self-managing database support and parallel computing for assistive systems*, Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments (New York, NY, USA), PETRA '15, ACM, 2015, pp. 25:1–25:4.

[51] *MATLAB*, https://www.mathworks.com/products/matlab.html.

[52] *Matplotlib*, https://matplotlib.org/.

[53] Q McCallum, Ethan, *Bad data handbook*, O'Reilly Media, Inc., 2012.

[54] Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, and Yusra AlSayyad, *Comparative evaluation of big-data systems on scientific image analytics workloads*, Proc. VLDB Endow. (2017), 1226–1237.

[55] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar, *Mllib: Machine learning in apache spark*, Journal of Machine Learning Research **17** (2016), no. 34, 1–7.

[56] *Microsoft SQL Server*, https://www.microsoft.com/en-ca/sql-server/sql-server-2017.

[57] *MySQL*, http://www.mysql.com.

[58] Fabian Nagel, Gavin Bierman, and Stratis D. Viglas, *Code generation for efficient query processing in managed runtimes*, Proc. VLDB Endow. **7** (2014), no. 12, 1095–1106.

[59] *NB Power*, https://tso.nbpower.com/Public/en/system_information_archive.aspx.

[60] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper, *Fast serializable multi-version concurrency control for main-memory database systems*, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '15, ACM, 2015, pp. 677–689.

[61] *Numpy, scientific computing with Python*, http://www.numpy.org/.

[62] *NYTimes article*, https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html.

[63] Kyle O'Brien, Vijay Gadepally, Jennie Duggan, Adam Dziedzic, Aaron J. Elmore, Jeremy Kepner, Samuel Madden, Tim Mattson, Zuohao She, and Michael Stonebraker, *Bigdawg polystore release and demonstration*, vol. abs/1701.05799, 2017.

[64] *Oracle*, https://www.oracle.com/ca-en/database/.

[65] Shoumik Palkar, J. Joshua Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab, *Weld : A common runtime for high performance data analytics*, 2016.

[66] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia, *Evaluating end-to-end optimization for data analytics applications in weld*, PVLDB **11** (2018), 1002–1015.

[67] *Pandas Python Data Analysis Library*, http://pandas.pydata.org/index.html.

[68] Linnea Passing, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann, *Sql- and operator-centric data analytics in relational main-memory databases*, EDBT, 2017.

[69] *PostgreSQL*, http://www.postgresql.org/.

[70] *Anaconda*, https://www.continuum.io/.

[71] X. Qin, Y. Chen, J. Chen, S. Li, J. Liu, and H. Zhang, *The performance of sql-on-hadoop systems - an experimental study*, 2017 IEEE International Congress on Big Data (BigData Congress), June 2017, pp. 464–471.

[72] *The R Project for Statistical Computing*, https://www.r-project.org/.

[73] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham, *Froid: Optimization of imperative programs in a relational database*, Proc. VLDB Endow. **11** (2017), no. 4, 432–444.

[74] Raghu Ramakrishnan and Johannes Gehrke, *Database management systems*, 3 ed., McGraw-Hill, Inc., New York, NY, USA, 2003.

[75] Suprio Ray, Bogdan Simion, and Angela Demke Brown, *Jackpine: A benchmark to evaluate spatial database performance*, ICDE, 2011, pp. 1139–1150.

[76] Erich Schubert, Alexander Koos, Tobias Emrich, Andreas Züfle, Klaus Arthur Schmid, and Arthur Zimek, *A framework for clustering uncertain data*, PVLDB **8** (2015), no. 12, 1976–1979.

[77] Rachel Schutt and Cathy O'Neil, *Doing data science: Straight talk from the frontline*, O'Reilly Media, Inc., 2013.

[78] *scikit-learn Machine Learning in Python*, http://scikit-learn.org/stable/.

[79] *SciPy*, https://www.scipy.org/.

[80] Amir Shaikhha, Yannis Klonatos, and Christoph Koch, *Building efficient query engines in a high-level language*, CoRR **abs/1612.05566** (2016).

[81] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, *The hadoop distributed file system*, Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (Washington, DC, USA), MSST '10, IEEE Computer Society, 2010, pp. 1–10.

[82] *Sisense*, https://www.sisense.com.

[83] *Apache Spark*, https://spark.apache.org/.

[84] I.O.F. Standardization, *Iso 9241-11: Ergonomic requirements for office work with visual display terminals (vdts): Part 11: Guidance on usability*, 1998.

[85] *Tableau*, https://www.tableau.com/.

[86] Rebecca Taft, Manasi Vartak, Nadathur Rajagopalan Satish, Narayanan Sundaram, Samuel Madden, and Michael Stonebraker, *Genbase: A complex analytics genomics benchmark*, SIGMOD, 2014, pp. 177–188.

[87] Ashish Tapdiya and Daniel Fabbri, *A comparative analysis of state-of-the-art sql-on-hadoop systems for interactive analytics*, 2017 IEEE International Conference on Big Data (Big Data), Dec 2017, pp. 1349–1356.

[88] *TensorFlow*, https://www.tensorflow.org/.

[89] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy, *Hive: A warehousing solution over a map-reduce framework*, Proc. VLDB Endow. **2** (2009), no. 2, 1626–1629.

[90] *The Transaction Processing Performance Council*, http://www.tpc.org.

[91] John W. Tukey, *Exploratory data analysis*, Addison-Wesley series in behavioral science : quantitative methods, 1977.

[92] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu, *The myria big data management and analytics system and cloud services*, CIDR, 2017.

[93] Alex Watson, Deepigha Shree Vittal Babu, and Suprio Ray, *Sanzu: A data science benchmark*, 2017 IEEE International Conference on Big Data (Big Data), Dec 2017, pp. 263–272.

[94] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, *Spark: Cluster computing with working sets*, USENIX HotCloud, 2010, pp. 10–10.

[95] Yi Zhang, Herodotos Herodotou, and Jun Yang, *RIOT: i/o-efficient numerical computing without SQL*, CoRR **abs/0909.1766** (2009).

Vita

Candidate's Full Name: Alex Watson
University Attended:

- UNB, Bachelor of Science in Computer Science

Publications:

1. The Falling of Momo: A Myo-Electric Controlled Game to Support Research in Prosthesis Training. Aaron Tabor, Alex Kienzle, Carly Smith, Alex Watson, Jason Wuertz, David Hanna. CHI PLAY Companion '16: Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts, Pages 71-77. ACM New York, NY, USA

2. Sanzu: A data science benchmark, A. Watson, D. S. V. Babu and Suprio Ray. 2017 IEEE International Conference on Big Data, pp. 263-272. December 2017. Boston, USA.

3. Alex Watson, Scott Bateman and Suprio Ray. PySnippet: Accelerating Exploratory Data Analysis in Jupyter Notebook through Facilitated Access to Example Code. International Workshop on Big Data Visual Exploration and Analytics (BigVis) (co-located with EDBT), Portugal 2019.

Conference Presentations:

1. Sanzu: A data science benchmark, A. Watson, D. S. V. Babu and Suprio Ray. 2017 IEEE International Conference on Big Data, December 2017. Boston, USA.

2. Alex Watson, Evaluating Systems for Data Science , 2017 Atlantic Universities, Math, Statistics and Computer Science, October 2017, Fredericton, UNB,